

CrestMuse Toolkit (CMX) ver.0.61

説明書

北原 鉄朗

(日本大学 文理学部 情報システム解析学科)

kitahara [at] kthrlab.jp

Lesson 0 概要

CMXで何ができるのか

- 音楽データに対する様々な処理を容易にするJavaライブラリ.
- たとえば,こんなことができる.
 - MusicXML, 標準MIDIファイルなどの入出力
 - MIDIやオーディオデバイスからの入力のリアルタイム処理
 - ベイジアンネットワークを用いた音楽データの自動生成
(weka.jarが別途必要)
- プログラミングのためのライブラリであって, 単体で便利に使えるアプリではない.
- Javaの他, JVM上の各種言語 (Processing, Groovy, etc.) から利用できる.

インストール方法

- zipファイルを展開し, cmx.jarと, libに入っているいくつかのjarファイルを, 所定のディレクトリ(フォルダ)にコピーすればOK
 - Processingから使う場合:
~/sketchbook/libraries/cmx/library
 - Groovyから使う場合: ~/.groovy/lib/
 - Javaから使う場合: (JDKのインストール先)/jre/lib/ext/
- 任意の場所に置いてCLASSPATHを通すのもよい.
- install.sh を実行すると, これを自動でやってくれる.
(ルート権限で実行すること. つまり, \$ sudo ./install.sh)

上記は, UNIX系OSの場合. OSによってファイルの置き場などが異なるので, 自身で確認すること.

使い方の基本

- その1 CMXScript
- その2 CMXAppletのサブクラスを作る
- その3 CMXControllerクラスを使う
- その4 コマンドラインからファイル変換器として使う

使い方その1 CMXScript

- 本バージョンから導入
- setup()、draw()などの関数を定義すれば、自動的にCMXAppletのサブクラスに変換される
- Processingとある程度互換性を保っている

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

左のコードを入力し、

```
$ cmxscript ファイル名
```

で実行する。

使い方その2 CMXAppletのサブクラスを作る

- CMXAppletのサブクラスを自分で作る方法。GroovyでもJavaでも可能。下はGroovyでの例。



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

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

MyApplet.start("MyApplet")
```

詳細は, CMXAppletのJavaDocを参照すること.

使い方その3 CMXControllerを使う

- Processingから使いたい場合は、その1、その2の方法は使えないので、CMXControllerクラス経由で機能呼び出す

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

CMXController cmx = CMXController.getInstance();

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

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



詳細は、CMXControllerのJavaDocを確認すること。

使い方その4 ファイル変換器として使う

- MusicXML, SCCXML, MIDIXML, 標準MIDIファイルなどの相互変換をコマンドラインから行うことができる。
- 基本的な使い方

```
$ cmx (Javaコマンドのオプション) コマンド名 (オプション)
```

- 例

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

「myfile.mid」という名の標準MIDIファイルをSCCXML形式に変換

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

上と同様の変換を行って「myscconfig.xml」に保存

- ヘルプ

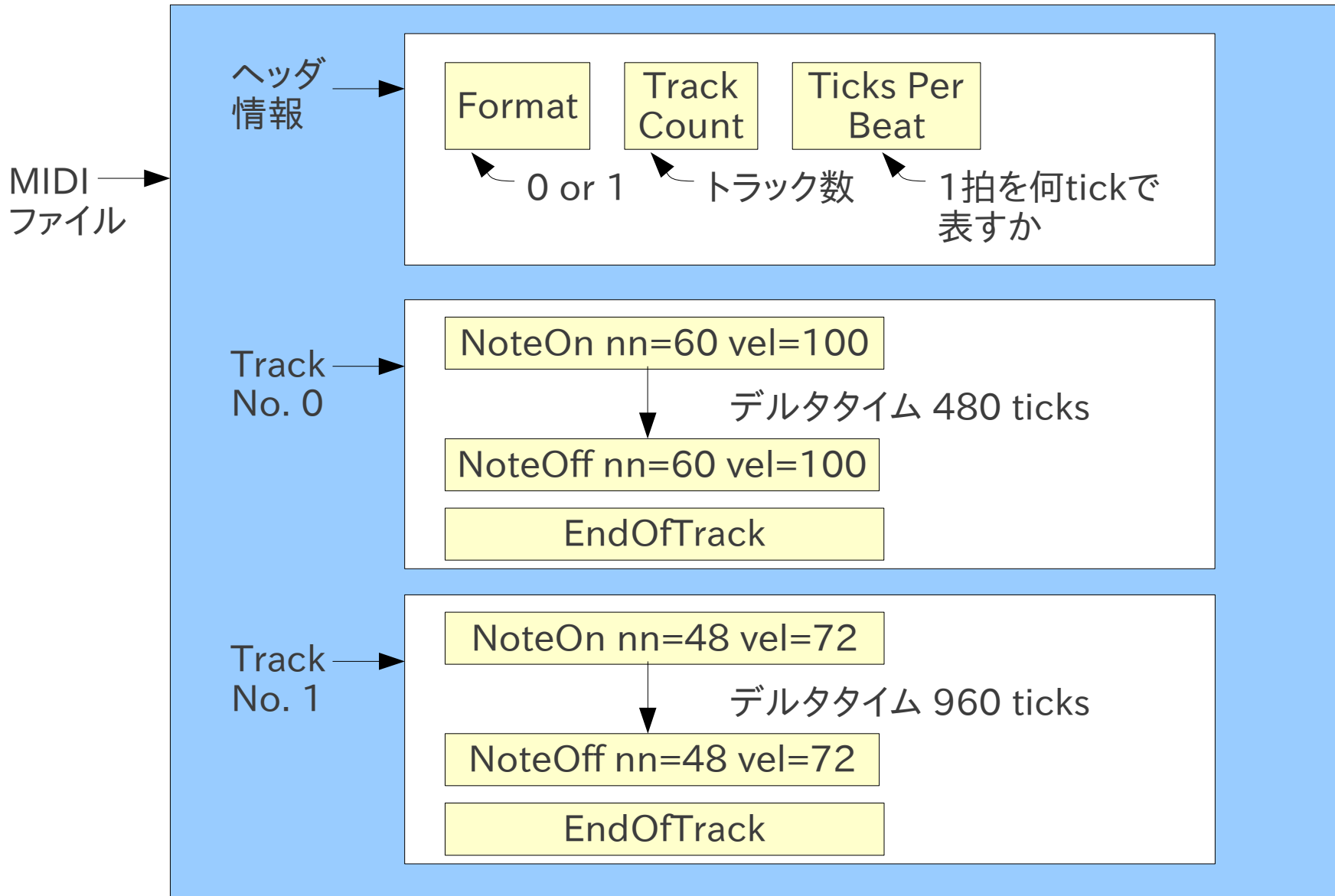
```
$ cmx help
```

以降、CMXScriptを用いた使い方のみ示す。

Lesson 1 MIDIファイルを読み書きする

標準MIDIファイル(SMF)とは

基本的にはMIDIケーブルに流れるMIDI信号をファイルに書き出したもの



SMFの中身を見たい

- SMFはバイナリーファイルなので、中身を簡単に見ることができない
- 既存のMIDIシーケンサ (e.g. Rosegarden) にインポートすれば中身を見ることはできるが、MIDIシーケンサでは音楽データを独自のフォーマットで表す場合が多く、インポート時に独自フォーマットに変換されてしまっていることが考えられる。
- CrestMuse Toolkitでは、SMFをそのままXML形式に書き直した「MIDI XML」という形式に対応している。この形式に変換してみよう。

SMFをMIDIXMLに変換して画面表示

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

- readSMFAsMIDIXMLという組み込み関数(実際にはCMXAppletクラスのメソッド)が、SMFを読み込んでMIDIXMLWrapperオブジェクトを返す。
- MIDIXMLWrapperクラスのprintlnメソッドで画面表示

MIDI XMLの例

