Alda

A Music Programming Language Built With Clojure

Clojure Remote 2016

Dave Yarwood

@daveyarwood

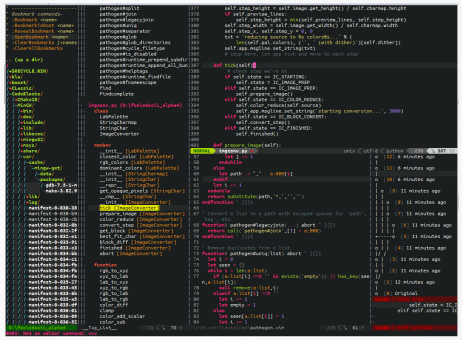
adzerk

This Talk

- Background on Music Programming Languages
- Basic introduction to Alda
- Broad overview of the "moving parts"
- Alda + Clojure = <3
- Dave's secret plot to lure contributors

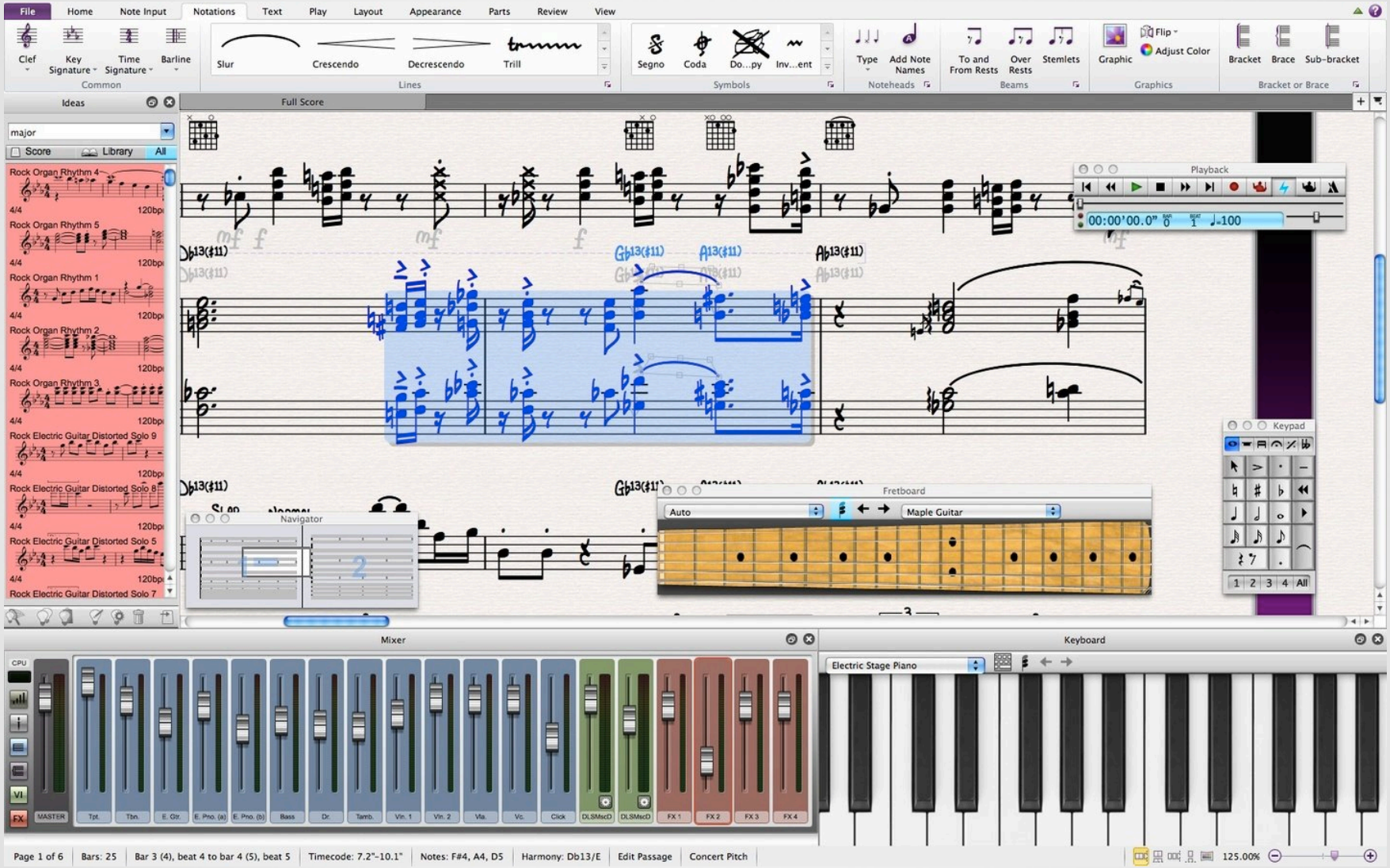
Music programming?

