

Introducing jMusic

Andrew Sorensen

GET Systems
Chatswood, Sydney
Australia

Andrew R. Brown

Queensland University of Technology
Victoria Park Rd.
Kelvin Grove, 4059
Australia

Abstract

This paper introduces the jMusic compositional language. jMusic is a package of Java classes which provides an environment for non real time music composition. jMusic consists of a music data structure for event organisation along the lines of Common Music and HSML, however it relies more heavily than these on the common practice notation score metaphor in an attempt to provide easy access to composers familiar with that environment. The music data structure is practical for analysis as well as composition, and jMusic reads and writes standard MIDI files to facilitate interaction with existing computer-based music systems. jMusic includes audio classes for sound synthesis, and signal processing, along the lines of Csound and Cmix, however its object oriented nature and the integration between compositional and synthesis functions provide a platform for efficiently combining music-event and audio processes. jMusic is not a scripting language but a direct extension of the Java programming language. This allows access to the full functionality of Java, including the ability to write applets for Internet usage, and for applications to be run unchanged on a wide range of platforms (even with graphical interfaces). Learning jMusic is learning Java, so computer musicians can leverage to mutual advantage jMusic and Java experience. This paper outlines the jMusic data structure and built-in functionality, includes code examples, and shows how jMusic can be extended by users. jMusic is available for free public download.

Introduction

jMusic is a programming library written for musicians in the Java programming language. While still relatively new, we hope jMusic will be a library that is simple enough for new programmers (as many musicians are) but sophisticated enough to enable composers to accomplish comprehensive work, whatever form that may take. jMusic is designed to be used as a compositional medium, therefore it is primarily designed for musicians—rather than computer programmers.

jMusic is a music research project based at the Queensland University of Technology (QUT) music program in Brisbane, Australia. The project was begun in 1987 as part of Andrew Sorensen's Master of Arts degree program, at which time it focused on algorithmic composition using Markov chains. Since that time it has been rewritten and expanded to be a more generic system capable of supporting, or being extended to support, many compositional processes.

jMusic is designed to assist the compositional process by providing a structured and extendable environment for musical exploration. While its primary goal is to support composition, it has also been effectively used for live performance, musical analysis and computer music education. The jMusic data structure uses metaphors from established musical practice (scores and instruments) in order to provide familiarity at the early stages of learning, in keeping with Alan Kay's advice that "simple things should be simple and complex things should be possible." We hope that jMusic is simple to learn, but powerful to use. As well, the interoperability of jMusic with other music software is facilitated by the easy importing and exporting of MIDI files and audio files. This means that a users' current knowledge and tools are not discarded when working with jMusic.

jMusic and Java

Composing in jMusic is programming in Java. In this way jMusic is an library that extends Java similar to the way Common Music is a library that extends Lisp. This is in contrast to computer music systems that are meta-languages or scripting environments that are subsets of a general purpose language, as provided, for example, by Csound and Cmix/Minc in relation to C. When designing jMusic we felt that any additional complexity imposed by having to work directly in Java was outweighed by the advantage of the widespread availability of general Java support materials and the advantage to musicians of learning programming skills useful beyond the musical domain. However, we have made a concerted effort in the core jMusic classes to hide programming complexity, such as exception handling, and to provide useful scaffolding classes such as the MIDI and audio file handling.

Maintaining compatibility with the standard Java libraries ensures that jMusic benefits from the cross-platform independence of Java and that the more you know about Java programming the more useful jMusic will be to you. Learning jMusic can be a fun way to gain Java programming skills while focusing on making music. Currently, jMusic works in Java versions 1.1 and higher. While more recent versions of Java are available we have kept the core jMusic classes compatible with Java 1.1 to maintain compatibility with the widest possible user base, including Personal Java on PDAs, Applets in web Browsers, and operating systems without 'cutting edge' Java support. jMusic has full access to the Java language and support structures, jMusic applications can be as extensive as Java allows, which is increasingly extensive. Some jMusic packages use features more recent Java version, in particular the sound and MIDI libraries in Java 1.3.

jMusic is an open source package distributed under the GNU General Public Licence. It is being developed to support computer music making and the development of shared tools. We hope that this will encourage people to become a part of the jMusic community and contribute to, as well as benefit from, the development of jMusic. Any music created with jMusic is not itself covered by any licence and can be freely distributed or sold by the composer. Extensions to the jMusic language and application built using the jMusic libraries are subject to the Licence which, in short, requires that source code be freely available for other musicians to utilise and extend further.

