BlueJam :: Core Package Documentation

March 23, 2008

Contents

1 Package net.parallaxed.bluejam.playback **3**
- 1.1 Interfaces . . . 4
  - 1.1.1 Interface **Listener** . . . 4
  - 1.1.2 Interface **Player** . . . 4
- 1.2 Classes . . . 5
  - 1.2.1 Class **MIDI** . . . 5
  - 1.2.2 Class **MIDI.GREATER_THAN** . . . 6
  - 1.2.3 Class **MIDI.LESS_THAN** . . . 6

2 Package net.parallaxed.bluejam.grammar **7**
- 2.1 Classes . . . 8
  - 2.1.1 Class **ModelParser** . . . 8
  - 2.1.2 Class **PitchModel** . . . 8

3 Package net.parallaxed.bluejam **11**
- 3.1 Interfaces . . . 13
  - 3.1.1 Interface **Function** . . . 13
  - 3.1.2 Interface **Heuristic** . . . 13
  - 3.1.3 Interface **NoteSequence** . . . 14
  - 3.1.4 Interface **Terminal** . . . 16
- 3.2 Classes . . . 16
  - 3.2.1 Class **Accidental** . . . 16
  - 3.2.2 Class **Evolve** . . . 17
  - 3.2.3 Class **EvolveHeuristic** . . . 19
  - 3.2.4 Class **HeuristicCollection** . . . 19
  - 3.2.5 Class **HeuristicCollection.SELECTION_TYPE** . . . 21
  - 3.2.6 Class **Individual** . . . 21
  - 3.2.7 Class **IndividualParameters** . . . 24
  - 3.2.8 Class **JamParamters** . . . 25
  - 3.2.9 Class **JamParamters.Config** . . . 28
  - 3.2.10 Class **Mutable** . . . 29
  - 3.2.11 Class **Note** . . . 29
  - 3.2.12 Class **NoteCollection** . . . 34
  - 3.2.13 Class **NoteLeaf** . . . 37
  - 3.2.14 Class **NoteLeafSet** . . . 42
  - 3.2.15 Class **NoteTree** . . . 44
  - 3.2.16 Class **Pitch** . . . 48
  - 3.2.17 Class **Population** . . . 50
  - 3.2.18 Class **PopulationParameters** . . . 54

3.2.19 Class **Rhythm** . . . 56
3.2.20 Class **Scale** . . . 58
3.2.21 Class **Scale.BLUES** . . . 59
3.2.22 Class **Scale.MAJOR** . . . 59
3.2.23 Class **Scale.MINOR** . . . 60
3.2.24 Class **ScaledSet** . . . 60
3.2.25 Class **SequenceParameters** . . . 62
3.2.26 Class **TerminalSet** . . . 63
3.2.27 Class **TreeParser** . . . 64

4 Package net.parallaxed.bluejam.pd **66**
4.1 Classes . . . 67
4.1.1 Class **Configure** . . . 67
4.1.2 Class **Jam** . . . 68

5 Package net.parallaxed.bluejam.evolution **70**
5.1 Interfaces . . . 72
5.1.1 Interface **Breeder** . . . 72
5.1.2 Interface **IndividualEvaluator** . . . 72
5.1.3 Interface **IndividualSelector** . . . 73
5.1.4 Interface **NoteSequenceInitializer** . . . 73
5.2 Classes . . . 74
5.2.1 Class **FitnessContour** . . . 74
5.2.2 Class **FitnessDistance** . . . 75
5.2.3 Class **FitnessInterval** . . . 75
5.2.4 Class **FitnessRandom** . . . 76
5.2.5 Class **FitnessStacked** . . . 77
5.2.6 Class **FitnessType** . . . 77
5.2.7 Class **Genotype** . . . 78
5.2.8 Class **HeuristicSelectionType** . . . 79
5.2.9 Class **InitializationType** . . . 80
5.2.10 Class **InitializeGrow** . . . 81
5.2.11 Class **InitializeHeuristic** . . . 81
5.2.12 Class **InitializeRandom** . . . 82
5.2.13 Class **NoteContext** . . . 82
5.2.14 Class **NoteContext.Contour** . . . 84
5.2.15 Class **PropertyFactory** . . . 84
5.2.16 Class **RhythmInitializer** . . . 85
5.2.17 Class **SelectionParameters** . . . 86
5.2.18 Class **SelectionType** . . . 87
5.2.19 Class **SelectProportional** . . . 87
5.2.20 Class **SelectTournament** . . . 88
5.2.21 Class **TreeBreeder** . . . 89