+



=

Sibelius: a graphical score editor



List of audio programming languages

From Wikipedia, the free encyclopedia

This article's **lead section** may not adequately **summarize** key points of its contents. Please c
accessible overview of all important aspects of the article. Please discuss this issue on the article's

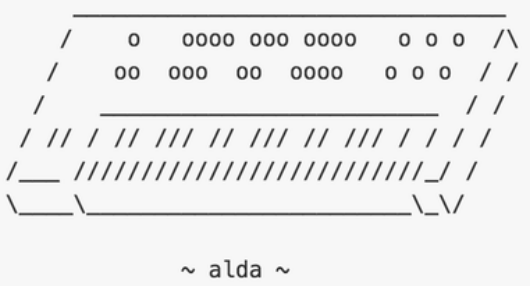
This article **does not cite any sources**. Please help improve this article by adding citations to relia
challenged and removed. *(January 2015)*

This is a **list of audio programming languages** including languages optimized for sound production, algorithmic composition, and

- abc, a language for notating music using the ASCII character set
- Alda, a music programming language for musicians (omg we're famous!)
- ChucK, strongly timed, concurrent, and on-the-fly audio programming language
- Cmix, Real-time Cmix, a MUSIC-N synthesis language somewhat similar to Csound
- CMusic
- Common Lisp Music (CLM), a music synthesis and signal processing package in the Music V family
- Csound, a MUSIC-N synthesis language released under the LGPL with many available unit generators
- Extempore, a live-coding environment which borrows a core foundation from the Impromptu environment
- FAUST, Functional Audio Stream, a functional compiled language for efficient real-time audio signal processing
- Hierarchical Music Specification Language (HMSL), optimized more for music than synthesis, developed in the 1980s in Forth
- Impromptu, a Scheme language environment for Mac OS X capable of sound and video synthesis, algorithmic composition, and
- jMusic
- JSyn
- Kyma (sound design language)
- Max/MSP The "lingua franca" for developing interactive music performance software
- Music Macro Language (MML), often used to produce chiptune music in Japan
- MUSIC-N, includes versions I, II, III, IV, IV-B, IV-BF, V, 11, and 360
- Nyquist
- OpenMusic
- Patchblocks
- Pure Data
- Reaktor
- Structured Audio Orchestra Language (SAOL), part of the MPEG-4 Structured Audio standard

README.md

github.com/alda-lang/alda



a music programming language for musicians

```
[alda "1.0.0-rc3"]          @clojars.org
```

Installation | **Docs** | **Changelog** | **Contributing** (In Progress: 3)

New to Alda? You may be interested in reading this blog post as an introduction.