Music, Composition, and the Computer

The marriage of music, composition and the computer may seem like an unlikely partnership, bringing together the arts, mathematics and the sciences in what many would see as an unholy trinity. It is not however as unnatural as it might at first appear. Many great scientists have also been exceptional artists and vice versa, the most famous of whom, arguably, is Leonard Da Vinci. Music and technology have in fact had an intimate relationship throughout history with technological advances aiding music in many diverse ways, examples include the theoretical understanding of frequency relations contributing to numerous tuning systems, the mechanical developments allowing more precise instrument manufacture, and the impact of the printing press and phonograph on music distribution.

What makes computer music significantly different from previous technological developments is that the use of computers in music making affects each process; composition, instrument building, performance, and distribution. Particularly relevant to jMusic is that computers have the ability to radically change the way in which we compose due to their ability to be programmed.

Computer developments have been central to developments in a twentieth century renowned for significant technological innovation. The influence of the computer has spread beyond the confines of the office to include activities such as sport, entertainment, and the arts. In these roles, the digital computer is often used to provide a means for modelling the complexity of real world patterns and associations. Machine abstractions of the world are becoming increasingly common, providing humanity with artificial representations of real world events and phenomena within digital micro-domains. The potential for designing machine abstractions of artistic structures has driven many artists to explore digital micro-domains as a means to create digital tools designed to investigate structural formalisms. At its heart, jMusic builds on a common practice notation model, but extends it and encourages it to be modified and expanded further.

Metaphors for music systems (separate article?)

- Mixing console metaphor - Javasound

- Tape recorder metaphor – MIDI Sequencers
- Flow chart metaphor – Music V, PatchWork, OpenMusic, Max
- Instrumental Performance metaphor – Csound, jMusic, Music Kit, Common music

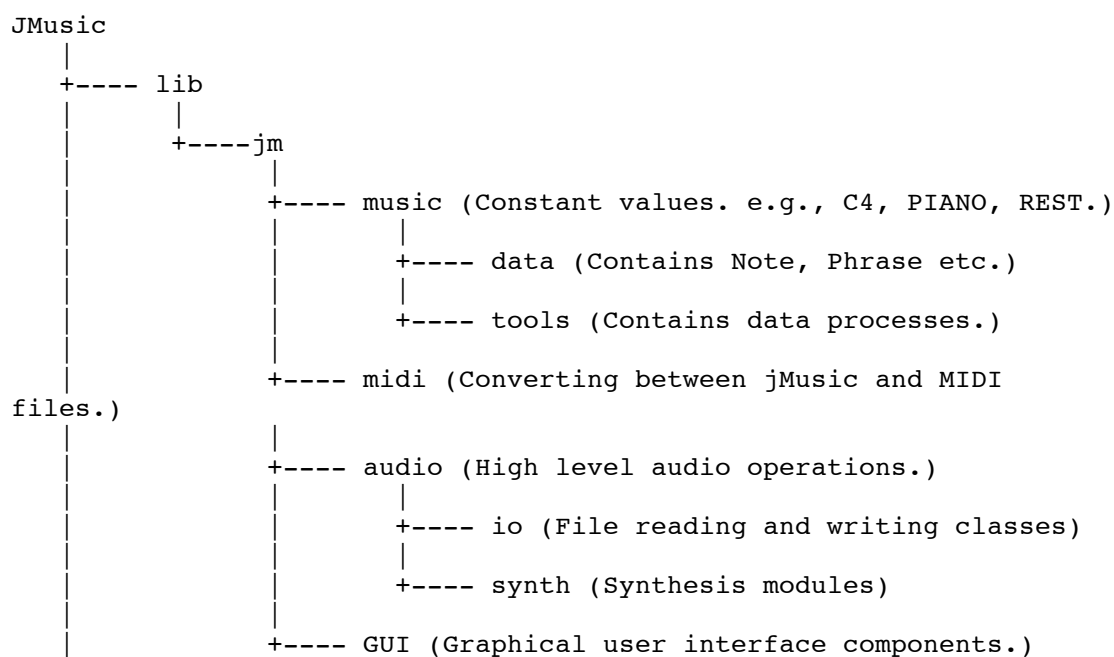
It seems to us that the function of the computer in music composition is to expose aspects of the music. That is, to assist structuring and to give form to the musical ideas of the composer, to reveal those ideas, not unlike the function of a score. Further, the computer also exposes the musical assumptions of the composer through the choice of representation system and associated methods of representational manipulation. In this regard the computer is implicated by requiring the composer to specify their musical intentions. The computer representation of music can also reveal, to the composer, potential in the material for further development. Additionally, the computer as digital sound medium creates or renders the sonic attributes of the composition highlighting certain characteristics over others. We hope that jMusic can provide a link from the musical past and present to the musical future, by providing familiar structures and functions at the outset and allowing the more experienced jMusic composer to evolve new musical structures, ideas, and processes. We seek to provide, with jMusic, a pathway for the evolution of musical ideas rather than a radical break from them.

