

# CrestMuse Toolkit (CMX) ver.0.61

## Manual

Tetsuro Kitahara

(College of Humanities and Sciences,  
Nihon University)

kitahara [at] kthrlab.jp

# Lesson 0 Overview

# What you can do with CMX

- CMX is a Java library that facilitates music processing
- CMX supports various music processing including:
  - Input/output of MusicXML, MIDI data, etc.
  - Realtime processing of MIDI and/or audio data
  - Automatic music generation based on Bayesian nets (weka.jar is needed)
- CMX is a library for programmers, not an application
- CMX can be used on various JVM languages (e.g. Processing, Groovy) as well as Java

# How to install it

- Extract the zip file, then copy cmx.jar and lib/\*.jar to the following directories:
  - ~/sketchbook/libraries/cmx/library for Processing
  - ~/.groovy/lib/ for Groovy
  - \$JAVA\_HOME/jre/lib/ext/ for Java

\$JAVA\_HOME stands for where the JDK is installed
- Instead, add all jar files to CLASSPATH
- install.sh automatically executes the installation  
(Note: Run as root. That is, “sudo ./install.sh” )

The above directories are for UNIX-style OSes.  
Check for your OS!

# Basic usages

- Way 1: Use CMXScript
- Way 2: Make a subclass of CMXApplet
- Way 3: Use the CMXController class
- Way 4: Use as a file converter from CLI

# Way 1: Use CMXScript

- Newly introduced scripting language
- A subclass of CMXApplet is automatically generated once setup() and draw() are coded
- Reasonably compatible to Processing

```
void setup() {  
    wavread("sample.wav")  
    playMusic()  
}  
  
void draw() {  
    // do nothing  
}
```

Run the script by:

```
$ cmxscript filename
```

# Way 2: Make a subclass of CMXApplet

- Define an original subclass of CMXApplet
- Suitable from Java, Groovy, etc.



```
import jp.crestmuse.cmx.processing.*

class MyApplet extends CMXApplet {
    void setup() {
        wavread("sample.wav")
        playMusic()
    }
    void draw() {
        // do nothing
    }
}

MyApplet.start("MyApplet")
```

See the Java Doc for CMXApplet for details

# Way 3: Use CMXController class

- Because ways 1 & 2 are unsuitable for Processing, CMXController can be used instead

```
import jp.crestmuse.cmx.processing.*;

CMXController cmx = CMXController.getInstance();

void setup() {
    cmx.wavread("sample.wav");
    cmx.playMusic();
}

void draw() {
    // do nothing
}
```



See the Java Doc of CMXController for details



# Way 4: Use as a file converter from CLI

- CMX converts files between MusicXML, SCCXML, MIDIXML, standard MIDI files from a command line:

```
$ cmx (Java's options) command (options)
```

- Example:

```
$ cmx smf2scc myfile.mid
```

Converts a standard MIDI file “myfile.mid” to SCCXML

```
$ cmx smf2scc myfile.mid -o mysccfile.xml
```

Converts to a standard MIDI file “myfile.mid” to SCCXML and writes it as “mysccfile.xml”