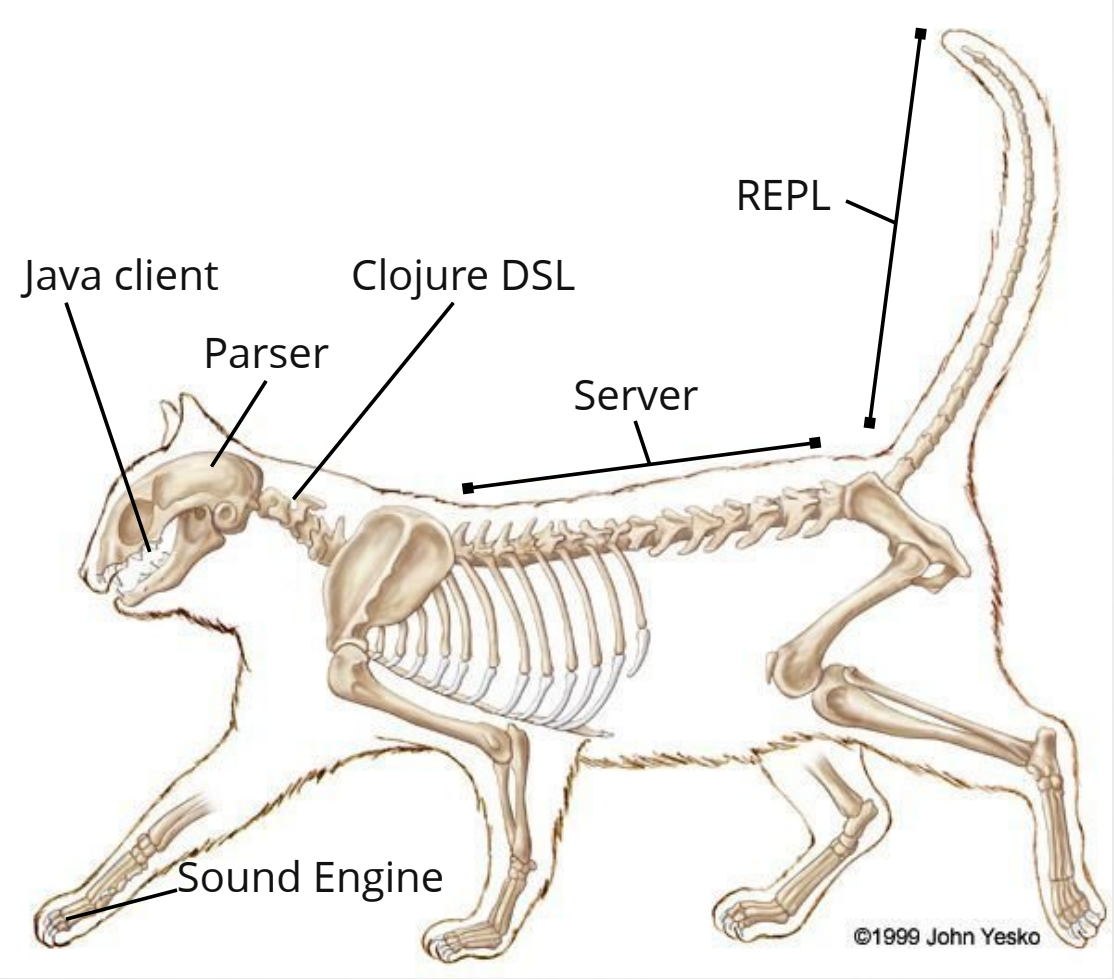
Inspired by other music/audio programming languages such as PPMCK, LilyPond and ChucK, Alda aims to be a powerful and flexible programming language for the musician who wants to easily compose and generate music on the fly, using naught but a text editor. Alda is designed in a way that equally favors aesthetics, flexibility and ease of use, with (eventual) support for the text-based creation of all manner of music: classical, popular, chiptune, electroacoustic, and more!

Features

- Easy to understand, markup-like syntax
- Perfect for musicians who don't know how to program and programmers who don't know how to music

DEMO TIME

Alda: Basic Anatomy



The Alda Client

```
$ alda up
[27713] Starting Alda server...
[27713] Server up ✓
```

Java/Clojure inter-op

Utility function for calling a Clojure function in the same project:

```
public static void callClojureFn(String fn, Object... args) {
  Symbol var = (Symbol)Clojure.read(fn);
  IFn require = Clojure.var("clojure.core", "require");
  require.invoke(Symbol.create(var.getNamespace()));
  ISeq argsSeq = ArraySeq.create(args);
  Clojure.var(var.getNamespace(), var.getName()).applyTo(argsSeq)
}
```

Using it to start an Alda server:

```
Util.callClojureFn("alda.server/start-server!", args);
```

The Alda Sound Engine

Data Representation of a Note

```
{:offset 1000.0, :instrument "piano-ERbWJ", :volume 1.0,
 :track-volume 0.7874015748031497, :panning 0.5, :midi-note 64
 :pitch 329.6275569128699, :duration 450.0}
```

- Instrument ID
- Pitch (frequency in Hz)
  - MIDI note (0-127)
- Duration (ms)
  - Note length (quarter, eighth, etc.)
  - Tempo (beats per minute)
- Offset (ms)
- Volume (0.0-1.0)
- Panning (0.0-1.0)

Data Representation of a Score

```
{:events
 #{{:offset 1000.0, :instrument "piano-ERbWJ", :volume 1.0,
    :track-volume 0.7874015748031497, :panning 0.5, :midi-note 64,
    :pitch 329.6275569128699, :duration 450.0}
   {:offset 500.0, :instrument "piano-ERbWJ", :volume 1.0,
    :track-volume 0.7874015748031497, :panning 0.5, :midi-note 62,
    :pitch 293.6647679174076, :duration 450.0}
   {:offset 0, :instrument "piano-ERbWJ", :volume 1.0,
    :track-volume 0.7874015748031497, :panning 0.5, :midi-note 60,
    :pitch 261.6255653005986, :duration 450.0}},
 :markers {:start 0},
 :instruments
 {"piano-ERbWJ"
  {:octave 4, :current-offset {:offset 1500.0}, :key-signature {},
   :config {:type :midi, :patch 1}, :duration 1, :volume 1.0,
   :last-offset {:offset 1000.0}, :id "piano-ERbWJ", :quantization 0.9
   :tempo 120, :panning 0.5, :current-marker :start,
   :stock "midi-acoustic-grand-piano",
   :track-volume 0.7874015748031497}}}
```