Some jMusic Basics

jMusic is currently a non real time environment. The compositional process follows a cycle, familiar to computer music composers, of write, compile, listen, rewrite. In support of a non real time process we could begin by giving a summation of composition over the centuries and all the non real time work that has been completed before now, but suffice it to say that non real time composing has had a successful run of it up until now, and there is still an argument that says non real time has benefits that outweigh its disadvantages. It is clear that jMusic will become almost-real-time through direct Quicktime playback, we have work is under way for real time Java Applet support, and in Java version 1.3 there are realtime MIDI and audio libraries which jMusic will utilise. At present jMusic outputs music in standard MIDI file format (SMF) or audio file (currently only .au format) for playback in other applications.

Packages

The classes that make up jMusic are organised into core classes in the lib directory (those kept under the jm directory are the very core classes) and additional classes are organised in surrounding directories such as util(ities), app(lication)s, and inst(ruments). The distinction is made to keep separate the *development* and *use* of jMusic. In general, users will program new compositional utilities, instruments, and applications without being concerned with the core classes in the lib directory.



```
|
+---- inst (jMusic audio instruments.)
|
+---- jmUtil (Composing helper classes, e.g. Score Visualisation.)
|
+---- apps (Complete programs that use jMusic.)
|
+---- demos (Example code.)
|
+---- doc (JavaDoc html files with the details of jMusic classes.)
```

The compositional process

The basic process behind non real time computer music composition, including jMusic follows these steps:

- Decide what idea or material you want to compose or explore.
- Work out the compositional techniques you will need to use.
- Write a Java program describing your compositional techniques to the computer.
- Compile and run the program.
- Play back your new composition as a MIDI or audio file and reflect on the result.
- Cycle through this process until complete.

Step 3, writing the program, involves some more specific jMusic understanding which will now elaborate upon.

The basic sound objects in jMusic are called Notes. The traditional conception of notes, that they are discrete sound objects with several dimensions (pitch, duration, loudness . . .) holds true for jMusic. Composing in jMusic involves constructing and organising note objects. Notes can be collected as a sequence of events into a Phrase, a phrase is monophonic and continuous, and each phrase has a start time. Phrases that will sound using the same instrument can be collected into a Part. Several parts can be collected into a score. This structure is outlined in more detail below.

Time in jMusic is measured in beats. Notes have, independently, a rhythmValue (crotchet, quaver . . .) and a duration, allowing for gaps between notes (staccato) or overlapping notes. Working in clock time in jMusic is a simple process of calculating beats per minute into seconds (working at 60 bpm makes this trivial as one beat = one second).