Chapter 1

Package net.parallaxed.bluejam.playback

Package Contents *Page*

Interfaces

Listener 4
This interface can be implemented by classes that wish to receive notifications from evolving populations.

Player 4
This interface is implemented by classes that produce output (default: Evolve).

Saveable 5
Returns a stringed version of this note compatible with the heuristic format.

Classes

MIDI 6
This is a static class used primarily for converting internal note representations into MIDI numbers and vice versa.

MIDI.GREATER_THAN 6
...no description...

MIDI.LESS_THAN 4
...no description...

1.1 Interfaces

1.1.1 Interface **Listener**

This interface can be implemented by classes that wish to receive notifications from evolving populations.

Declaration

public interface Listener

Methods

- *listen*
  `public void` **listen**`( net.parallaxed.bluejam.NoteSequence` **n** `)`
  - **Usage**
    - Called by the Evolving population when a candidate NoteSequence is ready to be played.
      A null noteSequence should be passed when no more generations will be calculated.
  - **Parameters**
    - `n` - The NoteSequence to be played.

1.1.2 Interface **Player**

This interface is implemented by classes that produce output (default: Evolve). These classes should run in their own threads and output NoteSequences to subscribed listeners.
Ideally, they should change their behaviour based on an established protocol for exchanging feedback between it's listeners.

Declaration

public interface Player

Methods

- *addListener*
  `public void` **addListener**`( net.parallaxed.bluejam.playback.Listener` **listener** `)`
- *feedback*
  `public void` **feedback**`( int` **feedback**`, net.parallaxed.bluejam.NoteSequence` **notes** `)`

1.1.3 Interface **Saveable**

Returns a stringed version of this note compatible with the heuristic format.

Declaration

public interface Saveable

Methods

- *stringify*
  `public String` **`stringify( )`**

1.2 Classes

1.2.1 Class **MIDI**

This is a static class used primarily for converting internal note representations into MIDI numbers and vice versa.
This class can discover if a particular pitch is relatively higher or lower than another given pitch, in the context of a single MIDI defined octave
(this does not work for notes spanning multiple octaves ...yet)
These values can be used at playback time.
TODO HIGH Uses a convoluted algorithm of O(n), simply cycling through the array - could be vastly improved.

Declaration

public final class MIDI **extends** java.lang.Object

Constructors

- *MIDI*
  `public` **`MIDI( )`**

Methods

- *noteToNumber*
  `public static float` **`noteToNumber(`** `net.parallaxed.bluejam.Note` **`n )`**
  - **Usage**
    - This method will return a MIDI number for the given note. This method DOES NOT validate that MIDI number (should be 0 <= n <= 127).

  - **Parameters**
    - `n` - The note to calculate for.
  - **Returns** - A MIDI number for the note.

- *numberToNote*
  `public static Note` **numberToNote**`( float` **noteNumber** `)`
  - **Usage**
    - Currently used by the Configure class in the PD implementation to tell what note is being set as the root pitch.
  - **Parameters**
    - `noteNumber` -
  - **Returns** - A Note instance configured to reflect the passed number.
  - **See Also**
    - `net.parallaxed.bluejam.Note` (in 1.2.1, page 5)

- *position*
  `public static int` **position**`( net.parallaxed.bluejam.Pitch` **pitch1** `)`

- *relative*
  `public static Pitch` **relative**`( net.parallaxed.bluejam.Pitch` **rootPitch**`, int` **relativeIndex** `)`

