JMUSIC

LA LIBRERIA JMUSIC

- JMusic è una libreria Java distribuita con licenza GNU GPL
- È un progetto nato alla Queensland University of Technology (QUT), in Australia
- Si può utilizzare per comporre musica, per analizzare musica esistente, o per creare strumenti musicali sintetici
- Può importare/esportare file audio e MIDI
- http://explodingart.com/jmusic: dal sito si può scaricare il file jar della libreria, da includere nei progetti

PRIMO ESEMPIO

- Suonare un C4 della durata di un intero, via MIDI

```
import jm.JMC;
import jm.music.data.*;
import jm.util.Play;

public class C4 implements JMC {
  public static void main(String[] args) {
    Play.midi(new Note(C4, WHOLE_NOTE));
  }
}
```

- `implements JMC` è usato perché l'interfaccia jm.JMC contiene le dichiarazioni di numerose costanti (come `C4`, `WHOLE_NOTE`) utili nella maggior parte dei software scritti con la libreria jMusic

- Viene usata un'implementazione del metodo Play.midi che accetta in ingresso una singola nota (ce ne sono molti altri)

- Alcuni parametri sono quelli di default (volume, timbro, ecc.)

STRUTTURE DI BASE

Le informazioni in jMusic sono strutturate nelle classi seguenti:

- `Score`: contiene più parti
  - `Part`: contiene più frasi
    - `Phrase`: contiene più note
      - `Note`: contiene le informazioni della singola nota

- Per ogni nota si possono specificare i parametri di: pitch, dinamica, valore ritmico, pan, durata assoluta, offset
- La frase può essere vista come la singola voce di uno strumento: contiene un java.util.Vector di note
- La parte accorpa più frasi (voci) di un singolo strumento; ha una descrizione testuale, un canale e uno strumento MIDI
- Lo Score ha un titolo testuale e contiene un java.util.Vector di parti

COSTANTI PRESENTI

- Numerose costanti permettono di utilizzare termini mnemonici al posto dei valori numerici per pitch, durate, articolazioni, oscillatori, ecc.

- Al link http://explodingart.com/jmusic/jmDocumentation/constant-values.html è presente una lista di tutte le costanti definite

- Dichiarando nella propria classe l'implementazione dell'interfaccia `jm.JMC` si può prescindere dalla singola classe di implementazione delle costanti (ad es. invece di usare `jm.constants.Pitches.A4` si può usare `A4`)

CLASSE NOTE - INTRODUZIONE

- L'altezza è impostabile utilizzando sia gli interi corrispondenti ai pitch MIDI (0..127), sia costanti presenti (a5, A5), sia frequenze in Hz

- La durata è esprimibile con un valore double (1.0 corrisponde alla durata di un quarto, 0.5 all'ottavo, ecc.) oppure usando una delle costanti presenti (QUARTER_NOTE, QN, CROTCHET, C, ecc.)

- Il volume è un valore intero che varia tra 0 (silenzio) e 127 (volume massimo) (esistono anche in questo caso costanti)

- Il pan è un valore double variabile tra 0.0 (solo canale sinistro) e 1.0 (solo canale destro) (esistono anche in questo caso costanti)

CLASSE NOTE - COSTRUTTORI

- Costruttore base:
  - Note(): assegna valori di default a tutti i parametri (C4 da un quarto, volume 85, pan 0.5)
- Costruttori che come primo parametro richiedono il pitch MIDI:
  - Note(int pitch, double rhythmValue)
  - Note(int pitch, double rhythmValue, int dynamic)
  - Note(int pitch, double rhythmValue, int dynamic, double pan)
- Costruttori che come primo parametro richiedono la frequenza in Hz:
  - Note(double frequency, double rhythmValue)
  - Note(double frequency, double rhythmValue, int dynamic)
  - Note(double frequency, double rhythmValue, int dynamic, double pan)
- Costruttore che come primo parametro richiede il nome della nota
  - Note(String note)
    - I valori accettati sono: C, C#, D, Eb, E, F, F#, G, Ab, A, Bb, B (ottava centrale)
    - Se il valore passato non è fra quelli accettati viene creata una nota con pitch 0
    - Gli altri parametri sono quelli di default