```
<?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>
```

SMFをそのままXML化したものなので、
SMFの中身のチェックに便利

```

<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>

```


もう1つのXML形式「SCCXML」

- MIDI XMLは、SMFを完全にそのままXML化しているので、どんなSMFでも変換できるが、煩雑。
- 特に、発音 (Note On) と消音 (Note Off) が分かれており、音符の長さを調べようと思ったら、Note On と Note Off の対応を取らないといけない。
- より単純化された形式として、SCCXML というものを用意している。

SMFをSCCXMLに変換して画面表示

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

- readSMFAsMIDIXMLメソッドで一旦MIDIXMLWrapperオブジェクトを取得
- toSCCXMLメソッドでSCCXMLWrapperオブジェクトに変換

SCCXMLの例

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE scc PUBLIC "-//CrestMuse//DTD CrestMuseXML SCCXML//EN"
    "http://www.crestmuse.jp/cmxml/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>
```

onset
time

offset
time

note
number

velocity

off
velocity

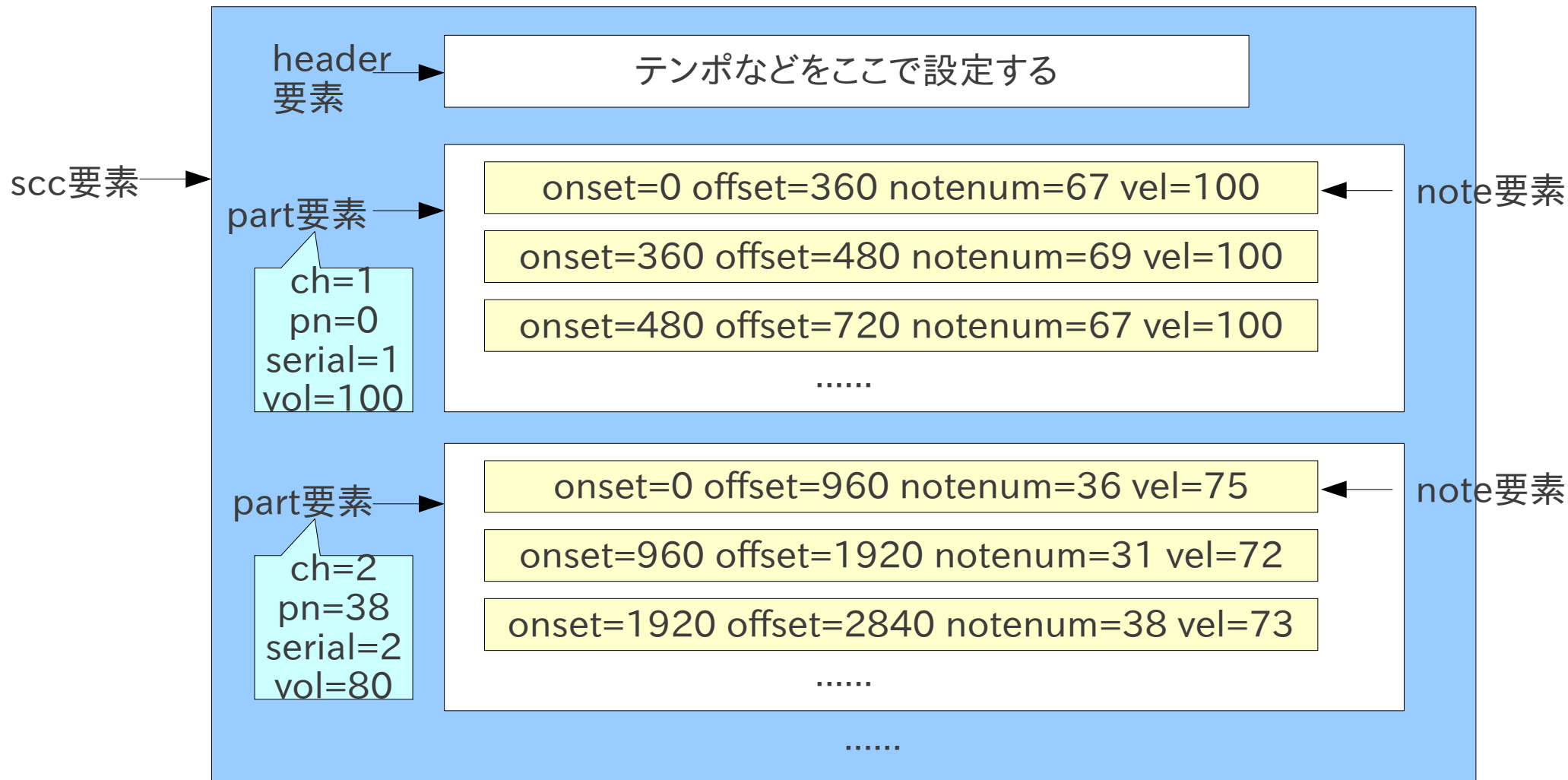
MIDIシーケンサの

「イベントリスト画面」に似せた表記

チャンネル番号(cn)、音色番号(pn)、
ボリューム(vol)をpartの属性として記述

頑張れば手入力もできる程度の複雑さ

SCCXMLの構造



- SCCXMLは、1つのscc要素で構成される
- scc要素は、1つのheader要素と0個以上のpart要素で構成される
- part要素は、0個以上のnote要素で構成される（他にもあるが）

CMXFileWrapperクラス

- CrestMuse Toolkitでは, 読み書きできるファイル形式の1個1個にラッパクラスが用意されている.
- ラッパクラスは, すべてCMXFileWrapperのサブクラス
 - MusicXML形式 → MusicXMLWrapperクラス
 - SCCXML形式 → SCCXMLWrapperクラス
 - MIDIXML形式 → MIDIXMLWrapperクラス
- CMXAppletクラスのreadfileメソッドを使うと, ファイル形式に合ったラッパクラスのオブジェクトが得られる

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

Javaなどの場合は
必要に応じてダウンキャスト

SMFを読み込んで、 各音符の情報を取得する

さきほどの方法でSMFからSCCXMLWrapperオブジェクトを得たら、SCCXMLWrapperクラスが提供する各メソッドを用いればよい

例：SMFを読み込んで、各パート/各音符の情報を標準出力

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

Lesson 2 MusicXMLを読み込み, 楽譜に 付与されている記号に従って演奏を生成する

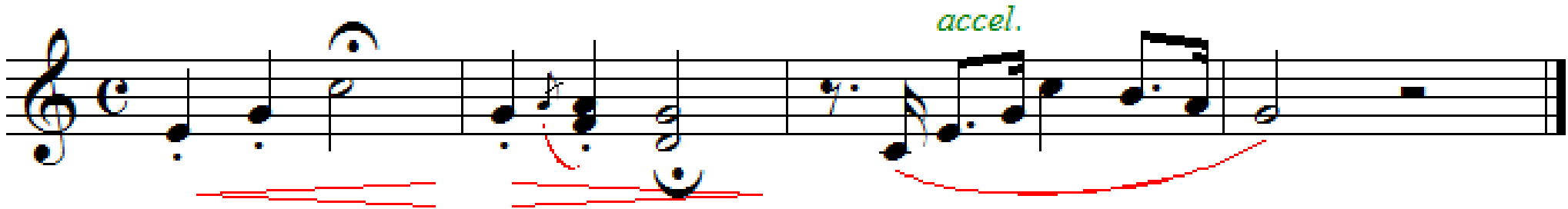
MusicXMLとは

楽譜を記述するためのXMLフォーマット

なんでMIDIファイルじゃダメなの？

- MIDIファイルは、楽譜ではなく「演奏」を記録するフォーマット
 - 小節線や休符という概念がない。
 - スタッカートやフェルマータなどを記述できない。

MusicXMLの例