1.2.2 Class MIDI.GREATER_THAN

Declaration

public static class MIDI.GREATER_THAN
extends java.lang.Object

Constructors

- *MIDI.GREATER_THAN*
  `public` **MIDI.GREATER_THAN**`( )`

Methods

- *eval*
  `public boolean` **eval**`( net.parallaxed.bluejam.Pitch` **pitch1**`,`
  `net.parallaxed.bluejam.Pitch` **pitch2** `)`

1.2.3 Class MIDI.LESS_THAN

Declaration

public static class MIDI.LESS_THAN
extends java.lang.Object

Constructors

- *MIDI.LESS_THAN*
  `public` **MIDI.LESS_THAN**`( )`

Methods

- *eval*
  `public static boolean` **eval**`( net.parallaxed.bluejam.Pitch` **pitch1**`,`
  `net.parallaxed.bluejam.Pitch` **pitch2** `)`

Chapter 2

Package net.parallaxed.bluejam.grammar

Package Contents *Page*

Classes

ModelParser 8
This class parses flat-file Markov models which can be used by sets during the selection process to assign proportionate probabilities to possible notes in the scale.

PitchModel 8
The PitchModel class represents a 1-order Markov model describing a probability matrix of Pitches.

2.1 Classes

2.1.1 Class **ModelParser**

This class parses flat-file Markov models which can be used by sets during the selection process to assign proportionate probabilities to possible notes in the scale.
Models describe what's likely to happen given note X has already happened.

Declaration

public class ModelParser **extends** java.lang.Object

Constructors

- *ModelParser*
  `public` **ModelParser**`( java.io.File` **modelFile** `)`
  - **Usage**
    - Constructs a ModelParser. Use getModel() to extract the parsed model.
  - **Parameters**
    - `modelFile` - A reference to a File object for the model file.

- *ModelParser*
  `public` **ModelParser**`( java.lang.String` **modelFile** `)`
  - **Usage**
    - Constructs a ModelParser. Use getModel() to extract the parsed model.
  - **Parameters**
    - `modelFile` - The absolute path to the model file.

Methods

- *getModel*
  `public PitchModel` **getModel**`( )`
  - **Returns** - The model that has been parsed by this ModelParser, or null if no Model has been parsed.

2.1.2 Class **PitchModel**

The PitchModel class represents a 1-order Markov model describing a probability matrix of Pitches. The model is a lookup table that will return P(X—Y), where X and Y are a set of notes in a given context. For more complex models, this class could be written to include support for cases where Y is a series of notes, increasing the order of the model up to the number of notes in Y.
Models must be locked after loading before use, such that no further changes can be made to a model, and the model is deemed valid.
Please see additional documentation and the technical report for information regarding pitch models.

Declaration

public class PitchModel
extends java.util.HashMap

Serializable Fields

- private String _name
  –
- private boolean _editable
  –
- private ArrayList _pitchOrder
  –
- private PitchModel _original
  –

Constructors

- *PitchModel*
  `public` **PitchModel**`( java.lang.String` **name** `)`
  - **Usage**
    - Creates a PitchModel instance.
  - **Parameters**
    - `name` -

Methods

- *get*
  `public double` **get**`( net.parallaxed.bluejam.Pitch` **key** `)`
  - **Parameters**
    - `key` - The pitch to return the model for.
  - **Returns** - The model for the given pitch.

- *getPitch*
  `public Pitch` **getPitch**`( int` **index** `)`
  - **Usage**
    - Pitch order is preserved when adding to the model. Models remain square matrices, for every new pitch, a new row and new column is appended to the table.
  - **Parameters**

    - `index` - The index sought.
  - **Returns** - The pitch at the passed index of the model

- *put*
  `public double` **put**`( net.parallaxed.bluejam.Pitch` **pitch**`, double []` **model** `)`
  - **Usage**
    - Adds a single pitch, giving the probability series for other notes in the model following that pitch. This function will replace any existing pitches matching the passed pitch. The model must be editable to make changes.
  - **Parameters**
    - `pitch` - The pitch to add.
    - `model` - The matrix of probabilities to append to the model for that pitch.
  - **Returns** - The model matrix that was replaced, if any.