JSyn

Home
Products
JSyn
pForth
Music
News
Links
Contact Us
About Us

JSyn | Documentation | Examples | Download | Beta | Support | Apps | Press | Events |

JSyn - Audio Synthesis Software API for Java

JSyn allows you to develop interactive computer music programs in Java. You can run them as stand-alone applications, or as Applets in a web page. JSyn can be used to generate sound effects, audio environments, or music. JSyn is based on the traditional model of unit generators which can be connected together to form complex sounds. For example, you could create a wind sound by connecting a white noise generator to a low pass filter that is modulated by a random contour generator.

softsynth.com/jsyn

Software Features

- Real-time, high fidelity audio synthesis software.
- Library of unit generators including oscillators, filters, envelopes, noise generators, effects.
- Most internal operations use 64 bit floating point values.
- Audio sample playback can be combined with other synthesis and processing units.
- Easy to use Java classes for creating, connecting and controlling unit generators.
- Time-stamping to allow scheduling of control events for rock solid timing.
- Sample and envelope data queuing supports flexible looping and splicing.
- Audio input support for voice recording and processing.
- JSyn can be used from a Java Application or as an Applet in a web browser.
- A number of example Applets are provided that demonstrate these features.
- Documentation includes JavaDocs, and a slide presentation from ICMC'98
- Visit our Developer Page for the info you need to develop JSyn programs and put them in a web page.

Follow JSyn on Google+

Software License

- JSyn is released under the Apache License V2.

javax.sound.midi

Package javax.sound.midi

Provides interfaces and classes for I/O, sequencing, and synthesis of MIDI (Musical Instrument Digital Interface) data.

See: Description

Interface Summary

Interface	Description
ControllerEventListener	The `ControllerEventListener` interface should be implemented by classes whose instances need to be notified when a `Sequencer` has processed a requested type of MIDI control-change event.
MetaEventListener	The `MetaEventListener` interface should be implemented by classes whose instances need to be notified when a **`Sequencer`** has processed a **`MetaMessage`**.
MidiChannel	A `MidiChannel` object represents a single MIDI channel.
MidiDevice	`MidiDevice` is the base interface for all MIDI devices.
MidiDeviceReceiver	`MidiDeviceReceiver` is a `Receiver` which represents a MIDI input connector of a `MidiDevice` (see **`MidiDevice.getReceiver()`**).
MidiDeviceTransmitter	`MidiDeviceTransmitter` is a `Transmitter` which represents a MIDI input connector of a `MidiDevice` (see **`MidiDevice.getTransmitter()`**).
Receiver	A `Receiver` receives **`MidiEvent`** objects and typically does something useful in response, such as interpreting them to generate sound or raw MIDI output.
Sequencer	A hardware or software device that plays back a MIDI **`sequence`** is known as a *sequencer*.
Soundbank	A `Soundbank` contains a set of `Instruments` that can be loaded into a `Synthesizer`.
Synthesizer	A `Synthesizer` generates sound.
Transmitter	A `Transmitter` sends **`MidiEvent`** objects to one or more **`Receivers`**.

Class Summary

Class	Description
Instrument	An instrument is a sound-synthesis algorithm with certain parameter settings, usually designed to emulate a specific real-world musical instrument or to achieve a specific sort of sound effect.
MetaMessage	A `MetaMessage` is a **`MidiMessage`** that is not meaningful to synthesizers, but that can be stored in a MIDI file and interpreted by a sequencer program.
MidiDevice.Info	A `MidiDevice.Info` object contains assorted data about a **`MidiDevice`**, including its name, the company who created it, and descriptive text.
MidiEvent	MIDI events contain a MIDI message and a corresponding time-stamp expressed in ticks, and can represent the MIDI event

alda.lisp
(The Alda Clojure DSL)