```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE score-partwise PUBLIC
"-//Recordare//DTD MusicXML 1.0 Partwise//EN"
"http://www.musicxml.org/dtds/partwise.dtd">
<score-partwise>
  (中略)
  <part-list>
    <score-part id="P1">
      (中略)
    </score-part>
  </part-list>
  <!--=====-->
  <part id="P1">
    <measure number="1">
      <attributes>
        <divisions>8</divisions>
        (中略)
      </attributes>
      <direction placement="below">
        <direction-type>
          <wedge default-y="-72" spread="0"
            type="crescendo"/>
        </direction-type>
        <offset>3</offset>
      </direction>
      <note>
        <pitch>
```

```
<step>E</step>
<octave>4</octave>
</pitch>
<duration>8</duration>
<voice>1</voice>
<type>quarter</type>
<stem>up</stem>
<notations>
  <articulations>
    <staccato placement="below"/>
  </articulations>
</notations>
</note>
<note>
  <pitch>
    <step>G</step>
    <octave>4</octave>
  </pitch>
  <duration>8</duration>
  <voice>1</voice>
  <type>quarter</type>
  <stem>up</stem>
  <notations>
    <articulations>
      <staccato placement="below"/>
    </articulations>
  </notations>
</note>
```

```

</note>
  <note>
    <pitch>
      <step>C</step>
      <octave>5</octave>
    </pitch>
    <duration>16</duration>
    <voice>1</voice>
    <type>half</type>
    <stem>down</stem>
    <notations>
      <fermata type="upright"/>
    </notations>
  </note>
  <direction>
    <direction-type>
      <wedge default-y="-71"
        spread="12" type="stop"/>
    </direction-type>
    <offset>-3</offset>
  </direction>
</measure>
<!--=====-->
<measure number="2">
  <direction placement="below">
    <direction-type>

```

以下省略

MusicXML(partwise)の基本構造

score-partwise (トップレベルタグ)

part-list
(パート
情報を
記述)

part

measure

note

note

...

measure

note

note

...