- *putAll*
  `public double` **putAll**`( java.util.Map` **m** `)`
  - **Usage**
    - Puts a Map of notes into the model.
  - **Parameters**
    - `m` - The map to add to the model
  - **Returns** - The last model replaced, if any.

- *validateModel*
  `public void` **validateModel**`( )`
  - **Usage**
    - Protected call to the internal _validateModel()
      This method locks the model if validation is successful.

Chapter 3

Package net.parallaxed.bluejam

Package Contents *Page*

Interfaces

Function 16
This interface defines the behaviours of a GP function.

Heuristic 17
A heuristic is any class that can provide a pattern template upon which a terminal set can evolve.

NoteSequence 19
NoteSequence is the interface describing operations that can be performed over a sequence of Notes.

Terminal 19
All terminals must have unique identifiers to satisfy uniqueness in a TerminalSet.

Classes

Accidental 21
Defines the possible accidentals.

Evolve 21
The evolve class carries out the main evolution cycles and outputs "winning" musical candidates by notifying listeners.

EvolveHeuristic 24
TODO Implement this.

HeuristicCollection 25
Provides a class to store population heuristics and select them using different modes/distributions.

HeuristicCollection.SELECTION_TYPE 28
This Enum provides defines the possible methods by which heuristics are selected from the collection

Individual 29
This class defines an individual.

IndividualParameters 29
If the individual desires different parameters from those defined in the population by default, it can override the population's decision by creating a class of IndividualParameters.

JamParamters 34
The ParameterCollection class is used by BlueJam to specify the properties of a cycle of evolution.

JamParamters.Config 37

Defines the typed configuration values in a ParameterCollection
Mutable 42
Utility Class defining which properties of a note can be changed.
Note 5
This class is currently under consideration for the "abstract" modifier.
NoteCollection 44
Simply contains a list of notes in their order of play.
NoteLeaf 48
Implements a unary note-sequence (only one note in the sequence) NoteLeaf instances extend the functionality of Note, such that it can be added and manipulated within a NoteTree.
NoteLeafSet 50
The note set of all possible pitch classes in the octave range supplied to the constructor.
NoteTree 54
Note trees are a data structure representing an unordered tree of NoteSequences, itself forming a note sequence.
Pitch 56
This enumeration represents the pitches and can be used to determine enharmonic equivalence between pitch classes.
Population 58
The population class holds a collection of individuals and evolves them with or without a set of heuristics.
PopulationParameters 59
Provides a moderately strongly typed parameter collection for the population properties and constants.
Rhythm 59
This Enum contains all the rhythm's BlueJam supports.
Scale 60
This Enum describes how to produce each scale using stepped jumps over pitch classes.
Scale.BLUES 60
Defines a blues scale with hexatonic pitch representation.
Scale.MAJOR 62
Defines a standard major scale.
Scale.MINOR 63
Defines a standard minor scale.
ScaledSet 64
Given a set of rules for the scale, ScaledSet can work out the set of notes that fit that scale, over a given number of octaves.
SequenceParameters 13
Currently, custom tree parameters aren't supported, so this default instantiation should cover the basics.
TerminalSet 13
Wraps an ArrayList to provide set-like functionality (one unique instance per set).
TreeParser 13
This class reads in information from passed tree files, which normally represent a serialized layout of a NoteSequence.

3.1 Interfaces

3.1.1 Interface **Function**

This interface defines the behaviours of a GP function.
Since this interface is only to facilitate a gateway to the NoteSequence provided by any implementing classes, there is only one function - getNoteSequence().
There may be multiple types of function (such as reverse, augment, swing etc), but the default implementation found here in the current classes is a vanilla "play()".