The pitch of a note is measured in one of two ways, as MIDI pitch numbers (i.e., middle C = 60) or as frequency for audio output (i.e., 440hz). The loudness of a note, its dynamic, is measured (like MIDI) as an integer value from 1 to 127. The timbre of a note depends on which part it is allocated to, and the 'instrument' that is associated with that part.

jMusic and other systems?

jMusic is most similar in conception and design to Common Music (Taube ????) with major differences being the use of the Java programming language rather than Lisp, and jMusic's CPN data structure metaphor rather than an idiosyncratic data structure. There are other non real time computer assisted composition languages with similar aims including PatchWork (Laurson 1996) and OpenMusic (Assayag & Agon 1996) both of which focus, like jMusic, on event-based structuring ahead of signal processing. These languages rely heavily on a graphical representation as a way of 'assisting' the composer. We feel that such interfaces are generally more limiting to structural possibilities than a text-based environment, however, jMusic does provide various visualisation tools for those who wish to use them. HMSL, Csound, Cmix and other systems provide score and instrument separation similar to jMusic, and most of those provide much more extensive audio processing features than jMusic currently has. This reflects a difference in priorities between event-based composition and signal processing and the maturity of the systems. These systems can be used in conjunction with jMusic. Using MIDI file output musicians may, for example, choose to do event generation in jMusic and convert the score to Csound for audio rendering. Presently, jMusic is extensively used in conjunction with MIDI sequencers and synthesizers in a two step process of writing then producing.

Music Data Structure

The core of the jMusic data structure are the five classes in the `jm.music.data` package contains four Java source code file:

`Note.java` - The `Note.java` file contains one class called `Note`.

`Phrase.java` - The `Phrase.java` file contains one class called `Phrase` (Phrases in jMusic are monophonic).

`CPhrase.java` - The `CPhrase.java` file contains one class called `CPhrase` (C stands for chord).

`Part.java` - The `Part.java` file contains one class called `Part`.

`Score.java` - The `Score.java` file contains one class called `Score`.

These classes form the backbone of the jMusic data structure which allows the composer to create, analyse, manipulate music. The next section looks at these classes in more detail.

The musical information in jMusic is stored in a hierarchical fashion based upon a conventional score on paper.

```
Score (Contains any number of Parts)
  |
  +---- Part (Contains any number of Phrases)
        |
        +---- Phrase (Contains any number of Notes.)
              |
              +---- Note (Holds information about a single
musical event.)
```

Notes

The `jm.music.data.Note` class defines the basic note structure used by jMusic. Note objects contain many attributes, including:

- Pitch - the note's pitch
- Dynamic - the loudness of the note
- RhythmValue - the length of the note (e.g., Crotchet)
- Pan - the notes position in the stereo (or more) spectrum.
- Duration - the length of the note in milliseconds
- Offset - an deviation from the 'normal' start time of the note

The `Note` class also has a collection of methods which carry out various manipulations on the data.

Phrases

The `Phrase` class is a little more complicated than note objects but can be simply explained as being voices. A piano part is a single part but can have multiple voices. Take a Bach Fugue as an obvious example. Phrase objects really only contain a single important attribute, a list of notes. Every `Phrase` object contains a list of notes that you can add to, subtract from and move all over the place. The list that does all this is called a *vector* and is a Java class found in the `util` package `java.util.Vector`. Phrases include:

- `noteList` – A vector of notes
- `startTime` – The beat position at which the phrase begins
- `title` – The name of the phrase

CPhrases

The `CPhrase` class allows the composer to construct homophonic musical structures easily; that is, have jMusic play chords. A chord is defined as a group of notes that share the same onset time and duration, but differ in other respects; in particular they have different pitches. The `CPhrase` class can be used just like the `Phrase` class, but behind the scenes `CPhrase` structures are converted into `Phrases`. This becomes important only when you work with relatively complex musical structures in jMusic.