...

part

measure

note

note

...

measure

note

note

...

...

⋮

演奏生成の基本方針

- ここでのタスクは、「楽譜を読んでそれを演奏すること」
- ここでは簡単に,
 - ・ スタッカートが付いている音符は半分の長さで,
 - ・ フェルマータが付いている音符は半分のテンポで演奏することとする.
- 楽譜はMusicXML, 演奏はMIDI(SCCXML)で表される
- 演奏は, 完全に楽譜通りの演奏から適度に逸脱させることで音楽らしさを作っている. この逸脱の情報の記述のためにDeviationInstanceXML が用意されている.
- まず, DeviationDataSetクラスのインスタンスを生成し, DeviationInstanceWrapperオブジェクトに変換し, その後, SCCXMLWrapperオブジェクトに変換する.

MusicXMLから各音符の情報を取得する

Lesson 1と同様に, CMXAppletのreadfileメソッドで

ファイルを読み込む (MusicXMLWrapperオブジェクトが得られる)

```
void setup() {
  def musicxml = readfile("sample.xml")
  musicxml.eachnote { note ->
    if (note.hasArticulation("stacatto") {
      /* スタッカートがあったときの処理 */
    } else {
      /* スタッカートがなかったときの処理 */
    }
    def notations = note.getFirstNotations()
    if (notations != null && notations.fermata() != null) {
      /* フェルマータがあったときの処理 */
    } else {
      /* フェルマータがなかったときの処理 */
    }
  }
}

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

演奏データを生成する

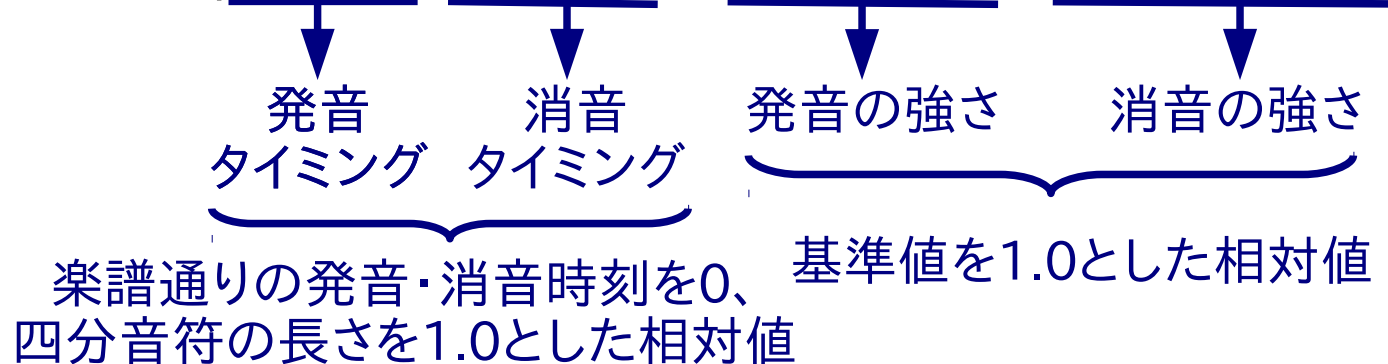
ここでは、実際の演奏データの代わりに、
演奏の楽譜からの差分 (deviationデータ) を生成する。

どうやって?

deviationの生成には、DeviationDataSetクラスを使う。

- ある音符の長さを短くするには、addNoteDeviationを使う。

`addNoteDeviation(attack, release, dynamics, endDynamics);`



- テンポを変えるには、addNonPartwiseControlを使う。

`addNonPartwiseControl(measure, beat, "tempo-deviation", value);`



```
void setup() {
  def musicxml = cmx.readfile("sample.xml")
  def deviation = new DeviationDataSet(musicxml)
  musicxml.eachnote { note ->
    if (note.hasArticulation("stacatto"))
      deviation.addNoteDeviation(note, 0.0,
                                -note.actualDuration()/2, 1.0, 1.0)
    def notations = note.getFirstNotations()
    if (notations != null && notations.fermata() != null)
      deviation.addNonPartwiseControl(note.measure().number(),
                                      note.beat(), "tempo-deviation", 0.5)
  }
  def devins = deviation.toWrapper()
  devins.finalizeDocument()
  devins.writefile("deviation.xml")
  devins.toSCXML(480).toMIDIXML().writefileAsSMF("result.mid")
}