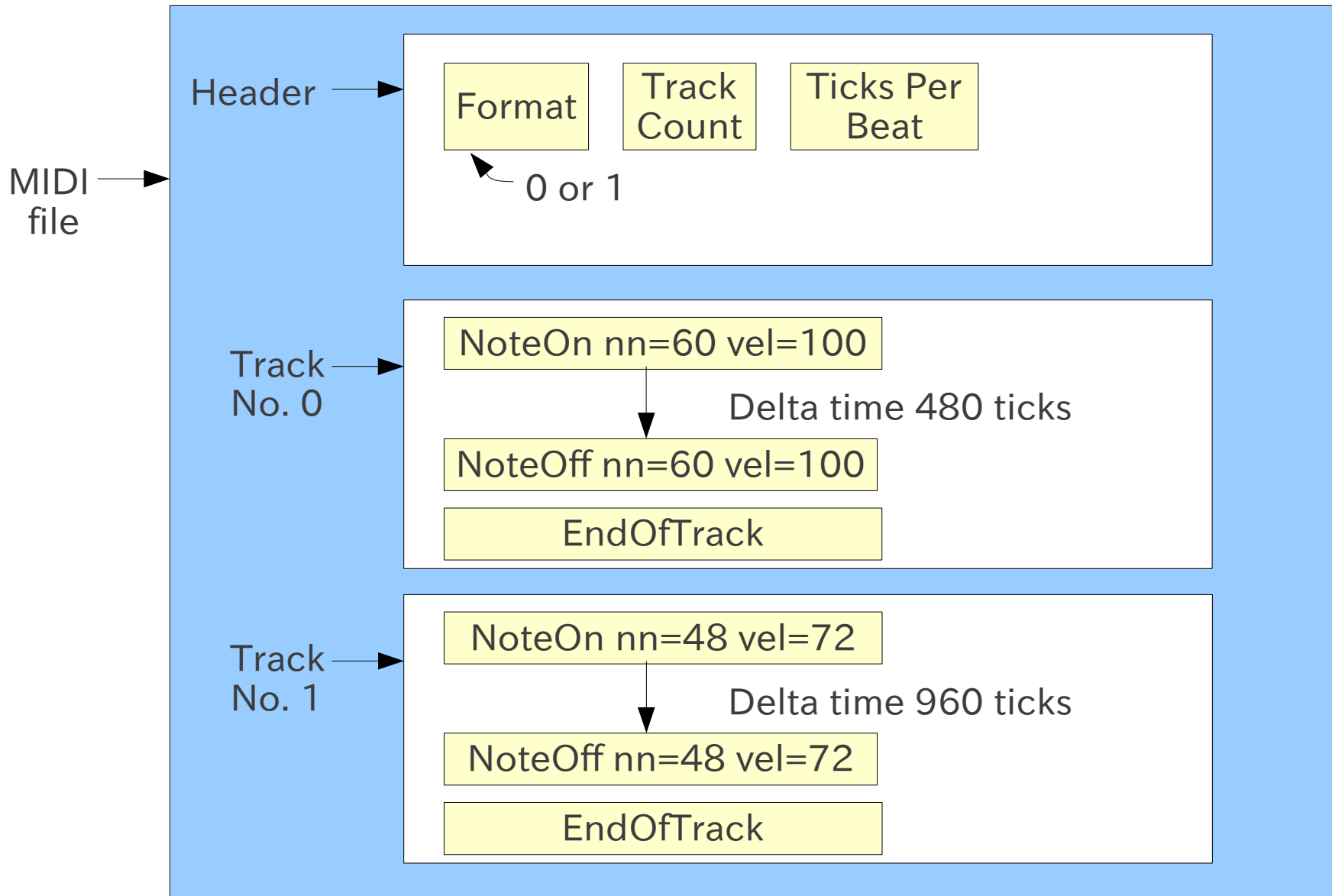
- Help:

```
$ cmx help
```

In the rest of this document, we will show  
only CMXScript code.

# Lesson 1 Read/write MIDI files

# What is a standard MIDI file (SMF)



# View the content of a SMF

- Viewing the content of a SMF is not easy because a SMF is binary
- You can view a SMF by importing it on an existing MIDI sequencer (e.g. Rosegarden), but the content may change when imported, because most MIDI sequencers internally use an original data structure.
- CrestMuse Toolkit supports MIDI XML (one-to-one XMLization of a standard MIDI file)

# Transform SMF to MIDI XML

```
void setup() {  
    def mid = readSMFAsMIDI XML("sample.mid")  
    mid.println()  
}  
  
void draw() {  
    // do nothing  
}
```

- The readSMFAsMIDI XML function (a method of the CMXApplet class) reads a SMF and returns a MIDI XML Wrapper object
- The println method of MIDI XML Wrapper prints the content

# An example of MIDIXML

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE MIDIFile PUBLIC "-//Recordare//DTD MusicXML 1.1 MIDI//EN"
    "http://www.musicxml.org/dtds/midixml.dtd">

<MIDIFile>
  <Format>1</Format>
  <TrackCount>2</TrackCount>
  <TicksPerBeat>480</TicksPerBeat>
  <TimestampType>Delta</TimestampType>
  <Track Number="1">
    <Event>
      <Delta>1920</Delta>
      <EndOfTrack/>
    </Event>
  </Track>
  <Track Number="2">
    <Event>
      <Delta>0</Delta>
      <NoteOn Channel="1" Note="67" Velocity="100"/>
    </Event>
    <Event>
      <Delta>0</Delta>
      <ControlChange Channel="1" Control="7" Value="100"/>
    </Event>
    <Event>
      <Delta>0</Delta>
      <ProgramChange Channel="1" Number="0"/>
    </Event>
  </Track>
</MIDIFile>
```

```
<Event>
  <Delta>360</Delta>
  <NoteOff Channel="1" Note="67" Velocity="100"/>
</Event>
<Event>
  <Delta>0</Delta>
  <NoteOn Channel="1" Note="69" Velocity="100"/>
</Event>
<Event>
  <Delta>120</Delta>
  <NoteOff Channel="1" Note="69" Velocity="100"/>
</Event>
<Event>
  <Delta>0</Delta>
  <NoteOn Channel="1" Note="67" Velocity="100"/>
</Event>
<Event>
  <Delta>240</Delta>
  <NoteOff Channel="1" Note="67" Velocity="100"/>
</Event>
<Event>
  <Delta>0</Delta>
  <NoteOn Channel="1" Note="65" Velocity="100"/>
</Event>
<Event>
  <Delta>240</Delta>
  <NoteOff Channel="1" Note="65" Velocity="100"/>
</Event>
```



# SCCXML: Yet another XML format

- MIDI XML is a simple XMLization of a SMF, so it is sometimes unuseful
- In particular, the data of a single note is divided into two messages (NoteOn and NoteOff), so the NoteOn and NoteOff messages should be matched
- CMX supports a simplified format, SCCXML

# Convert SMF to SCCXML

```
void setup() {  
    def mid = readSMFAsMIDIXML("sample.mid")  
    def scc = mid.toSCCXML()  
    scc.println()  
}  
  