Parts

`Part` is a class that, surprisingly enough, holds the notes (in phrases) to be played by an instrument. A part contains a vector of phrases. A part also has a title ("Violin 1" for example), a channel, and an

instrument (in MIDI, a program change number - in audio an index in the instrument array). Again don't worry about the details too much just take a look. Parts include:

- phraseList – A vector of phrases
- progChg – The instrument used for notes in this part
- title – The name of the phrase

Scores

The Score class represents the top level of our data structure and it contains a vector of parts, and also has a title (name). Scores include:

- partList – A vector of parts
- title – The name of the score

jMusic constants

The JMC class contains a collection of constants (unchangeable variables) which help the computer refer to musical events using musical language. In this paper we will introduced some key words which jMusic uses to refer to musical attributes. Constants allow your code to be more human readable and 'musical'. For example, you can specify a pitch using the constant C4 rather than the number 60. Behind the scenes Java translates the constant C4 to 60.

The constants are set up in the class called JMC and to use them you must import that class. You will notice that most jMusic code has this line toward the start: `import jm.JMC;`

Secondly, when the class is declared it needs to implement the JMC class so the constants can be used. Therefore class declarations in jMusic tend to look like this;

```
public final class AudioScale implements JMC{
```

There are currently jMusic constants for pitch, rhythmic values, dynamics, and MIDI program changes. In this paper we will examine each in turn.

Pitch

Note constants are written in pitch-class/octave notation, such that middle C is C4, the D above it is D4 and the B below it is B3. Accidentals are indicated with S for sharp and F for flat, e.g., C sharp above middle C is CS4 or DF4. The pitch constants are converted into integer numbers for absolute pitch as used in MIDI. C4 (middle C) = 60. The range is from G9 - CN1 (C negative one).

Pitch can also be specified as frequency for jMusic audio output. Middle C, for example, becomes a frequency of 261.63. The constants convert equally tempered pitches to frequency. The syntax is `FRQ[n]` where n is the MIDI note number to be converted.

Rhythmic Value

Full names of abbreviations for English and American terms are provided for most common rhythmic values. jMusic is based on a beat pulse where one beat is a value of 1.0 and all other rhythms are relative to that. A rhythmValue can have several constants that equal it: `CROTCHET`, `C`, `QUARTER_NOTE`, and `QN`.

Dynamic

jMusic allows integers from 0-127 to indicate the dynamic (loudness) of a note. The constants, shown below, specify some set levels using abbreviations for the common Italian terms for dynamics (pianissimo etc). For example, `SILENT = 0`, `MF = 70`, and `FFF = 120`;

Panning

There are jMusic constants for panning notes across the stereo image. jMusic supports more than two (stereo) outputs but the constants only support the normal two channel output. Here are a few of the constants which relate the panning of a note: `PAN_CENTRE = 0.5`, `PAN_LEFT = 0.0`, `PAN_RIGHT = 1.0`;

Duration Articulation

In many cases jMusic notes will have a duration which relates to the rhythmicValue. By default this is 85% of the rhythmic value. So a crotchet note will have a rhythmicValue of 1.0 and a duration of 0.85. The duration constants are designed to be multiplied by the rhythmicDuration (CROTCHET * STACCATO), that way they can be applied to any rhythmicValue. There are a few common Italian terms for articulation defined as constants for the duration of a note. For example, STACCATO = 0.2, LEGATO = 0.95.

Timbre (MIDI Program Change)

The General MIDI set of program changes are included as jMusic constants. Each constant equates to a integer number from 1-127. These will only be relevant when playing back MIDI files generated from jMusic on General MIDI equipment (such as PC sound cards or Quicktime Musical Instruments). Most orchestral and rock band instruments are part of the GM set and you should be lucky enough to guess the name in most cases as several alternate wordings are provided for many instruments. For example, PIANO = 0, EPIANO = 4 or ELECTRIC_PIANO = 4 or ELPIANO = 4.