Declaration

public interface Function

Methods

- *getNoteSequence*
  `public NoteSequence` **`getNoteSequence( )`**
  - **Usage**
    - Since the default function is to "play" the contents of the tree, this function should return a reference to a playable note sequence.
      Provides a reference to the note sequence contained in the implementing class.
  - **Returns** - A reference to the NoteSequence

3.1.2 Interface **Heuristic**

A heuristic is any class that can provide a pattern template upon which a terminal set can evolve.
In BlueJam, the default implementation is the HeuristicTree.
The heuristic tree is

Declaration

public interface Heuristic **implements** Function, NoteSequence

Methods

- *setSequenceParameters*
  `public void` **`setSequenceParameters(`**
  `net.parallaxed.bluejam.SequenceParameters` **`sequenceParameters )`**
  - **Usage**

    * Since NoteSequences should by default have some reference to sequenceParamaters, and the Genome (i.e. NoteTree) by default accepts this only in the constructor, we should have some method for overriding the sequenceParameters after construction on the heuristic.

- *toString*
  `public String` **`toString( )`**
  - **Usage**
    * This defines a name for the heuristic, such that a particular instance can be pulled from a HeuristicCollection by name.
      Note this does not have to be unique globally, only within the collection if you wish to address the heuristic individually.
  - **Returns** - The name of this heuristic.

3.1.3 Interface **NoteSequence**

NoteSequence is the interface describing operations that can be performed over a sequence of Notes.
This class fully abstracts it's implementing classes to potentially support alternatives to tree-based GP/EA approaches.
The default implementation of NoteSequence is NoteTree/NoteLeaf, each having additional operations supporting a subset of GP operations.
NoteSequence provides the basic functionality for 1-point crossover and mutation, with replaceNotes().

Declaration

public interface NoteSequence **implements** java.lang.Cloneable

Methods

- *addNotes*
  `public boolean` **`addNotes(`** `net.parallaxed.bluejam.NoteSequence` **`notes )`**
- *clone*
  `public NoteSequence` **`clone( )`**
  - **Usage**
    * If supported, this should return a copy of this NoteSequence independent from the original. This should be used mainly when an implementation wants to produce children or copies of itself.
      Heuristics use this method to return copies of themselves which can then be mutated.
  - **Returns** - A copy of this NoteSeqeunce
  - **Exceptions**
    * `java.lang.CloneNotSupportedException` -
- *contains*
  `public boolean` **`contains(`** `net.parallaxed.bluejam.NoteSequence` **`n )`**

- **Parameters**
  - `n` - The NoteSequence to look for
- **Returns** - True if this NoteSequence contains n

- *getNotes*
`public Iterator` **`getNotes( )`**
  - **Usage**
    - This abstracted function returns an iterator capable of accessing each note in the NoteSequence in an ordered fashion. Since an the representation of a note tree may be in no particular order, implementation of this function forces an order to be imposed at some point.
      In short, this allows the NoteSequence to be read into a buffer for playback through one of the appropriate classes
      NoteSequences are normally collapsed into NoteCollections using this iterator, to fix the ordering before playback.
      __ADD SEE CLAUSES__
  - **Returns** - An iterator over all the notes in the NoteSequence

---

- *removeNotes*
`public void` **`removeNotes(`** `net.parallaxed.bluejam.NoteSequence` **`notes )`**
  - **Usage**
    - Removes a note from the NoteSequence.
      Removal of a note should (in a linear representation) shift all subsequent notes left. For removing a note without this behaviour, see restNote()
  - **Parameters**
    - `notes` - A reference to the note to be removed.

- *sequenceParameters*
`public SequenceParameters` **`sequenceParameters( )`**
  - **Usage**
    - Returns the parameters of this sequence. Normally only valid in function implementations, Terminals are not required to provide a link to their parameters.
  - **Returns** - A reference to the sequence parameters for this NoteSequence.

- *swapNotes*
`public boolean` **`swapNotes(`** `net.parallaxed.bluejam.NoteSequence` **`swapOut,`**
`net.parallaxed.bluejam.NoteSequence` **`swapIn )`**
  - **Usage**
    - Swaps notes in a sequence. The supplied NoteSequence is located, and replaced by the given NoteSequence.
      Depending on the implementation, this function can be designed to act as mutation, 1-point crossover, or both.
  - **Parameters**
    - `swapOut` - First argument to swap
    - `swapIn` - Second argument to swap

- *validateNotes*
  `public void` **validateNotes**`( )`
  - **Usage**
    - Called before locking the note to make sure all properties covered by the lockMask are set.
      i.e. if lockMask = MUTABLE_RHYTHM, all the rhythm related properties will be checked before setting this.mutable.
      This method ensures all notes in the sequence are ready for playback. Must be called before any attempt to read the noteValue or duration fields.