void draw() {  
    // do nothing  
}
```

- Get a MIDIXMLWrapper object with the readSMFAsMIDIXML method
- Convert it to an SCCXMLWrapper object with the toSCCXML method

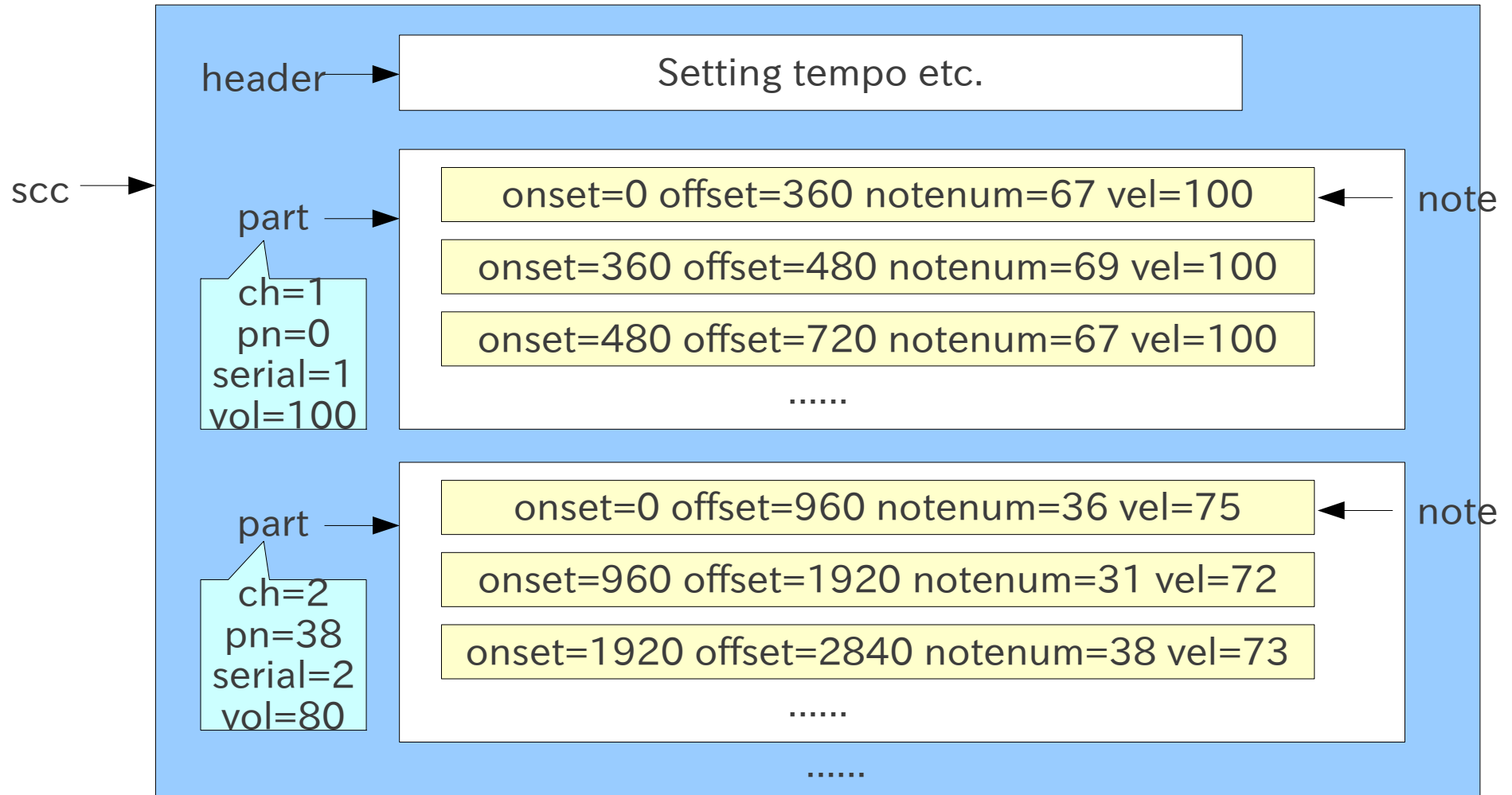
# SCCXMLの例

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE scc PUBLIC "-//CrestMuse//DTD CrestMuseXML SCCXML//EN"
    "http://www.crestmuse.jp/cmx/dtds/sccxml.dtd">
<scc division="480">
  <header/>
  <part ch="1" pn="0" serial="1" vol="100">
    <note>0 360 67 100 100</note>
    <note>360 480 69 100 100</note>
    <note>480 720 67 100 100</note>
    <note>720 960 65 100 100</note>
    <note>960 1200 64 100 100</note>
    <note>1200 1440 65 100 100</note>
    <note>1440 1920 67 100 100</note>
    <note>1920 2160 62 100 100</note>
    <note>2160 2400 64 100 100</note>
    <note>2400 2880 65 100 100</note>
    <note>2880 3120 64 100 100</note>
    <note>3120 3360 65 100 100</note>
    <note>3360 3840 67 100 100</note>
  </part>
</scc>
```

Diagram illustrating the mapping of SCCXML note elements to musical parameters:

Parameter	Value
onset time	0
offset time	360
note number	67
velocity	100
off velocity	100

# Structure of SCCXML



- SCCXML consists of one scc element
- An scc element consists of one header and more-than-zero part elements
- A part element consist of more-than-zero note and other elements

# CMXFileWrapper class

- CMX has one wrapper class for every supported file format
- All wrapper classes are subclasses of CMXFileWrapper
  - MusicXML format → MusicXMLWrapper class
  - SCCXML format → SCCXMLWrapper class
  - MIDIXML format → MIDIXMLWrapper class
- The readfile method of CMXApplet returns a proper wrapper class by checking the content of the file

```
void setup() {  
    def file = readfile("sample.xml")  
    ...  
}
```

# Get the information of each note

Once you get a SCCXMLWrapper object from a SMF, you can use various methods provided by this wrapper class

Example: Read a SMF and print the data of each note

```
void setup() {  
  def scc = readSMFAsMIDIXML("sample.mid").toSCCXML()  
  scc.eachpart { p ->  
    p.eachnote { n ->  
      println(n.onset() + " " + n.offset() + " " +  
              n.notenum() + " " + n.velocity())  
    }  
  }  
}  
  
void draw() {  
  // do nothing  
}
```