alda.lisp

```
(score
  (part {:names ["piano"]}
    (note (pitch :c) (duration (note-length 8)))
    (note (pitch :e))
    (note (pitch :g))
    (chord
      (note (pitch :d) (duration (note-length 2 {:dots 1})))
      (note (pitch :f))
      (note (pitch :a)))))
```

```
alda.parser=> (parse-input "piano: c8 e g d2./f/a")

(alda.lisp/score
  (alda.lisp/part {:names ["piano"]}
    (alda.lisp/note (alda.lisp/pitch :c)
                    (alda.lisp/duration (alda.lisp/note-length 8)))
    (alda.lisp/note (alda.lisp/pitch :e))
    (alda.lisp/note (alda.lisp/pitch :g))
    (alda.lisp/chord
      (alda.lisp/note (alda.lisp/pitch :d)
                      (alda.lisp/duration (alda.lisp/note-length 2 {:dots 1})))
      (alda.lisp/note (alda.lisp/pitch :f))
      (alda.lisp/note (alda.lisp/pitch :a)))))
```

alda.now

```
(require '[alda.lisp :refer :all])
(require '[alda.now  :refer (set-up! play!)])

(score*)
(part* "upright-bass")

; This is optional. If left out, Alda will set up the MIDI synth the firs
; time you tell it to play something.
(set-up! :midi)

(play!
  (octave 2)
  (note (pitch :c) (duration (note-length 8)))
  (note (pitch :d))
  (note (pitch :e))
  (note (pitch :f))
  (note (pitch :g) (duration (note-length 4))))
```

Inline Clojure Code

entropy.alda

```
(def REST-RATE 0.15)
(def MS-LOWER 30)
(def MS-UPPER 3000)
(def MAX-OCTAVE 8)

(defn random-note
  "Plays a random note in a random octave,
   for a random number of milliseconds.

   May randomly decide to rest, instead,
   for a random number of milliseconds."
  []
  (let [ms (ms (rand-nth (range MS-LOWER MS-UPPER)))
    (if (< (rand) REST-RATE)
      (pause (duration ms))
      (let [o (rand-int (inc MAX-OCTAVE))
            n [(keyword (str (rand-nth "abcdefg")))
               (rand-nth [:sharp :flat :natural])]]
       (octave o)
       (note (apply pitch n) (duration ms))))))
```

```
# continued from left

midi-electric-piano-1:
  (panning 25)
  (random-note) * 20

midi-timpani:
  (panning 50)
  (random-note) * 20

midi-celesta:
  (panning 75)
  (random-note) * 20
```

What's next?

- Microtonal music
- Waveform synthesis
- Variables

```
riffA = f8 f g+ a > c c d c <
riffB = b-8 b- > c+ d f f g f <
riffC = > c8 c d+ e g g a g <
riffD = f8 f g+ a > c c d < b > | c c < b- b- a a g
```

```
riffA # expands to "f8 f g+ a > c c d c <"
```

```
riffA*4
riffB*2 riffA*2   # or riffB** riffA**
riffC riffB riffD
```

```
rockinRiff = [
  riffA*4
  riffB*2 riffA*2
  riffC riffB riffD
]
```

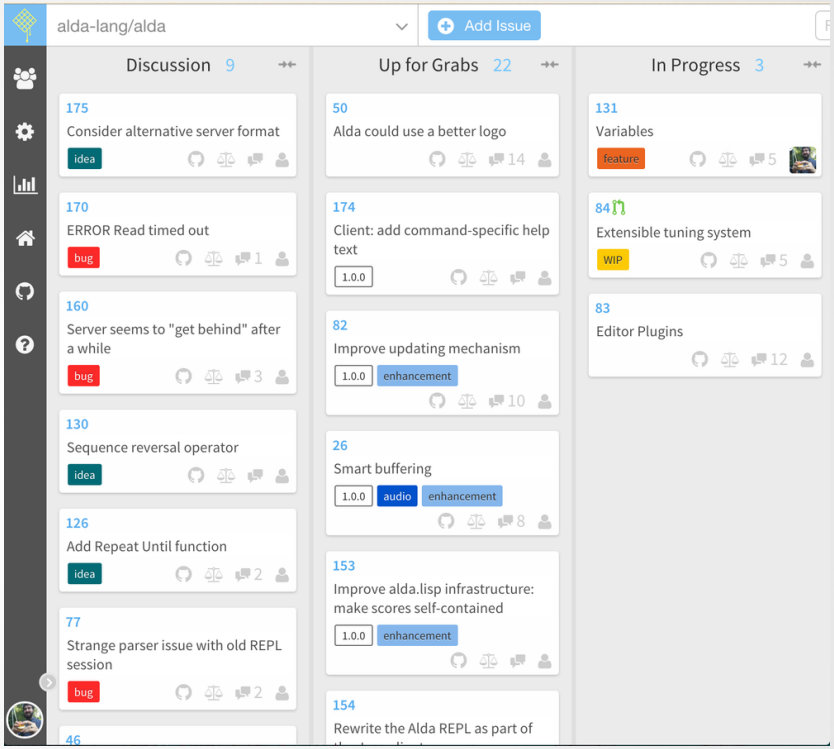
```
guitar/saxophone:
    rockinRiff*9999999
```

What's next?