3.1.4 Interface **Terminal**

All terminals must have unique identifiers to satisfy uniqueness in a TerminalSet.

Declaration

public interface Terminal
implements java.lang.Cloneable

Methods

- *getValue*
  `public int` **getValue**`( )`
  - **Usage**
    - Returns this terminal's value.
  - **Returns** - A unique identifier assigned by the implementing object.

3.2 Classes

3.2.1 Class **Accidental**

Defines the possible accidentals.
KEY is a special accidental remarking that the accidental should be the same as that of the root pitch.
This class can be used to mark accidentals in relation to scale, but also in relation to a particular note. In null cases, the null accidental (NONE) is returned.

Declaration

public final class Accidental
extends java.lang.Enum

FIELDS

- public static final Accidental SHARP
  –
- public static final Accidental FLAT
  –
- public static final Accidental NATURAL
  –
- public static final Accidental KEY
  – The KEY accidental implies that if the note occurs inside a pitch class, the accidental should be the same as the accidental of the root pitch in that scale.
- public static final Accidental NONE
  – The NONE accidental is returned when the note is not inside a pitch class (on it's own). When used to evaluate the presence of an accidental in a scale, it behaves in a similar way to KEY, but in the case of a natural present on the note, NONE should always bias itself to remove the accidental on that NOTE.

METHODS

- *valueOf*
  `public static Accidental` **valueOf**`( java.lang.String` **name** `)`
- *values*
  `public static final Accidental` **values**`( )`

3.2.2 CLASS **Evolve**

The evolve class carries out the main evolution cycles and outputs "winning" musical candidates by notifying listeners. Instances of this object are meant to be run in their own thread.
As the generations are run, registered listeners will receive references to NoteSequences. Feedback can be given on these sequences by calling the feedback() method with a score and a reference back to the NoteSequence that achieved that score. In the case of positive feedback, the system can choose to make that candidate and other high-scoring candidates from the population "elite", which means they get passed in to subsequent generations.
This class implements the Player interface, a primitive events pattern.

DECLARATION

public class Evolve
extends java.lang.Object
implements java.lang.Runnable, net.parallaxed.bluejam.playback.Player

Fields

- public boolean running
  –

Constructors

- *Evolve*
  `public` **Evolve**`( net.parallaxed.bluejam.SequenceParameters` **sequenceParameters**`, int` **populationCount** `)`
  - **Usage**
    * Creates an instance of the Evolve class
  - **Parameters**
    * `sequenceParameters` - The sequenceParameters to use.
  - **See Also**
    * `net.parallaxed.bluejam.SequenceParameters` (in 3.1.1, page 13)

- *Evolve*
  `public` **Evolve**`( net.parallaxed.bluejam.SequenceParameters` **sequenceParameters**`, int` **populationCount**`,` `net.parallaxed.bluejam.HeuristicCollection` **heuristics** `)`
  - **Usage**
    * Creates an instance of the Evolve class.
  - **Parameters**
    * `sequenceParameters` - The sequenceParameters to use.
    * `heuristics` - The HeuristicCollection to use.
  - **See Also**
    * `net.parallaxed.bluejam.SequenceParameters` (in 3.1.1, page 13)
    * `net.parallaxed.bluejam.HeuristicCollection` (in 3.2.8, page 25)

Methods

- *addListener*
  `public void` **addListener**`( net.parallaxed.bluejam.playback.Listener` **listener** `)`
  - **Usage**
    * Adds a listener to this player.

- *feedback*
  `public void` **feedback**`( int` **feedback**`, net.parallaxed.bluejam.NoteSequence` **notes** `)`
  - **Usage**

    - Accepts a positive or negative feedback value (usually -1 or 0 or 1), and turns on elitism for that NoteSequence making it appear in subsequent generations.

- *generations*
  `public int` **generations**`( )`
  - **Usage**
    - The number of generations we're going to run.