# Lesson 2 Realtime processing of MIDI input

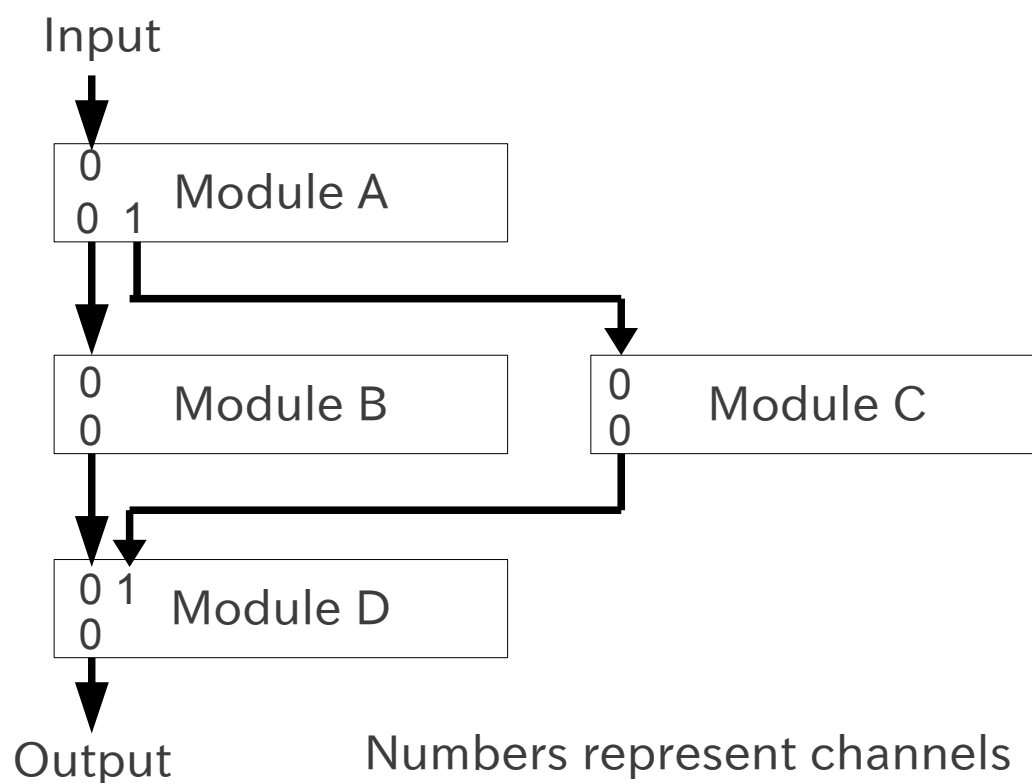
# Basic concept

The whole procedure is divided to some modules.

Data flows from module to module.

||

## Data-flow programming



### Feature of “modules”

- Given data, they are processed and output
- Inputs/outputs may multi-type data
- Processing automatically runs, given data



# Basis

- Generate “modules”
  - Modules are objects of subclasses of the SPModule class
  - Methods for generating some commonly used modules are defined in the CMXApplet class
- Register “modules”
  - Use the addSPModule method of CMXApplet
- Connect the registered “modules”
  - Use the connect method of CMXApplet
- Start the execution of “modules”
  - Automatically started right before draw()

# Step 1 Use built-in modules

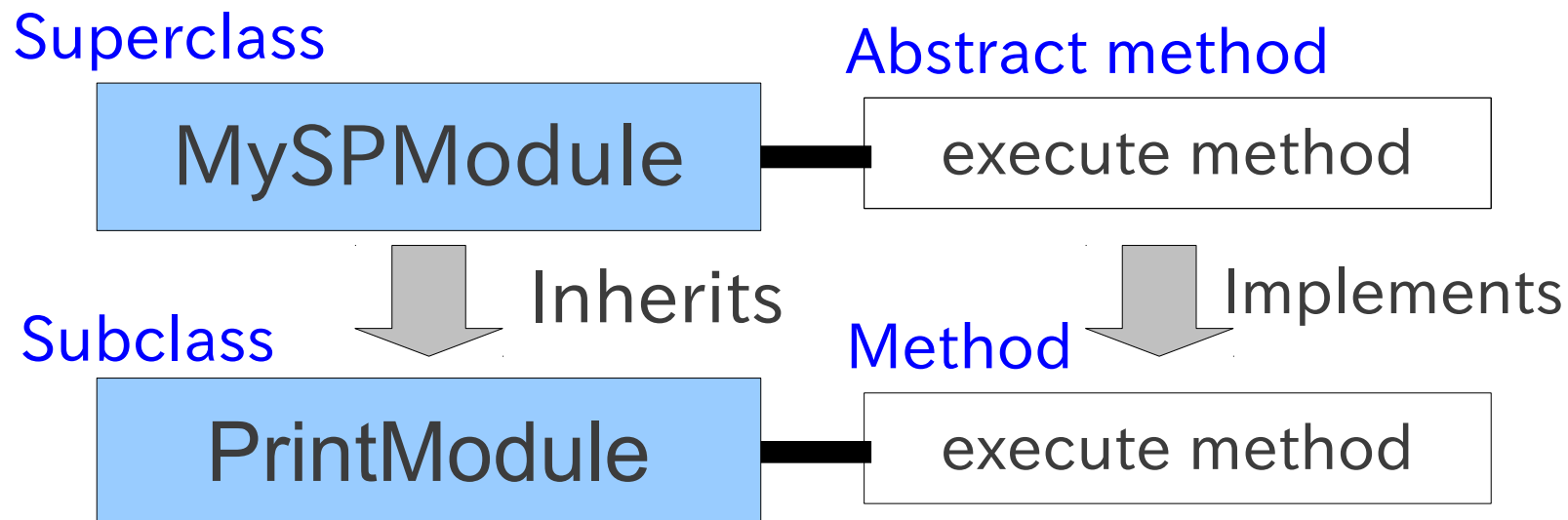
- The following program simply outputs the MIDI data given MIDI performance
  - createVirtualKeyboard ... launches a virtual keyboard
  - createMidiOut ... outputs MIDI data to a MIDI device

```
void setup() {  
    def vk = createVirtualKeyboard()  
    def mo = createMidiOut()  
    addSPModule(vk)  
    addSPModule(mo)  
    connect(vk, 0, mo, 0)  
}  
  
void draw() {  
    // do nothing  
}
```