COSTANTI PITCH

- Note naturali:
  - c0, C0, d0, D0, …, g9, G9 – *pitch 12, 12, 14, 14, …, 127, 127*
  - {Nota}{Ottava}
- Note con diesis:
  - cs0, CS0… – *pitch 13, 13…*
  - {Nota}S(harp){Ottava}
- Note con bemolle:
  - cf0, CF0… – *pitch 11, 11…*
  - {Nota}F(lat){Ottava}
- Note al di sotto del pitch 12:
  - cn1, CN1… – *pitch 0, 0…*
  - {Nota}[S|F]N(egative)1
- Note degeneri (pause):
  - REST – *pitch -2147483648 (Integer.MIN_VALUE)*

COSTANTI DURATE

Figura ritmica	Costante (brit.)	Costante (brit. abbr.)	Costante (amer.)	Costante (amer. abbr.)	Valore numerico
	SEMIBREVE	SB	WHOLE_NOTE	WN	4.0
	DOUBLE_DOTTED_MINIM	DDM, MDD	DOUBLE_DOTTED_HALF_NOTE	DDHN	3.5
	DOTTED_MINIM	DM, MD	DOTTED_HALF_NOTE	DHN	3.0
	MINIM	M	HALF_NOTE	HN	2.0
	DOUBLE_DOTTED_CROTCHET	DDC, CDD	DOUBLE_DOTTED_QUARTER_NOTE	DDQN	1.75
	DOTTED_CROTCHET	DC, CD	DOTTED_QUARTER_NOTE	DQN	1.5
 (sotto terzina)	MINIM_TRIPLET	MT	HALF_NOTE_TRIPLET	HNT	1.33333...
	CROTCHET	C	QUARTER_NOTE	QN	1.0
	DOUBLE_DOTTED_QUAVER	DDQ, QDD	DOUBLE_DOTTED_EIGHTH_NOTE	DDEN	0.875
	DOTTED_QUAVER	DQ, QD	DOTTED_EIGHTH_NOTE	DEN	0.75
 (sotto terzina)	CROTCHET_TRIPLET	CT	QUARTER_NOTE_TRIPLET	QNT	0.66666...
	QUAVER	Q	EIGHTH_NOTE	EN	0.5
	DOTTED_SEMI_QUAVER	SQD	DOTTED_SIXTEENTH_NOTE	DSN	0.375
 (sotto terzina)	QUAVER_TRIPLET	QT	EIGHTH_NOTE_TRIPLET	ENT	0.33333...
	SEMI_QUAVER	SQ	SIXTEENTH_NOTE	SN	0.25
 (sotto terzina)	SEMI_QUAVER_TRIPLET	SQT	SIXTEENTH_NOTE_TRIPLET	SNT	0.1666...
	DEMI_SEMI_QUAVER	DSQ	THIRTYSECOND_NOTE	TN, TSN	0.125
 (sotto terzina)	DEMI_SEMI_QUAVER_TRIPLET	DSQT	THIRTYSECOND_NOTE_TRIPLET	TNT, TSNT	0.08333...

COSTANTI DINAMICA E PAN

- Costanti dinamica:
  - SILENT: 0
  - PPP: 10
  - PP, PIANISSIMO: 25
  - P: 50
  - MP, MEZZO_PIANO: 60
  - MF, MEZZO_FORTE: 70
  - F, FORTE: 85
  - FF, FORTISSIMO: 100
  - FFF: 120

- Costanti pan:
  - PAN_CENTER, PAN_CENTRE: 0.5
  - PAN_LEFT: 0.0
  - PAN_RIGHT: 1.0