Code syntax

JMusic follows all the convention of Java syntax, so there is little need to cover that in detail here. Below is an example of a class that generates a jMusic score of one octave ascending chromatic scale, and outputs it as both a MIDI file and .au audio file using a simple triangle waveform oscillator.

```
import jm.JMC;
import jm.music.data.*;
import jm.midi.*;
import jm.audio.*;
import jm.audio.synth.*;
import jmInst.*;

public final class Scale implements JMC{
    public static void main(String[] args){
        Score s = new Score("JMDemo - Audio Scale");
        Part p = new Part("Flute", 1, 0);
        Phrase phr = new Phrase(0.0);

        TriangleInst ti = new TriangleInst(44100);
        Instrument[] ensemble = {ti};

        for(short i=0;i<12;i++){
            Note n = new Note(i+60, Q, 127, 0.5);
            phr.addNote(n);
        }

        p.addPhrase(phr);
        s.addPart(p);
        s.writeMIDI("Scale.mid");
        s.writeAU("Scale.au", ensemble);
    }
}
```

Standard Methods

As well as a data structure for music, jMusic provides a collection of methods (procedures) for operations on the data. Each class—Note, Phrase, CPhrase, Part, and Score—has methods for acting on its data.

Get and Set

There has been a deliberate attempt to make the jMusic classes operate and use syntax similar to the Java libraries. In keeping with this idea each of the class attributes can be retrieved with a method call starting with 'get'. As in getPitch for the Note class. To define the value of object attributes there are methods beginning with 'set'. As in setStartTime for a Phrase.

Manipulations

There are methods for making structural changes to the jMusic data. These aim to be fundamental processes from which the user can construct more complex methods. For example, the user might combine copy and transpose methods to effect a melodic sequence. Some of the methods in the music.data package include:

- copy
- repeat
- size
- getEndTime
- fadeIn
- fadeOut
- append
- empty
- compress

Audio Instruments

There are a whole set of audio classes in jMusic. They are arranged into three subdirectories, each of which should be imported. The jm.audio.data directory contains the audio structure information and supports reading and writing audio files. The jm.audio.synth directory contains various synthesis functions and the oscillator class. The jmInst directory contains the jMusic Instruments. The first order of audio business is to declare an instance of the Instruments used to play the jMusic score. In the code above this involves creating an instance of the TrangleInst which produces a simple triangle waveform. The instruments need to be collected into an array. This array is called 'ensemble' in the example, and only contains one instrument. More complex scores may use several instruments.

Creating an audio instrument involves extending the Instrument class and by chaining together audio components, either built-in processes or those developed by the user. For example, the SimpleSampleInst class is chain of the following audio processes;

- AUIn
- resample
- envelope
- sampleOut

The details of creating audio modules and processes is beyond the scope of this paper, but is not dissimilar to creating instruments in Csound or Supercollider.

Obtaining jMusic

JMusic is available for download from the jMusic web site: <http://jmusic.ci.qut.edu.au>.

The web site contains installation instructions and an extensive range of tutorials to help users get started. In addition the web site has a list of suggested references on computer assisted composition, related topics, links to similar computer music projects, and examples of music created with jMusic.

Conclusion

In this paper we have provided an overview of the jMusic language. In doing so we have described the aims of jMusic and how it relates to other computer music languages. The data structure and other aspects of jMusic have been introduced and some of the main issues in the jMusic composing process have been explored. The jMusic project continues to develop, and we hope that more musicians may make use of jMusic and that their feedback and coding efforts might contribute to its ongoing

development. Most of all we hope that jMusic might be a pathway for many musicians to explore algorithmic composition and that plenty of interesting music will result.

References

Assayag, G. & Agon, C. 1996. "Open Music Architecture." In *Proceeding of the 1996 International Computer Music Conference, Hong Kong*. San Francisco: ICMA. pp.339-340

Laurson, Mikael. 1996. *Patchwork: A visual programming language and some musical applications*. *Studia Music 6*. Helsinki: Finland.