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

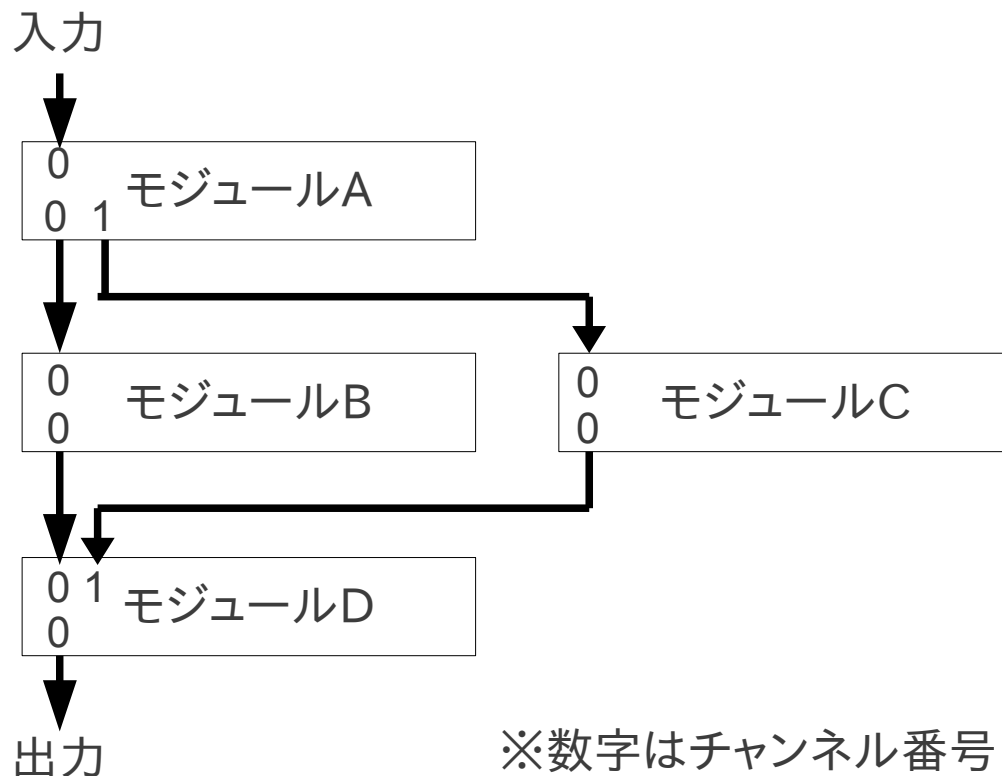
Lesson 3 MIDI入力をリアルタイムで処理する

基本的な考え方

処理全体がいくつかの「モジュール」に分割され、
モジュールの中をデータが流れていくものとする

II

データフロー型プログラミング



「モジュール」の特徴

- 何らかのデータが入力され、処理された後、出力される。
- 複数種のデータが入力or出力されてもいい。
- データが到着するたびに、処理が自動的に行われる。

基本的な処理の流れ

- 「モジュール」のオブジェクトを生成する
 - モジュールは, SPModuleクラスのサブクラスである
 - 主要なモジュールは, CMXAppletクラスに生成用メソッド用意
- 「モジュール」を「登録」する
 - CMXApplet の addSPModuleメソッドを用いる
- 「モジュール」の接続方法を定義する
 - CMXApplet の connectメソッドを用いる
- 「モジュール」を実行を開始する
 - draw関数実行直前に、自動的に開始する

Step 1 既存のモジュールを組み合わせて使う

- 下は, 仮想鍵盤での演奏をMIDIデバイスに出力するプログラム
 - createVirtualKeyboard ... 仮想鍵盤のモジュール
 - createMidiOut ... MIDIデバイスに出力するモジュール

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

Step 2 独自モジュールを作成する

MIDIメッセージを受け取って画面表示する
モジュール「PrintModule」を作成してみよう

基本的な考え方

MySPModuleクラスを継承し、必要なメソッドを実装

スーパークラス

MySPModule

抽象メソッド

executeメソッド

宣言のみで
処理内容は
未定義

継承

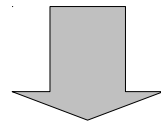
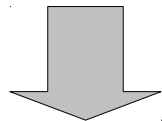
サブクラス

PrintModule

実装

executeメソッド

具体的な
処理内容を
定義



SPModuleのサブクラスを作成するには, 次の3つのメソッドを実装する

```
class PrintModule extends MySPModule {
```

```
  def execute(src, dest) {
```

ここにモジュールがすべき処理内容を記述する

```
}
```

```
def inputs() {
```

ここに、このモジュールが受け付けるオブジェクトのクラス名を書く

```
}
```

```
def outputs() {
```

ここに、このモジュールが出力するオブジェクトのクラス名を書く

```
}
```

```
}
```

PrintModuleの基本方針

- PrintModuleは, 仮想鍵盤の演奏データを受け取り, 画面表示し, そのまま出力する
 - 演奏データは, MIDIEventWithTicktimeクラスとして扱う
- ➡ src[0]がMIDIEventWithTicktimeオブジェクトなので, 必要なメソッドを使用したのち, dest[0]にそのままadd

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

各入力チャンネルから
得られたデータがここにある

今回は、src[0]が
入力されたMIDIデータ

dest[チャンネル番号].add(出力データ)
とすればデータが出力される

今回は、dest[0].add(MIDIデータ)
とすればよい

```
class PrintModule extends SPMModule {
  def execute(src, est) {
    // src[0]からステータスバイトとデータバイトを取得
    def (status, data1, data2) =
      src[0].getMessageInByteArray()
    // 取得したステータスバイトとデータバイトを画面表示
    println(status + " " + data1 + " " + data2)
    // 入力されたデータをそのまま出力
    dest[0].add(src[0])
  }

  def inputs() {
    [MidiEventWithTicktime.class]
  }

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

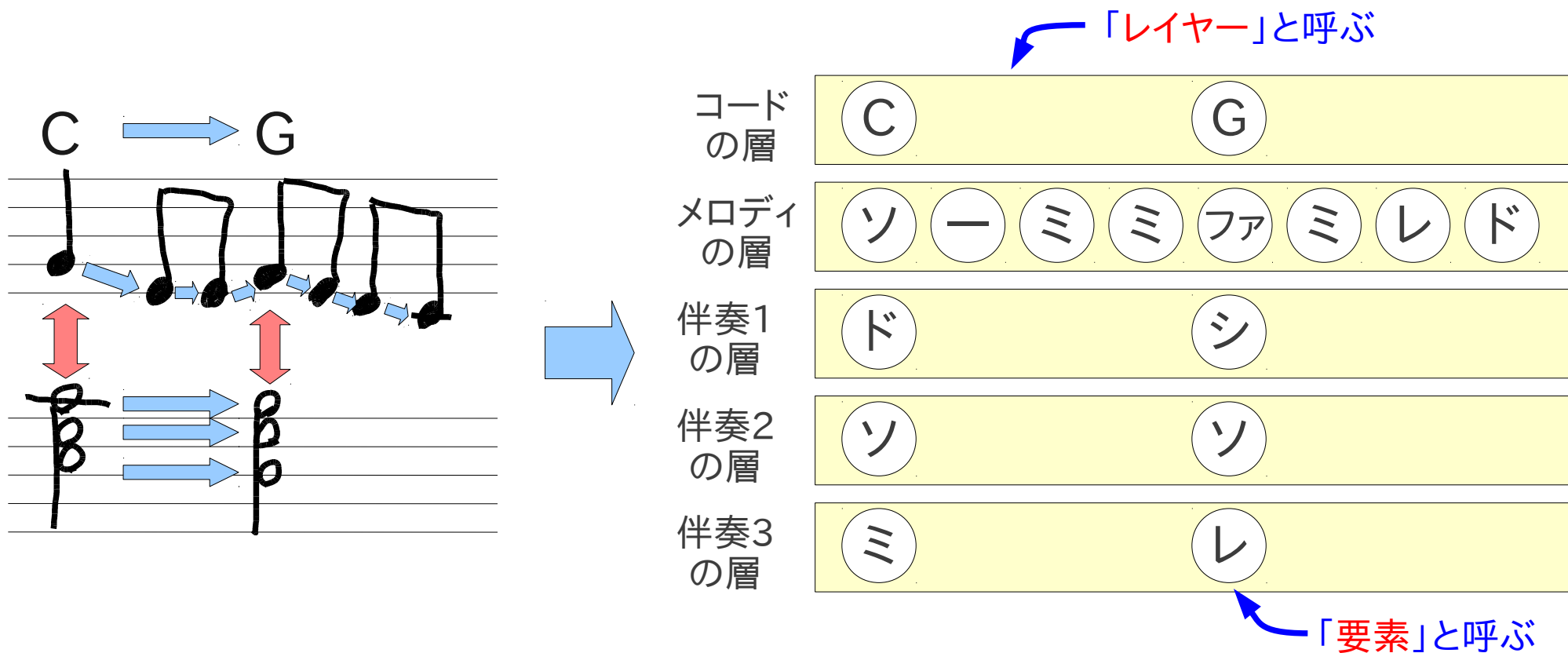
(続き)