# Step 2 Define original modules

Let's define "PrintModule" that receives and prints MIDI messages.

Define a subclass of MySPModule



You implements the following three methods:

```
class PrintModule extends MySPModule {  
  def execute(src, dest) {  
    What the module does  
  }  
  
  def inputs() {  
    The class name of objects that the module can receive  
  }  
  
  def outputs() {  
    The class name of objects that the module outputs  
  }  
}
```

# Basic specification of PrintModule

- PrintModule receives, prints, and outputs MIDI messages.
- MIDI messages are treated as MIDIEventWithTicktime objects.

➡ src[0] is a MIDIEventWithTicktime object.

PrintModule outputs it as is after printing it.

```
class PrintModule extends MySPModule {  
  def execute(src, dest) {  
  }  
}
```

An array of received data

Add data by dest[i].add( *data* )

src[0] is a MIDI message

Add data by  
dest[0].add( *MIDI data* )

```
class PrintModule extends SPMModule {
  def execute(src, dest) {
    // Get the status and data bytes from src[0]
    def (status, data1, data2) =
      src[0].getMessageInByteArray()
    // Prints the status and data bytes
    println(status + " " + data1 + " " + data2)
    // Outputs the received data as is
    dest[0].add(src[0])
  }

  def inputs() {
    [MidiEventWithTicktime.class]
  }

  def outputs() {
    [MidiEventWithTicktime.class]
  }
}
```

(Cont'd)

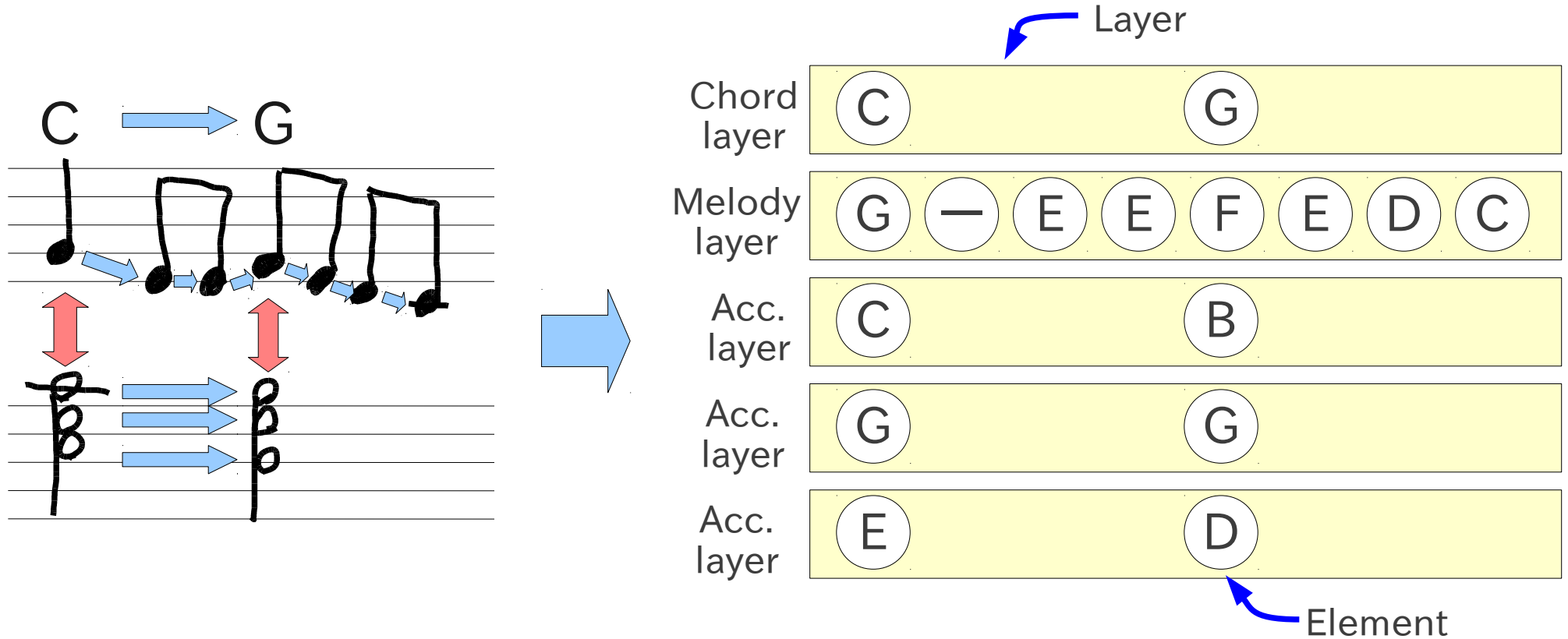
```
void setup() {  
    def vk = createVirtualKeyboard()  
    def pm = new PrintModule()  
    def mo = createMidiOut()  
    addSPModule(vk)  
    addSPModule(pm)  
    addSPModule(mo)  
    connect(vk, 0, pm, 0)  
    connect(pm, 0, mo, 0)  
}  
  
void draw() {  
    // do nothing  
}
```

# Lesson 4 Use MusicRepresentation (a data structure for music inference)



# Basic concept

Music consists of multiple layers, each of which consists of a sequence of elements



Each element is a discrete random variable.