NOTE: METODI

- `int getPitch(), setPitch(int pitch)`
  Restituisce/imposta il pitch
- `double getFrequency(), setFrequency(double frequency)`
  Restituisce/imposta la frequenza (in Hz)
- `int getDynamic(), setDynamic(int dynamic)`
  Restituisce/imposta il volume
- `double getPan(), setPan(double pan)`
  Restituisce/imposta il pan
- `double getRhythmValue(), setRhythmValue(double rhythmValue)`
  Restituisce/imposta la durata
- `double getDuration(), setDuration(double duration)`
  Restituisce/imposta la durata reale (dipende dal tipo di articolazione)

PAUSE

- Le pause possono essere create come note con pitch uguale alla costante REST (o al valore -2147483648 (Integer.MIN_VALUE))

- Esiste anche la classe Rest, che estende Note e ha i seguenti costruttori:
  - Rest()
  - Rest(double rhythmValue)

PHRASE

- Per creare sequenze di note si utilizza la classe Phrase
- Costruttori:
  - Phrase(): frase con impostato il flag di "append" a true
  - Phrase(double startTime): frase che inizia al tempo specificato (imposta il flag di append a false)
  - Phrase(double startTime, int instrument): (vedere slide seguente)
  - Phrase(Note note): frase con la prima nota specificata
  - Phrase(Note[] notes): frase con le note specificate
  - Phrase(Note[] notes, String title)
  - Phrase(Note note, double startTime)
  - Phrase(Note note, double startTime, String title)
  - Phrase(String title)
  - Phrase(String title, double startTime)
  - Phrase(String title, double startTime, int instrument)
- Il flag di "append" impostato a true fa sì che quando la frase viene aggiunta a una parte, venga accodata a quest'ultima

STRUMENTI

- Sia per le frasi che per le parti è possibile specificare lo strumento che riprodurrà le note aggiunte, mediante il corrispondente valore General MIDI

- Sono definite le costanti corrispondenti a nomi degli strumenti MIDI:
  - PIANO, ACOUSTIC_GRAND: 0
  - BRIGHT_ACOUSTIC: 1
  - ELECTRIC_GRAND: 2
  - HONKYTONK, HONKYTONK_PIANO: 3
  - ELECTRIC_PIANO, RHODES: 4
  - …

PHRASE: METODI PRINCIPALI

- add(Note note), addNote(Note note)
- addNote(int pitch, double rhythmValue): parametri pitch e valore ritmico
- addChord(int[] pitches, double rhythmValue)
- addRest(Rest rest)
- addNoteList(…): vari metodi per aggiungere array di note (v. documentazione)
- setNote(Note note, int index): sostituisce una nota nella lista
- removeNote(int index): rimuove una nota dalla lista
- removeNote(Note note): rimuove la prima occorrenza della nota specificata
- removeLastNote()
- setAppend(boolean append), boolean getAppend(): imposta/legge il valore del flag di "append"

PHRASE: ALTRI METODI

- int length(), int size(): restituisce il numero di note
- setInstrument(int instrument), int getInstrument()
- setStartTime(double startTime), double getStartTime()
- setTempo(double tempo): imposta i BPM
- int getHighestPitch()
- int getLowestPitch()
- double getLongestRhythmValue()
- double getShortestRhythmValue()
- int[] getPitchArray(): tutti i valori di pitch delle note presenti
- double[] getRhythmArray(): tutti i valori ritmici delle note presenti

PART

- Gruppi di frasi formano parti (classe Part)
- Costruttori:
  - Part()
  - Part(int instrument)
  - Part(int instrument, int midiChannel)
  - Part(Phrase phrase)
  - Part(Phrase phrase, String title)
  - Part(Phrase phrase, String title, int instrument)
  - Part(Phrase phrase, String title, int instrument, int midiChannel)
  - Part(Phrase[] phrases)
  - Part(Phrase[] phrases, String title)
  - Part(Phrase[] phrases, String title, int instrument)
  - Part(Phrase[] phrases, String title, int instrument, int midiChannel)
  - Part(String title)
  - Part(String title, int instrument)
  - Part(String title, int instrument, int midiChannel)
  - Part(String title, int instrument, int midiChannel, Phrase phrase)
  - Part(String title, int instrument, Phrase phrase)