- Modules
- Export to MusicXML, MIDI, LilyPond, etc.
- Import from MusicXML, MIDI, LilyPond, etc.
- Export audio to file
- Plugin system

THANKS

waffle.io/alda-lang/alda



Try Alda

github.com/ alda-lang / alda

Want to get in on this?

We could use your help!

Slide graveyard

I over-planned and ended up with way more slides than I had time to present.

Stashing the slides I got rid of here.

Music Macro Language (MML)

(Classical MML, est. 1978)

```
1 PRINT "TOORYANSE"
2 PRINT "ARRANGED BY"
3 PRINT " (C)2012 MOTOI KENKICHI"
4 PRINT " THANKS ALL WIKIPEDIANS."
10 TEMPO 4
20 A$="E5R1E3R0D3R0E3R0E1R0D1R0-G4R1"
30 B$="F3R0F1R0F1R0A3R0F1R0E1R0D1R0D1R0E5R0"
40 C$="C3R0C1R0C1R0E3R0C1R0-B1R0C1R0-B1R0-A1R0-A1-B5R0"
50 D$="E1R0E1R0E1R0E1R0E1R0E1R0D1R0E1R0E1R0E1R0D1R0-A1R0-A1R0B3R1"
60 E$="-A1R0-B1R0C1R0D1R0E1R0F1R0E1R0F3R1A3R1B1R0A1R0F3R0E3R0E1R0E4R0
100 MUSIC A$+B$+B$
110 MUSIC C$+C$+B$
120 MUSIC C$+D$+E$
```

Music Macro Language (MML)

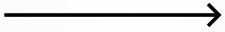
(Modern MML, est. 2001)

```
#TITLE My First NES Chip
#COMPOSER Nullsleep
#PROGRAMER 2003 Jeremiah Johnson

@v0 = { 10 9 8 7 6 5 4 3 2 }
@v1 = { 15 15 14 14 13 13 12 12 11 11 10 10 9 9 8 8 7 7 6 6 }
@v2 = { 15 12 10 8 6 3 2 1 0 }
@v3 = { 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 }

ABCDE t150

A l8 o4 @01 @v0
A [c d e f @v1 g4 @v0 a16 b16 >c c d e f @v1 g4 @v0 a16 b16 >c<<]2

C l4 o3 q6
C [c e g8 g8 a16 b16 >c8 c e g8 g8 a16 b16 >c8<<]4

D l4 o1 @0 @v2
D [@v2 b @v3 e @v2 b @v3 e @v2 b @v3 e @v2 b @v3 e8 @v2 b8]4
```

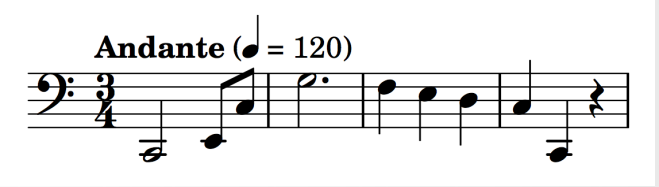
LilyPond

my_score.ly

```
\relative c, {
  \clef "bass"
  \time 3/4
  \tempo "Andante" 4 = 120
  c2 e8 c'
  g'2.
  f4 e d
  c4 c, r
}
```



my_score.pdf



Csound

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs  = 1

instr 1

iflg = p4
asig oscils .7, 220, 0, iflg
     outs asig, asig

endin
</CsInstruments>
<CsScore>
i 1 0 2 0
i 1 3 2 2
e
</CsScore>
</CsoundSynthesizer>
```

Using JSyn to schedule events

```
(import '[com.softsynth.shared.time TimeStamp ScheduledCommand]
        '[com.jsyn.engine SynthesisEngine])

(def engine
  (doto (SynthesisEngine.) .start))
```

Define events:

```
(defn event [f]
  (reify ScheduledCommand
    (run [_]
      (f))))