# MusicRepresentation Interface

CMX provides the above-mentioned data structure through **MusicRepresentation** interface. Let's write the code for obtaining a MusicRepresentation object.

In the setup() method...

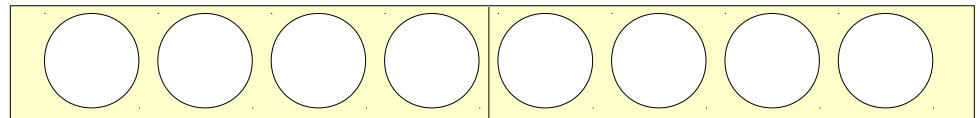
```
def mr = createMusicRepresentation(2, 4)
```

The num  
of measures

The num  
of elements

Then, add a melody layer.

Melody  
layer



```
def notenames = ["C", "C#", "D", "D#", "E", "F",  
                "F#", "G", "G#", "A", "A#", "B"]  
mr.addMusicLayer("melody", notenames)
```

The name of the layer

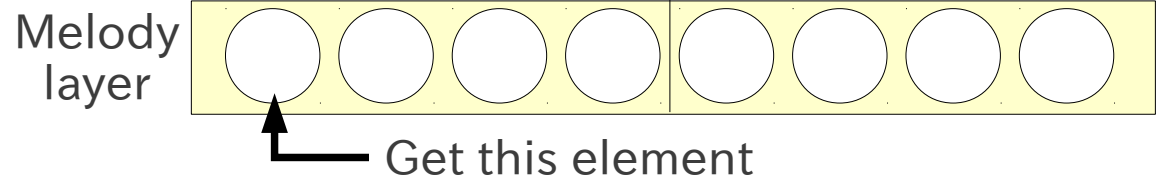
The list of the possible values  
for the elements in the layer

# MusicElement interface

Next, access to some elements in the melody layer.

An element is treated through **MusicElement interface**.

```
def e0 = mr.getMusicElement("melody", 0, 0)
```



Let's set the evidence to “G” for this element using the **setEvidence method**.

```
e0.setEvidence("G")
```

Print the probabilities of each value for this element.

The probabilities will be  $p(\text{"G"})=1$ ,  $p(\text{else})=0$ .

```
notenames.each { x ->
  println("p(" + x + ")=" + e0.getProb(x) )
}
```

Let's set probabilities to of each values for another element.

```
def e1 = mr.getMusicElement("melody", 0, 1)
e1.setProb("A", 0.6)
e1.setProb("F", 0.2)
e1.setProb("G", 0.1)
```

The most value with the highst probability can be obtained with the **getMostLikely** method.

```
println(e1.getMostLikely())
```

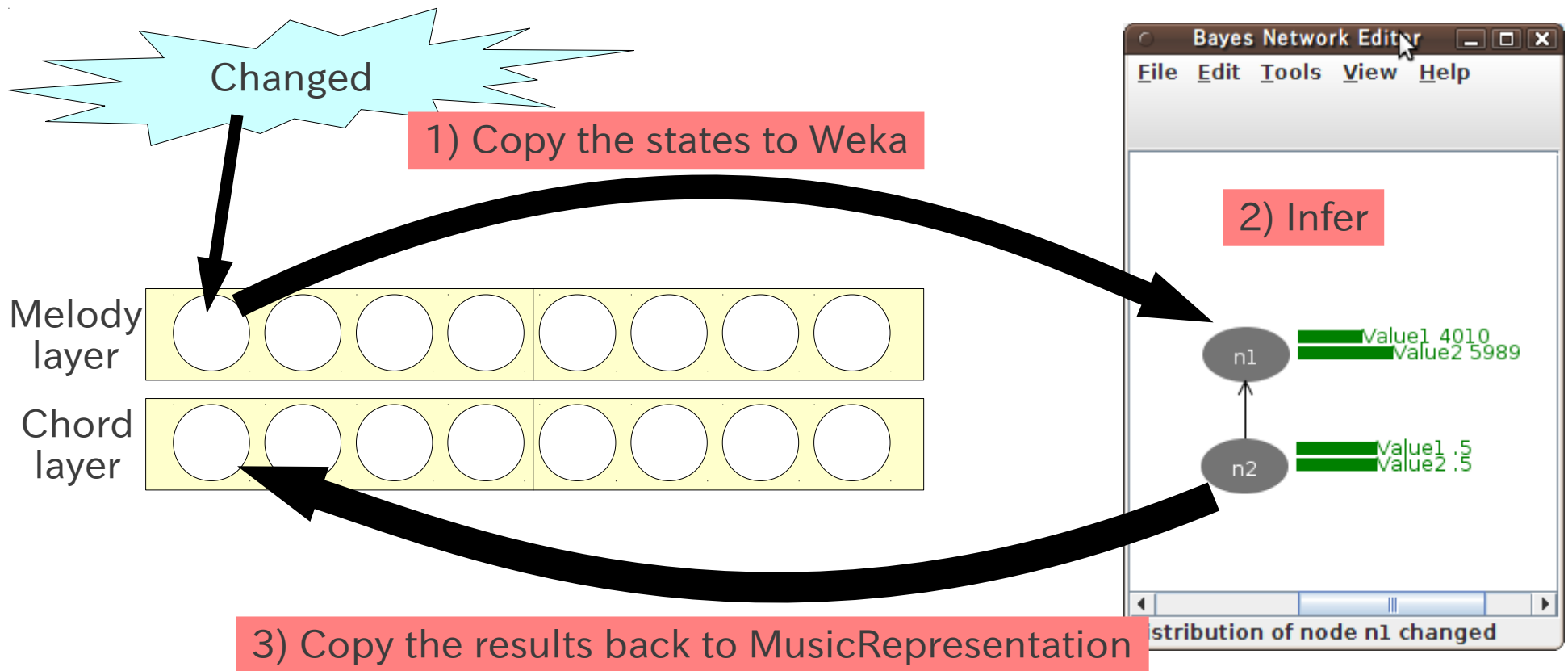
The **generate** method outputs a value at random following the probability distribution.

```
20.times {
  println(e1.generate())
}
```