PART: METODI

- add(Phrase phrase), addPhrase(Phrase phrase): aggiunge la frase (attenzione al flag "append" della stessa)
- addNote(Note note, double startTime)
- appendPhrase(Phrase phrase)
- addPhraseList(Phrase[] phrases)
- removeAllPhrases()
- removeLastPhrase()
- removePhrase(int index): rimuove la frase nella posizione specificata
- removePhrase(Phrase phrase): rimuove la prima occorrenza della frase specificata
- empty(): rimuove le frasi presenti
- clean(): rimuove le frasi vuote presenti
- int length(), int size(), int getSize(): numero di frasi presenti
- int getHighestPitch()
- int getLowestPitch()
- double getLongestRhythmValue()
- double getShortestRhythmValue()

SCORE

- Varie parti compongono uno Score
- Costruttori:
  - Score()
  - Score(double tempo): specifica del tempo in BPM
  - Score(Part part)
  - Score(Part part, String title)
  - Score(Part part, String title, double tempo)
  - Score(Part[] parts)
  - Score(Part[] parts, String title)
  - Score(Part[] parts, String title, double tempo)
  - Score(String title)
  - Score(String title, double tempo)
  - Score(String title, double tempo, Part part)

SCORE: METODI

- add(Part part), addPart(Part part)
- addPartList(Part[] parts)
- removeAllParts()
- removeLastPart()
- removePart(int index): rimuove la parte nella posizione specificata
- removePart(Part part): rimuove la prima occorrenza della parte specificata
- empty(): rimuove le parti presenti
- clean(): rimuove le parti vuote presenti
- int length(), int size(), int getSize(): numero di parti presenti
- int getHighestPitch()
- int getLowestPitch()
- double getLongestRhythmValue()
- double getShortestRhythmValue()

MIDI E GRAFICA

- La classe jm.util.Play presenta alcuni metodi statici che possono essere utilizzati per suonare via MIDI le strutture dati create: il metodo midi() riceve in input Note, Phrase, Part o Score e li riproduce

- La classe jm.util.Write si può usare invece per salvare l'output su file attraverso i metodi statici midi(), che ricevono in input Note, Phrase, Part o Score e (opzionalmente) un secondo parametro indicante il nome del file MIDI su cui salvare (se non viene specificato viene usato il titolo dell'oggetto passato)

- La classe jm.util.View presenta una serie di metodi statici che creano una finestra spartana di rappresentazione di note, frasi, parti e/o score:
  - histogram(): istogramma delle note presenti
  - internal(), print(): stampa su standard output in formato testuale il contenuto
  - notate(), notation(): rappresentazione notazionale
  - pianoRoll(), show(): rappresentazione in piano roll

ESEMPIO

(Chorale.java)

Nell'applicazione:

- Viene creato un corale di Bach
- Vengono usati array per inizializzare pitch e durate delle note delle 4 voci (soprano, contralto, tenore e basso)
- Vengono usati gli strumenti MIDI 52 e 53, corrispondenti ai cori AAH e OOH
- Viene mostrata una rappresentazione in piano-roll
- Viene salvato il file MIDI

ESERCIZIO

(Random.java)

Scrivere un'applicazione che produca in output un file MIDI con le seguenti specifiche:

- È presente il singolo strumento HAMMOND_ORGAN
- Viene creata una sequenza di accordi la cui durata è quella della voce di soprano dell'esempio precedente
- Ogni accordo è formato da 3 note:
  - La prima nota è scelta casualmente tra il pitch 48 e il pitch 84 (escluso)
  - La seconda nota è scelta trasponendo casualmente in alto o in basso la prima di una terza maggiore
  - La terza nota è scelta trasponendo casualmente in alto o in basso la prima di una quinta giusta