(def hello-event   (event #(println "hello")))
(def clojure-event (event #(println "clojure")))
(def remote-event  (event #(println "remote")))
```

Schedule events:

```
(let [start (.getCurrentTime engine)]
  (.scheduleCommand engine (TimeStamp. (+ start 1.0)) hello-event)
  (.scheduleCommand engine (TimeStamp. (+ start 1.5)) clojure-event
  (.scheduleCommand engine (TimeStamp. (+ start 2.0)) remote-event)
```

javax.sound.midi

Package javax.sound.midi

Provides interfaces and classes for I/O, sequencing, and synthesis of MIDI (Musical Instrument Digital Interface) data.

See: Description

Interface Summary

Interface	Description
ControllerEventListener	The `ControllerEventListener` interface should be implemented by classes whose instances need to be notified when a `Sequencer` has processed a requested type of MIDI control-change event.
MetaEventListener	The `MetaEventListener` interface should be implemented by classes whose instances need to be notified when a **`Sequencer`** has processed a **`MetaMessage`**.
MidiChannel	A `MidiChannel` object represents a single MIDI channel.
MidiDevice	`MidiDevice` is the base interface for all MIDI devices.
MidiDeviceReceiver	`MidiDeviceReceiver` is a `Receiver` which represents a MIDI input connector of a `MidiDevice` (see **`MidiDevice.getReceiver()`**).
MidiDeviceTransmitter	`MidiDeviceTransmitter` is a `Transmitter` which represents a MIDI input connector of a `MidiDevice` (see **`MidiDevice.getTransmitter()`**).
Receiver	A `Receiver` receives **`MidiEvent`** objects and typically does something useful in response, such as interpreting them to generate sound or raw MIDI output.
Sequencer	A hardware or software device that plays back a MIDI **`sequence`** is known as a *sequencer*.
Soundbank	A `Soundbank` contains a set of `Instruments` that can be loaded into a `Synthesizer`.
Synthesizer	A `Synthesizer` generates sound.
Transmitter	A `Transmitter` sends **`MidiEvent`** objects to one or more **`Receivers`**.

Class Summary

Class	Description
Instrument	An instrument is a sound-synthesis algorithm with certain parameter settings, usually designed to emulate a specific real-world musical instrument or to achieve a specific sort of sound effect.
MetaMessage	A `MetaMessage` is a **`MidiMessage`** that is not meaningful to synthesizers, but that can be stored in a MIDI file and interpreted by a sequencer program.
MidiDevice.Info	A `MidiDevice.Info` object contains assorted data about a **`MidiDevice`**, including its name, the company who created it, and descriptive text.
MidiEvent	MIDI events contain a MIDI message and a corresponding time-stamp expressed in ticks, and can represent the MIDI event

javax.sound.midi

javax.sound.midi

Interface Synthesizer

All Superinterfaces:

AutoCloseable, MidiDevice