# Lesson 5 Use a Bayesian network built on Weka

# Basic concept

Copy the states in MusicRepresentation to Weka's Bayesian network, execute the inference in Weka, then copy the results back to MusicRepresentation



# 0. Make an MusicRepresentation object

Make a MusicRepresentation object

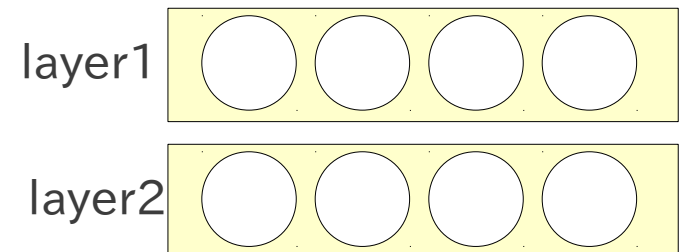
```
def mr = createMusicRepresentation(1, 4)
```

Next, make two layers.

The layers' names are “layer1” and “layer2”.

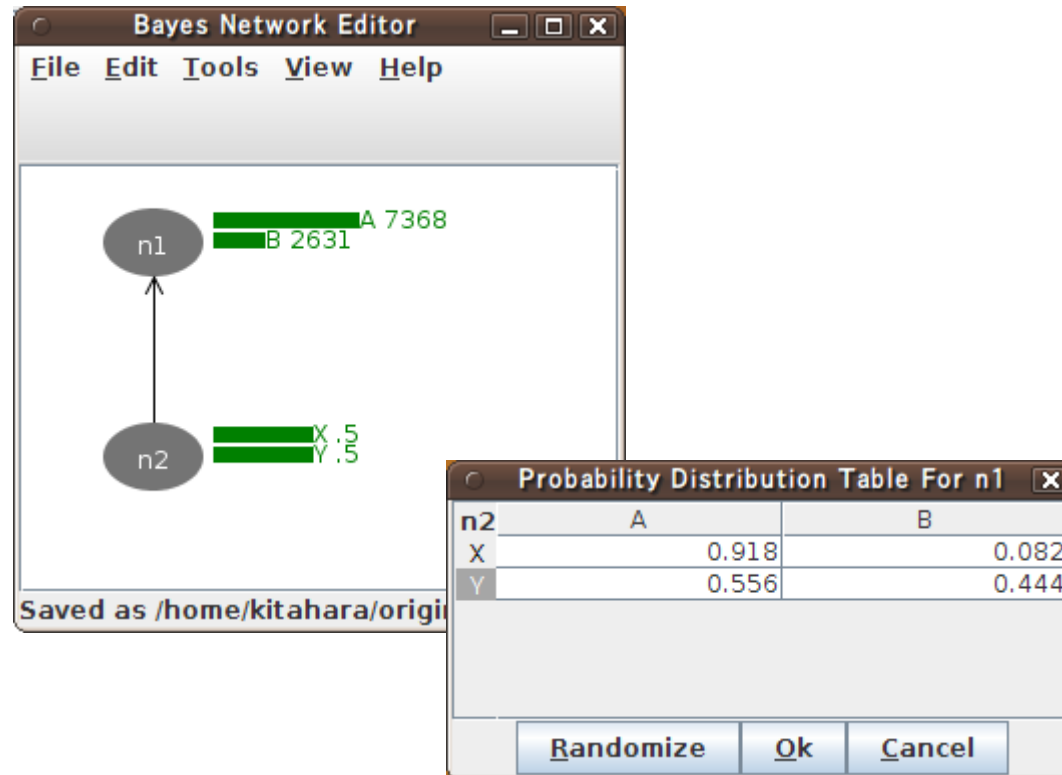
The possible values are “A” and “B” for layer1 and “X” and “Y” for layer2.

```
def values1 = ["A", "B"]  
def values2 = ["X", "Y"]  
mr.addMusicLayer("layer1", values1)  
mr.addMusicLayer("layer2", values2)
```



# 1. Build a Bayesian network on Weka

Build a Bayesian network on Weka and save it in BIFXML





## 2. Construct BayesianCalculator objects

An evidence is set to an element  
in MusicRepresentation

BayesianCalculator  
automatically does

```
graph TD; A([An evidence is set to an element in MusicRepresentation]) --> B[Copy the set evidence to Weka's Bayesian network]; B --> C[Execute the inference on Weka's Bayesian network]; C --> D[Copy the results back to MusicRepresentation];
```