```
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 音楽データの確率推論のための データ構造「MusicRepresentation」を使う

基本的な考え方

音楽は、時間軸に沿って一列に並んだデータ列が、何層に渡って出来ていると考えるとわかりやすい。



個々の要素は、離散確率変数になっている

MusicRepresentationインタフェース

前述のデータ構造を実現するものとして, **MusicRepresentation** というインタフェースが用意されているので, これを利用する.

では, さっそくMusicRepresentationのインスタンスを用意しよう.

setupメソッドの中で...

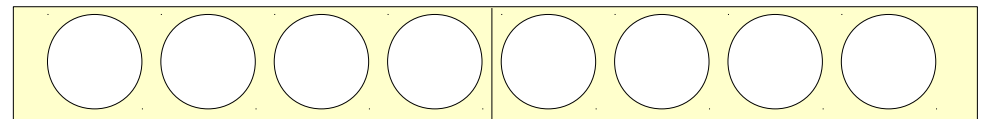
```
def mr = createMusicRepresentation(2, 4)
```

小節数

1小節に何個
要素を作るか

この段階では, レイヤーが
1つもない空の状態なので,
メロディレイヤーを追加しよう.

メロディ
の層



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

レイヤーの名前を
文字列で指定

そのレイヤーの各要素がとりうる
値の一覧を文字列の配列で指定

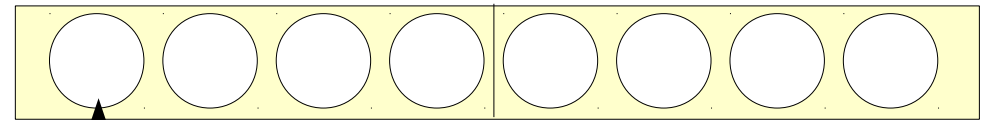
MusicElementインタフェース

次に、今追加したメロディレイヤーの各要素をいろいろいじってみよう。
各要素は**MusicElementインタフェースのオブジェクト**として扱われる。

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

↑ melodyレイヤー, 0小節め,
0番めの要素を取得

メロディ
の層



↑ この要素を取得

この要素に「ソ」をエビデンスとしてセットしてみよう。

エビデンスをセットするには、**setEvidenceメソッド**を使う。

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

この要素が各値を取る確率を出力してみよう。"G"にエビデンスを
セットしたのだから、 $p("G")=1$, $p(\text{それ以外})=0$ になるはずだ。

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

今度は, その次の要素の各値に確率をセットしてみよう.

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

確率が最も高いラベルは, **getMostLikely**メソッドで取得できる.