```
public interface Synthesizer
extends MidiDevice
```

A `Synthesizer` generates sound. This usually happens when one of the `Synthesizer`'s `MidiChannel` objects receives a `noteOn` message, either directly or via the `Synthesizer` object. Many `Synthesizer`s support `Receivers`, through which MIDI events can be delivered to the `Synthesizer`. In such cases, the `Synthesizer` typically responds by sending a corresponding message to the appropriate `MidiChannel`, or by processing the event itself if the event isn't one of the MIDI channel messages.

The `Synthesizer` interface includes methods for loading and unloading instruments from soundbanks. An instrument is a specification for synthesizing a certain type of sound, whether that sound emulates a traditional instrument or is some kind of sound effect or other imaginary sound. A soundbank is a collection of instruments, organized by bank and program number (via the instrument's `Patch` object). Different `Synthesizer` classes might implement different sound-synthesis techniques, meaning that some instruments and not others might be compatible with a given synthesizer. Also, synthesizers may have a limited amount of memory for instruments, meaning that not every soundbank and instrument can be used by every synthesizer, even if the synthesis technique is compatible. To see whether the instruments from a certain soundbank can be played by a given synthesizer, invoke the `isSoundbankSupported` method of `Synthesizer`.

"Loading" an instrument means that that instrument becomes available for synthesizing notes. The instrument is loaded into the bank and program location specified by its `Patch` object. Loading does not necessarily mean that subsequently played notes will immediately have the sound of this newly loaded instrument. For the instrument to play notes, one of the synthesizer's `MidiChannel` objects must receive (or have received) a program-change message that causes that particular instrument's bank and program number to be selected.

See Also:

`MidiSystem.getSynthesizer()`, `Soundbank`, `Instrument`, `MidiChannel.programChange(int, int)`, `Receiver`, `Transmitter`, `MidiDevice`

Nested Class Summary

Nested classes/interfaces inherited from interface javax.sound.midi.MidiDevice
`MidiDevice.Info`

Method Summary

javax.sound.midi

javax.sound.midi

Interface MidiChannel

`public interface MidiChannel`

A `MidiChannel` object represents a single MIDI channel. Generally, each `MidiChannel` method processes a like-named MIDI "channel voice" or "channel mode" message as defined by the MIDI specification. However, `MidiChannel` adds some "get" methods that retrieve the value most recently set by one of the standard MIDI channel messages. Similarly, methods for per-channel solo and mute have been added.

A `Synthesizer` object has a collection of `MidiChannels`, usually one for each of the 16 channels prescribed by the MIDI 1.0 specification. The `Synthesizer` generates sound when its `MidiChannels` receive `noteOn` messages.

See the MIDI 1.0 Specification for more information about the prescribed behavior of the MIDI channel messages, which are not exhaustively documented here. The specification is titled `MIDI Reference: The Complete MIDI 1.0 Detailed Specification`, and is published by the MIDI Manufacturer's Association (http://www.midi.org).

MIDI was originally a protocol for reporting the gestures of a keyboard musician. This genesis is visible in the `MidiChannel` API, which preserves such MIDI concepts as key number, key velocity, and key pressure. It should be understood that the MIDI data does not necessarily originate with a keyboard player (the source could be a different kind of musician, or software). Some devices might generate constant values for velocity and pressure, regardless of how the note was performed. Also, the MIDI specification often leaves it up to the synthesizer to use the data in the way the implementor sees fit. For example, velocity data need not always be mapped to volume and/or brightness.

See Also:

`Synthesizer.getChannels()`

Method Summary

Methods

Modifier and Type	Method and Description
`void`	**`allNotesOff()`** Turns off all notes that are currently sounding on this channel.
`void`	**`allSoundOff()`** Immediately turns off all sounding notes on this channel, ignoring the state of the Hold Pedal and the internal decay rate of the current `Instrument`.
`void`	**`controlChange(int controller, int value)`** Reacts to a change in the specified controller's value.
`int`	**`getChannelPressure()`** Obtains the channel's keyboard pressure.
`int`	**`getController(int controller)`**

javax.sound.midi

boolean	**getMute**() Obtains the current mute state for this channel.
boolean	**getOmni**() Obtains the current omni mode.
int	**getPitchBend**() Obtains the upward or downward pitch offset for this channel.
int	**getPolyPressure**(int noteNumber) Obtains the pressure with which the specified key is being depressed.
int	**getProgram**() Obtains the current program number for this channel.
boolean	**getSolo**() Obtains the current solo state for this channel.
boolean	**localControl**(boolean on) Turns local control on or off.
void	**noteOff**(int noteNumber) Turns the specified note off.
void	**noteOff**(int noteNumber, int velocity) Turns the specified note off.
void	**noteOn**(int noteNumber, int velocity) Starts the specified note sounding.
void	**programChange**(int program) Changes a program (patch).
void	**programChange**(int bank, int program) Changes the program using bank and program (patch) numbers.
void	**resetAllControllers**() Resets all the implemented controllers to their default values.
void	**setChannelPressure**(int pressure) Reacts to a change in the keyboard pressure.
void	**setMono**(boolean on) Turns mono mode on or off.
void	**setMute**(boolean mute) Sets the mute state for this channel.
void	**setOmni**(boolean on) Turns omni mode on or off.
void	**setPitchBend**(int bend) Changes the pitch offset for all notes on this channel.
void	**setPolyPressure**(int noteNumber, int pressure) Reacts to a change in the specified note's key pressure.
void	**setSolo**(boolean soloState) Sets the solo state for this channel.

javax.sound.midi

Initialize a synth and get the first channel:

```
(import '[javax.sound.midi MidiSystem])

(def synth
  (doto (MidiSystem/getSynthesizer) .open))

(def channel
  (aget (.getChannels synth) 0))
```

Change patches, play notes:

```
(defn play-note [chan n]
  (.noteOn chan n 127)
  (Thread/sleep 300)
  (.noteOff chan n))

; set instrument to slap bass
(.programChange channel 36)

; slap a C major scale
(doseq [n [36 38 40 41 43 45 47 48]]
  (play-note channel n))
```