LA CLASSE MOD

- La classe jm.music.tools.Mod presenta dei metodi statici per la manipolazione di strutture musicali precedentemente create
- Alcuni metodi che modificano oggetti di classe Phrase, Part o Score:
  - append(obj1, obj2): accoda il secondo oggetto al primo
  - repeat(obj[, int times]): accoda l'oggetto a se stesso una o più volte
  - elongate(obj, double scaleFactor): modifica la durata delle note con un fattore di scala
  - retrograde(Phrase phrase): inverte la sequenza di note della frase
  - invert(Phrase phrase): inverte specularmente i pitch delle note, considerando la prima come centro e rispetto al numero di semitoni di distanza dal centro
  - diatonicInvert(Phrase phrase, int[] scale, int root): inverte specularmente i pitch delle note, considerando la prima come centro e la scala specificata (root=0: do, root=1: do#...)
  - rotate(Phrase phrase[, int numSteps]): ruota la sequenza di note di uno o più posti
  - palindrome(Phrase phrase[, boolean repeatLastNote]): accoda alla frase le note presenti prese al contrario (si può specificare se ripetere l'ultima nota)
  - shuffle(obj): randomizza la sequenza delle note presenti, frase per frase
  - transpose(obj, int transposition): traspone il pitch del numero di semitoni specificato
  - transpose(obj, int transposition, int[] mode, int key): traspone il pitch del numero di gradi specificato, usando un certo modo (key=0: do, key=1: do#...)
  - increaseDynamic(obj, int amount): modifica il volume sommando il valore specificato
  - bounce(obj): modifica il pan delle note per alternare nota per nota i canali sinistro/destro

COSTANTI USATE PER LE SCALE

- In alcuni metodi possono essere specificati array di interi che indicano una particolare scala musicale. Di seguito le costanti già presenti:
  - CHROMATIC_SCALE = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}
  - MAJOR_SCALE = {0, 2, 4, 5, 7, 9, 11}
  - MINOR_SCALE = {0, 2, 3, 5, 7, 8, 10}
  - HARMONIC_MINOR_SCALE = {0, 2, 3, 5, 7, 8, 11}
  - MELODIC_MINOR_SCALE = {0, 2, 3, 5, 7, 8, 9, 10, 11} (mix of ascend and descend)
  - NATURAL_MINOR_SCALE = {0, 2, 3, 5, 7, 8, 10}
  - DIATONIC_MINOR_SCALE = {0, 2, 3, 5, 7, 8, 10}
  - AEOLIAN_SCALE = {0, 2, 3, 5, 7, 8, 10}
  - DORIAN_SCALE = {0, 2, 3, 5, 7, 9, 10}
  - LYDIAN_SCALE = {0, 2, 4, 6, 7, 9, 11}
  - MIXOLYDIAN_SCALE = {0, 2, 4, 5, 7, 9, 10}
  - PENTATONIC_SCALE = {0, 2, 4, 7, 9}
  - BLUES_SCALE = {0, 2, 3, 4, 5, 7, 9, 10, 11}
  - TURKISH_SCALE = {0, 1, 3, 5, 7, 10, 11}
  - INDIAN_SCALE = {0, 1, 1, 4, 5, 8, 10}

ESERCIZIO

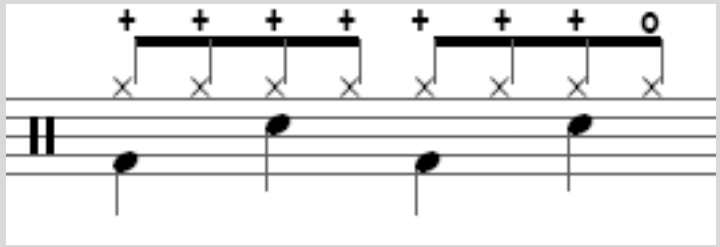
(Drums.java)

Scrivere un'applicazione che produca in output un file MIDI con le seguenti specifiche:

- Si utilizzi il canale MIDI 10 (percussioni)

- Sapendo che i timbri da utilizzare sono:
  - 36 - Bass drum
  - 38 - Snare drum
  - 42 - Hi-hat (closed)
  - 46 - Hi-hat (open)

  si crei un sequenza di 4 battute uguali a quella in figura



- Si simuli una variazione del tempo:
  - inizia a 60BPM
  - ogni quarto dura il 95% del quarto precedente