Copy the set evidence to Weka's Bayesian network

Execute the inference on Weka's Bayesian network

Copy the results back to MusicRepresentation

```
/* Read a Bayesian network file built on Weka */  
def bn = new BayesNetWrapper("mybn.xml")
```

```
/* Construct a BayesianCalculator object */  
def bc = new BayesianCalculator(bn)
```

# 3. Construct BayesianMapping objects

Defines the mapping between MusicRepresentation and Weka

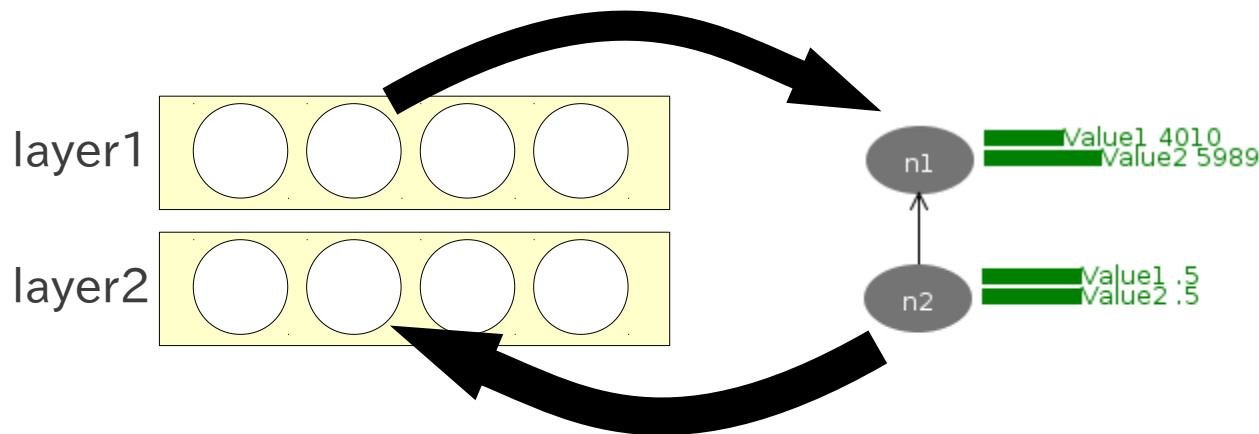
How to construct BayesianMapping objects:

`new BayesianMapping(layer name, 0, 0, node name, bn)`  
MusicRepresentation-side      Weka-side      Bayesian network

**Example** An evidence is set to the i-th element in layer1

➡ `new BayesianMapping("layer1", 0, 0, "n1", bn)`

The i-th element in layer1 corresponds to n1 in Weka



Use addReadMapping  
of BayesianCalculator

Use addWriteMapping  
of BayesianCalculator

The i-th element in layer2 corresponds to n2 in Weka

➡ `new BayesianMapping("layer2", 0, 0, "n2", bn)`

```
bc.addReadMapping(new BayesianMapping("layer1", 0, 0, "n1", bn))  
bc.addWriteMapping(new BayesianMapping("layer2", 0, 0, "n2", bn))
```

## 4. Register BayesianCalculator object

Register the constructed BayesianCalculator object to MusicRepresentation

```
mr.addMusicCalculator("layer1", bc)
```

## 5. Set evidences to MusicRepresentation

- Set an evidence to the first element in layer1
  - The first element in layer2 should be updated

```
def e1 = mr.getMusicElement("layer1", 0, 0)
e1.setEvidence("A")

def e2 = mr.getMusicElement("layer2", 0, 0)
println(e2.getProb("X"))
println(e2.getProb("Y"))
```

- Set an evidence to the second element in layer1
  - The second element in layer2 should be updated

```
def e3 = mr.getMusicElement("layer1", 0, 1)
e3.setEvidence("B")

def e4 = mr.getMusicElement("layer2", 0, 1)
println(e4.getProb("X"))
println(e4.getProb("Y"))
```

# Complete program list

```
void setup() {
    def mr = cmx.createMusicRepresentation(1, 4)
    def values1 = ["A", "B"]
    def values2 = ["X", "Y"]
    mr.addMusicLayer("layer1", values1)
    mr.addMusicLayer("layer2", values2)

    def bn = new BayesNetWrapper("mybn.xml")
    def bc = new BayesianCalculator(bn)
    bc.addReadMapping(new BayesianMapping("layer1", 0, 0, "n1", bn))
    bc.addWriteMapping(new BayesianMapping("layer2", 0, 0, "n2", bn))
    mr.addMusicCalculator("layer1", bc)

    def e1 = mr.getMusicElement("layer1", 0, 0)
    e1.setEvidence("A")

    def e2 = mr.getMusicElement("layer2", 0, 0)
    println(e2.getProb("X"))
    println(e2.getProb("Y"))

    def e3 = mr.getMusicElement("layer1", 0, 1)
    e3.setEvidence("B")

    def e4 = mr.getMusicElement("layer2", 0, 1)
    println(e4.getProb("X"))
    println(e4.getProb("Y"))
}

void draw() {
}
```

# Application

- Set evidences to the melody layer and then infer the values for the chord layer  
→ Automatic harmonization

Finally

# Other features

- Supporting various XML formats
  - MusicXML, DeviationInstanceXML, MusicApexXML, etc.
- Using external MIDI devices
  - A chooser of external MIDI devices is supported.
- Advanced MIDI processing
  - For example, integration of playback of MIDI files and realtime MIDI processing.
- Audio signal processing
  - For example, a realtime Fourier transform for WAV files or microphone inputs



# Contact us

- Web: <http://cmx.sourceforge.jp/>  
<http://sourceforge.jp/projects/cmx/>
- E-mail: `kitahara [at] kthrlab.jp`