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

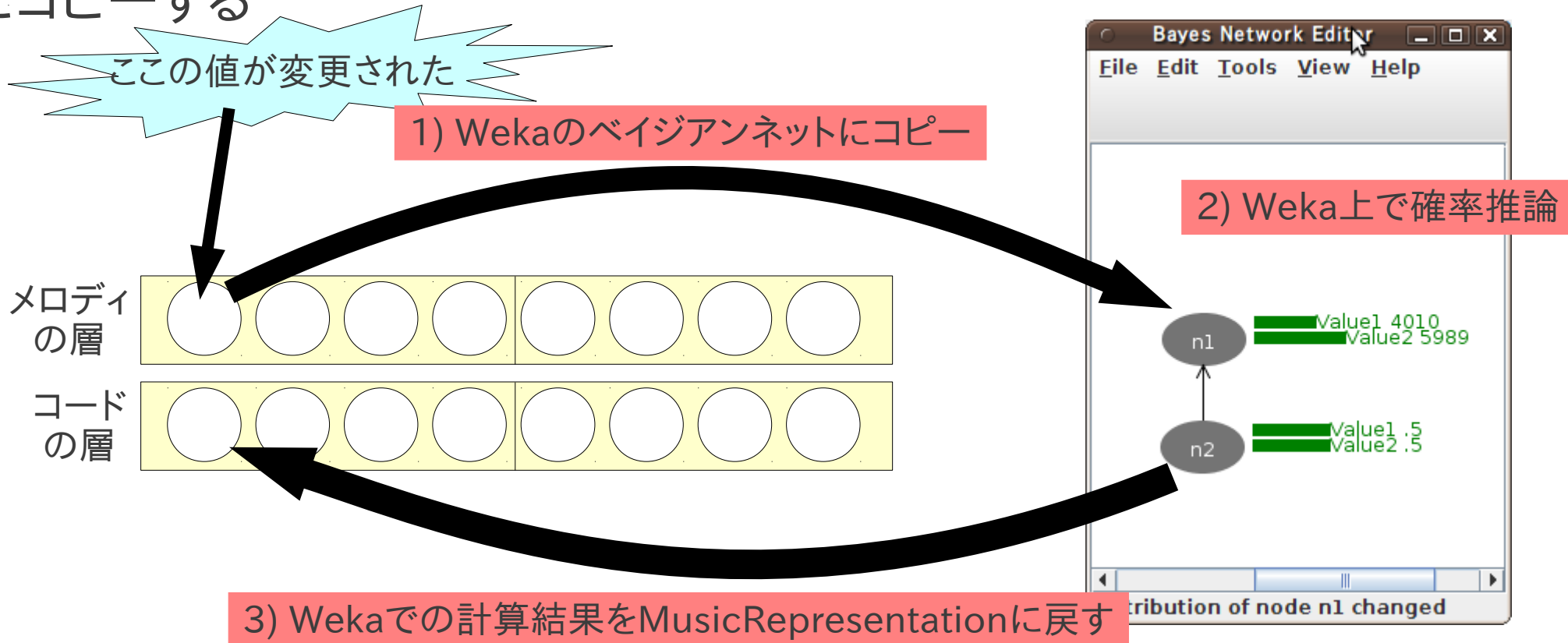
generateメソッドを使うと, セットされている確率分布に従って,
ランダムにラベルを出力する.

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

Lesson 5 データマイニングツール「Weka」で 構築したベイジアンネットワークを使う

基本的な考え方

MusicRepresentation上の状態をWekaのベイジアンネットにコピーし、Weka上で確率推論を行って、その結果を再びMusicRepresentationにコピーする



上の例では、メロディに関する情報をWeka上では「n1」が、コードに関する情報を「n2」が表していることを前提としている。

基本的な手順

1. Weka上でベイジアンネットを作成して, ファイルに保存
 - 作成する際には, 個々のノードが MusicRepresentation のどのレイヤーに対応するのかを考えること
 - 各ノードのとりうる値の個数と, MusicRepresentation 上で対応するレイヤーでのとりうる値の個数が一致すること
2. Wekaへのコピーなどを自動的に行ってくれる「BayesianCalculator」オブジェクトを作成
3. WekaとMusicRepresentationの対応を示す「BayesianMapping」オブジェクトをBayesianCalculatorに登録
4. BayesianCalculatorをMusicRepresentationに登録
5. あとは, MusicRepresentationからMusicElementを取得してsetEvidenceなどすれば, 自動的に推論してくれる

0. MusicRepresentationオブジェクトを作成

Lesson 4を参考に, MusicRepresentationオブジェクトを作成しよう.

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

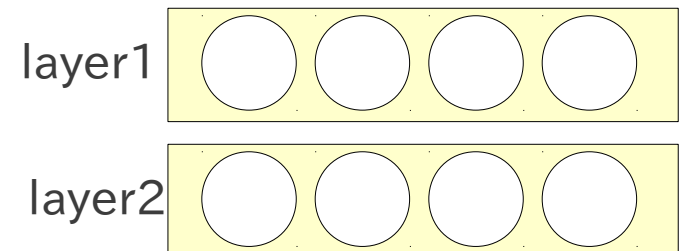
次に, レイヤーを2つ作ってみよう.

ここでは, 名前を「layer1」, 「layer2」とする(名前は何でもよい).

また, layer1は"A", "B"の2つの値を, layer2は"X", "Y"の2つの値をとりうるものとする.

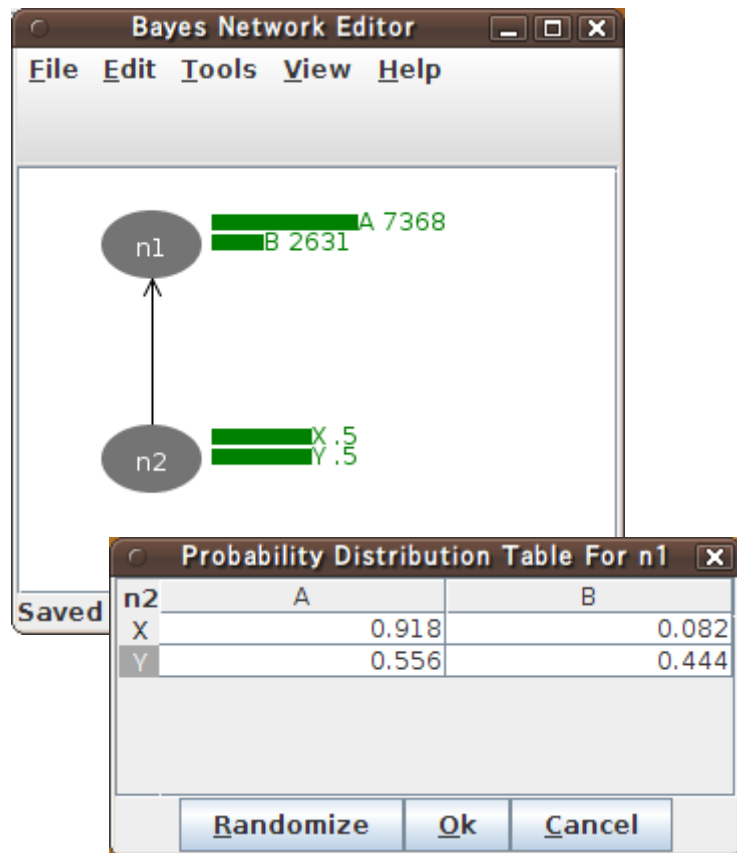
```
def values1 = ["A", "B"]  
def values2 = ["X", "Y"]  
mr.addMusicLayer("layer1", values1)  
mr.addMusicLayer("layer2", values2)
```

ここまではLesson 4の内容



1. Wekaでベイジアンネットを作る

Wekaで適当なベイジアンネットを作って, BIFXML形式で保存しよう.