OLTRE IL MIDI: GENERARE SUONI

- Finora il suono è stato sempre prodotto dal sintetizzatore MIDI, ma jMusic fornisce una serie di classi per la generazione e il trattamento di segnali audio

- Dal sito è possibile scaricare una serie di strumenti già pronti, che estendono la classe base jm.audio.Instrument:
  - Segnali sinusoidali, triangolari, dente di sega…
  - Sintesi AM
  - Sintesi FM
  - Sintesi additiva
  - Sintesi granulare
  - Ring Modulation
  - …

ESEMPI DI STRUMENTI FORNITI

- SimpleSineInst: suono sinusoidale
- SawtoothInst: dente di sega
- NoiseInst: rumore bianco
  - [tutti con costruttori: new *strumento*(int sampleRate)]
- AddSynthInst: sintesi additiva
  - new AddSynthInst(int sampleRate, double[] overtoneRatios, double[] overtoneVolumes, double[][] envPointArray)
  - Es.: new AddSynthInst(44100, {1.0f, 3.0f, 5.0f},
    {1.0f, 0.5f, 0.35f},
    {{0.0, 0.0, 0.05, 1.0, 0.15, 0.4, 0.9, 0.3, 1.0, 0.0},
    {0.0, 0.0, 0.07, 1.0, 0.15, 0.6, 0.8, 0.2, 1.0, 0.0},
    {0.0, 0.0, 0.03, 1.0, 0.25, 0.3, 0.9, 0.5, 1.0, 0.0}})
    Frquenze f, 3*f, 5*f, con volumi 1, 0.5, 0.35, e con inviluppi corrispondenti

ESEMPI DI STRUMENTI FORNITI

- BreathyFluteInst: flauto con componenti di rumore bianco
- PluckInst: corda pizzicata
- SuperSawInst: denti di sega multipli con frequenze leggermente fuori intonazione
- TimpaniInst: suono percussivo di timpani
- VibesInst: vibrafono
- ecc. (v. https://explodingart.com/jmusic/Instruments.html)

PRIMO ESEMPIO AUDIO

```
import jm.music.data.*;
import jm.JMC;
import jm.audio.*;
import jm.util.*;

public class WaveformExample implements JMC {

  public static void main(String[] args) {
    // make a jMusic score
    Note n = new Note(C4, MINIM);
    Score score = new Score(new Part(new Phrase(n)));

    // set up an audio instrument
    Instrument sineWave = new SimpleSineInst(44100);

    // render audio file of the score
    Write.au(score, "WaveformExample.au", sineWave);
  }
}
```

ESEMPIO: USO DI CAMPIONI ESTERNI

(AudioDrums.java)

- Vengono usati 4 campioni audio per riprodurre 4 suoni di batteria
- Vengono create 4 parti: in questo caso il secondo parametro del costruttore di Part identifica la corrispondenza con l'array (drumKit) di strumenti usati per il rendering audio
- Il costruttore di SimpleSampleInst riceve in ingresso il nome del file contenente il sample da caricare e la frequenza considerata di base del campione
- Viene usato l'array predefinito FRQ, che converte da pitch MIDI a corrispondente frequenza in Hz. Questa operazione consente di mappare il pitch MIDI corrispondente allo strumento General MIDI corretto sul campione caricato: in questo modo è possibile poi salvare sia il MIDI che l'AU con gli stessi strumenti
- L'hi-hat viene suonato ogni sedicesimo, ma con volume random e con la probabilità del 10% che suoni aperto sul secondo e quarto sedicesimo

ESERCIZIO

(RandomAudio.java)

Scrivere un'applicazione che produca in output un file audio con le seguenti specifiche:

- Si utilizzino 2 strumenti: SineInst e PluckInst (entrambi hanno un costruttore in cui specificare la frequenza di campionamento (usare 44100))
- Si consideri la sequenza dei numeri di Fibonacci (F[0] = 1; F[1] = 1; F[n] = F[n-1] + F[n-2]) per calcolare alcuni parametri (vedi sotto)
- La sequenza dei singoli pitch P[i] delle note deve essere uguale alle note della scala pentatonica (partendo da C4) prese modulo 5 rispetto al numero di Fibonacci in posizione corrispondente (P[i] = C4 + PENTATONIC_SCALE[F[i] % 5])
- Il timbro usato per la nota i-esima dipende dal fatto che F[i] sia pari o dispari
- La durata delle note varia casualmente tra 1.0 e 2.0, la dinamica tra 60 e 120 e il pan tra 0 e 1
- Si suonino le prime 20 note

CREAZIONE DI NUOVI STRUMENTI

- Come nel caso di jSyn e di altri ambiti, il paradigma usato per implementare nuovi strumenti è quello della creazione di catene di moduli (AudioUnit) che vanno a formare l'output finale
- Va dichiarata una classe che estende jm.audio.Instrument
- Va implementato all'interno della classe il metodo astratto createChain()
- Nel metodo createChain() viene creata la sequenza di stadi che porta alla produzione del suono finale
- Ogni stadio, corrispondente all'istanziazione di una classe, per convenzione riceve in ingresso il primo parametro identificante lo stadio precedente a cui collegarsi
- L'ultimo stadio, nella maggior parte dei casi, è un'istanza della classe SampleOut, che produce i dati finali sui campioni
  Es.: SampleOut sout = new SampleOut(env);

ESEMPIO DI INSTRUMENT

(SampleInstrument.java)

```
public void createChain() {
  Oscillator wt = new Oscillator(this,
    Oscillator.SINE_WAVE,
    sampleRate, channels);
  Envelope env = new Envelope(wt,
    new EnvPoint[] {
      new EnvPoint(0.0f, 0.0f),
      new EnvPoint(0.5f, 1.0f),
      new EnvPoint(0.9f, 0.4f),
      new EnvPoint(1.0f, 0.0f)
    });
  SampleOut sout = new SampleOut(env);
}
```

- Nell'implementazione del metodo createChain viene creata una prima AudioUnit che corrisponde a un oscillatore sinusoidale
- sampleRate e channels sono attributi valorizzati nel costruttore
- Viene creato un inviluppo d'ampiezza (primo valore tra 0 e 1 indicante il tempo, secondo valore tra 0 e 1 indicante l'ampiezza)
- Il suono viene mandato in uscita con l'oggetto SampleOut

ESEMPI DI AUDIOOBJECT

- Oscillator
  Usato sia come oggetto primario, sia come come oggetto la cui frequenza/ampiezza viene modulata da oggetti precedenti
  - Oscillator(Instrument inst, [int waveType[, int sampleRate[, int channels]]])
    Oggetto primario, in cui waveType definisce il tipo di forma d'onda (costanti SINE_WAVE, SAWTOOTH_WAVE, SQUARE_WAVE, TRIANGLE_WAVE…)
  - Oscillator(AudioObject ao, int waveType, int choice)
    Oggetto il cui parametro specificato in choice viene modulato da oggetti precedenti (costanti FREQUENCY, AMPLITUDE)
- Noise(Instrument inst[, int noiseType[, int sampleRate[, int channels]]])
  Usato per generare vari tipi di rumore (costanti per noiseType: WHITE_NOISE, BROWN_NOISE, GAUSSIAN_NOISE, FRACTAL_NOISE…)
- Filter(AudioObject ao, double freq, int type)
  Filtro passa-basso/alto, in cui specificare frequenza e tipo (costanti HIGH_PASS, LOW_PASS)

ESEMPI DI AUDIOOBJECT

- Envelope(AudioObject ao, EnvPoint[] graphPoints)
  Usato per definiri inviluppi, con i punti definiti come EnvPoint, connessi tramite linee
- EnvPoint(float x, float y)
  Singoli punti di un inviluppo (entrambi i valori variano tra 0 e 1)
- Add(AudioObject[] ao)
  Somma le unità in ingresso
- Volume(AudioObject ao, double volume)
  Imposta il volume dei campioni prodotto dall'oggetto in ingresso
- Value(Instrument inst, int sampleRate, int channels, {vedi sotto})
  Usato per passare delle costanti alle unità: ha due costruttori, che si diversificano per l'ultimo parametro in ingresso:
  - float fixedValue: viene passato un valore fisso
  - int noteAttribute: viene legato a un parametro della nota suonata, usando una delle costanti NOTE_DURATION, NOTE_DYNAMIC, NOTE_PITCH, NOTE_RHYTHM_VALUE