注意点

- 先ほど作ったMusicRepresentationどのレイヤーが, どのノードに対応するかを考えながら作ること.
 - ここでは, 「layer1」が「n1」に, 「layer2」が「n2」に対応している.
- 各ノードがとりうる値は, MusicRepresentation側の対応するレイヤーのとりうる値に合わせること
 - 「layer1」と「n1」のとりうる値が"A", "B"
 - 「layer2」と「n2」のとりうる値が"X", "Y"

ここでは, 「layer1」の値を「n1」にコピーして Wekaの確率推論機能で求めた「n2」の値を再び「layer2」にコピーする, というのを想定

2. BayesianCalculatorオブジェクトを作成

MusicRepresentationのある要素に
エビデンスがセットされた

BayesianCalculatorが
自動でしてくれる

セットされたエビデンスを Weka のベイジアンネットにコピー

Weka のベイジアンネット上で確率推論を実行

確率推論の結果をMusicRepresentationに戻す

```
/* Wekaで保存したベイジアンネットのファイルを読み込む部分 */  
def bn = new BayesNetWrapper("mybn.xml")
```

```
/* BayesianCalculatorオブジェクトの作成(読み込んだベイジアンネットを  
指定する) */  
def bc = new BayesianCalculator(bn)
```

3. BayesianMappingオブジェクトを作成

MusicRepresentation と Weka のベイジアンネットの対応関係を定義

BayesianMappingオブジェクトの作り方:

```
new BayesianMapping(レイヤー名, 0, 0, ノード名, bn)
```

MusicRepresentation側

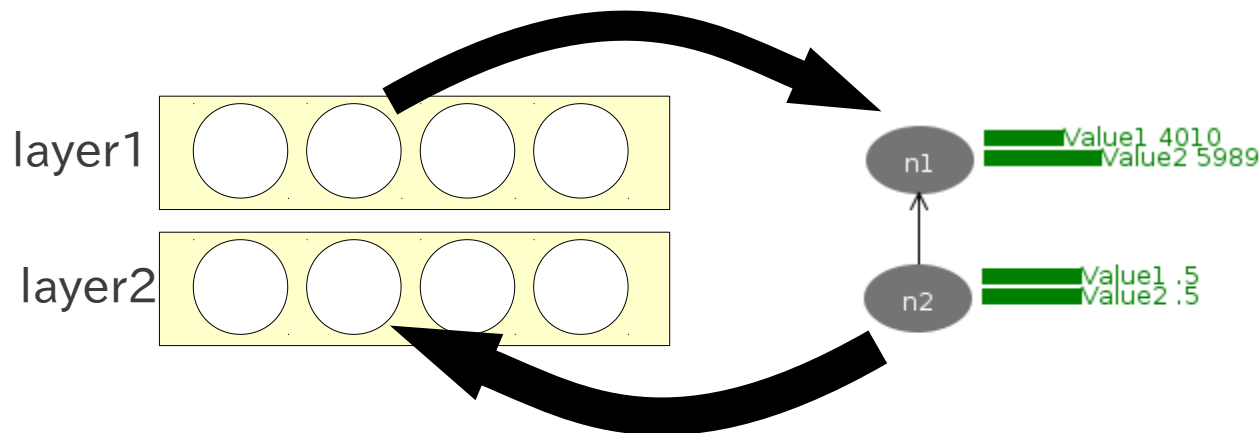
Weka側

さきほど読み込んだ ベイズアンネット

例 layer1 の i 番目の要素にエビデンスがセットされた場合

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

layer1 の i 番めの要素が 右側の n1 に対応



addReadMappingメソッドで BayesianCalculatorに登録

addWriteMappingメソッドで BayesianCalculatorに登録

layer2 の i 番めの要素が 右側の n2 に対応

```
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. BayesianCalculatorオブジェクトを登録

後は, BayesianCalculatorをMusicRepresentationに登録すれば
準備完了

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

5. MusicRepresentationにエビデンスをセット

- レイヤー「layer1」の最初の要素に"A"をエビデンスとしてセット
 - レイヤー「layer2」の最初の要素が自動的に更新されればOK

```
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"))
```

- レイヤー「layer1」の2番めの要素に"B"をエビデンスとしてセット
 - レイヤー「layer2」の2番めの要素が自動的に更新されればOK

```
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"))
```

プログラムリスト 完全版

```
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() {
}
```

応用例

- メロディレイヤーとコードレイヤーを用意して、メロディレイヤーの各要素にエビデンスをセットしてコードの要素を推論すれば、自動和声づけができる

おわりに

この説明書で扱わなかったこと

- MusicXML, DeviationInstanceXML, SCCXML以外のXML
 - 旋律の階層構造を記述するMusicApexXMLなどがある
- 外部のMIDIデバイスの利用
 - 外部のMIDI音源やMIDIキーボードを簡単に選ぶことができる
- MIDIファイルの再生とリアルタイムMIDI処理の連携
 - MIDIキーボードが打鍵される度に, MIDIファイルの再生位置を取得して処理を切り替えたり, 様々なことができる
- 音響信号処理
 - WAVファイルやマイクからの入力音声に対してフーリエ変換などをすることができます
- 行列計算
 - Groovyの演算子オーバーロードと組み合わせると便利

お問い合わせ先

- Web: <http://cmx.sourceforge.jp/>
<http://sourceforge.jp/projects/cmx/>
- メール: kitahara [at] kthrlab.jp