ESERCIZIO

(SampleInstrument2.java, SampleInstrumentScore.java)

Implementare un nuovo strumento composto dalla somma di:

- Oscillatore sinusoidale, volume relativo 1, frequenza relativa 1
- Oscillatore onda quadra, volume relativo 0.5, frequenza relativa 1.02
- Rumore bianco filtrato con passa-alto a 4000 Hz, volume relativo 0.3

Inviluppo d'ampiezza con valori (tempo > ampiezza):

0.0 > 0.0

0.2 > 1.0

0.7 > 0.7

0.9 > 0.1

1.0 > 0.0

JMUSIC E REALTIME

- jMusic supporta la creazione di suoni in realtime attraverso classi apposite

- Nell'uso in realtime non si utilizzano le strutture Phrase, Part, Score, ma solo la generazione di singole note

- La classe base è RTLine, che è concettualmente simile a Phrase, anche se, invece di memorizzare note, le riproduce direttamente
  - Costruttore: RTLine(Instrument[] instrs)
  - Viene passato un array di strumenti ma si assume che siano diverse istanze dello stesso strumento, con ugual numero di canali e sample rate, usate per suonare accordi o creare effetti

- Uno o più oggetti RTLine devono essere poi fatti confluire in un RTMixer, che si occupa di fornire l'output audio
  - La generazione viene avviata richiamando il metodo begin()

PRIMO ESEMPIO REALTIME

(RealtimeTest.java)

```
import jm.JMC;
import jm.music.data.*;
import jm.audio.*;
import jm.music.rt.*;

public final class RealtimeTest extends RTLine implements JMC {

    public static void main(String[] args) {
        Instrument inst = new SimpleFMInst(44100, 800, 34.4);
        new RealtimeTest(new Instrument[] {inst});
    }

    public RealtimeTest(Instrument[] insts) {
        super(insts);
        RTMixer mixer = new RTMixer(new RTLine[] {this});
        mixer.begin();
    }

    @Override
    public Note getNextNote() {
        Note n = new Note((int)(Math.random() * 12 + 60), QN);
        return n;
    }
}
```

- Il costruttore richiama come prima cosa il costruttore ereditato
- Il metodo getNextNote() (notare l'override) è quello che fornisce le note da suonare al motore realtime. Ogni volta che la nota attuale ha finito di essere suonata, il motore richiama nuovamente questo metodo per ottenere la nota successiva

MODIFICARE PARAMETRI IN REALTIME

- Per modificare i parametri relativi a una RTLine è possibile fare l'override del metodo externalAction:

```
@Override
public void externalAction(Object obj, int actionNumber) { … }
```

- In ingresso si riceve un oggetto (spesso un componente della GUI che può essere castato per ottenere informazioni) e un id che convenzionalmente indica l'oggetto passato

- Chi si occupa di informare la RTLine del fatto che qualcosa è cambiato è l'RTMixer, con il metodo `actionLines(Object obj, int actionNumber)`
  - I parametri sono quelli esposti sopra
  - Es.: nei gestori dell'evento di clic su due diversi pulsanti, rispettivamente:

```
mixer.actionLines(jButtonChangeFreq, 1);
mixer.actionLines(jButtonChangeAmp, 2);
```

ESERCIZIO

(RealtimeController.java)

Implementare un'applicazione grafica contenente un solo slider

- Viene creata una RTLine contenente uno strumento:
  new SimpleFMInst(44100, 800, 34.4)

- Viene generata una sequenza continua di note QUAVER, il cui pitch dipende dal valore dello slider: il valore corrisponde al pitch centrale P di note casuali scelte tra P-6 e P+6