- *generations*
  `public void` **generations**`( int` **numberOfGenerations** `)`
  - **Usage**
    - Sets the number of generations. Limit: 0 <x <100
  - **Parameters**
    - `numberOfGenerations` - The number of generations to complete

- *getPopulationParameters*
  `public PopulationParameters` **getPopulationParameters**`( )`

- *matingPoolSize*
  `public void` **matingPoolSize**`( int` **matingPoolSize** `)`
  - **Usage**
    - Sets the size of the mating pool. Limit: 1 <x <21
  - **Parameters**
    - `matingPoolSize` - The new size of the pool

- *run*
  `public void` **run**`( )`
  - **Usage**
    - Runs the main evolution thread.

- *setPopulationCount*
  `public void` **setPopulationCount**`( int` **memberCount** `)`

- *togglePause*
  `public void` **togglePause**`( )`
  - **Usage**
    - Pauses the evolution from outside this thread.

3.2.3 Class **EvolveHeuristic**

TODO Implement this.
This class should deal with outputting the evolved heuristics into a serial format.

Declaration

public class EvolveHeuristic **extends** java.lang.Object

Constructors

- *EvolveHeuristic*
  `public` **EvolveHeuristic**`( )`

3.2.4 Class HeuristicCollection

Provides a class to store population heuristics and select them using different modes/distributions.
Calling selectHeuristic() returns a heuristic from the collection.
This class uses the ECJ implementation of MersenneTwisterFast.

Declaration

public class HeuristicCollection
extends java.util.ArrayList

Serializable Fields

- private int _index
  –
- private MersenneTwisterFast _mt
  –
- public HeuristicCollection.SELECTION_TYPE MODE
  – Specifies which selection type to use in this collection.
- private HashMap _byName
  –

Fields

- public HeuristicCollection.SELECTION_TYPE MODE
  – Specifies which selection type to use in this collection.

Constructors

- *HeuristicCollection*
  `public` **HeuristicCollection**`( )`
  – **Usage**
    * Initialises a HeuristicCollection (trivial)

Methods

- *add*
  `public boolean` **add**`( net.parallaxed.bluejam.Heuristic` **heuristic** `)`
- *add*
  `public boolean` **add**`( net.parallaxed.bluejam.Heuristic` **heuristic**`,`
  `java.lang.String` **name** `)`
- *getHeuristic*
  `public Heuristic` **getHeuristic**`( java.lang.String` **name** `)`
  - **Usage**
    - Returns a heuristic by name if it is present in the collection.
  - **Parameters**
    - `name` - The name of the desired heuristic
  - **Returns** - A reference to the named heuristic, or null.
- *loadHeuristics*
  `public static HeuristicCollection` **loadHeuristics**`( java.lang.String` **path** `)`
- *selectHeuristic*
  `public Heuristic` **selectHeuristic**`( )`
  - **Usage**
    - Returns a heuristic in line with the method selected by MODE.
  - **Returns** - A heuristic from the collection
  - **See Also**
    - `net.parallaxed.bluejam.HeuristicCollection.SELECTION_TYPE` (in 3.2.9, page 28)

3.2.5 Class HeuristicCollection.SELECTION_TYPE

This Enum provides defines the possible methods by which heuristics are selected from the collection

Declaration

public static final class HeuristicCollection.SELECTION_TYPE
extends java.lang.Enum

Fields

- public static final HeuristicCollection.SELECTION_TYPE EVEN
  - An even distribution will iterate over the contents of the collection, then return to normal, so each heuristic is selected evenly.
- public static final HeuristicCollection.SELECTION_TYPE RANDOM
  - A random selection uses an instance of MersenneTwisterFast to pick out a heuristic.

METHODS

- *valueOf*
  `public static HeuristicCollection.SELECTION_TYPE` **valueOf**`( java.lang.String` **name** `)`
- *values*
  `public static final HeuristicCollection.SELECTION_TYPE` **values**`( )`

3.2.6 CLASS **Individual**

This class defines an individual. Also could be termed a chromosome in GP, but our individuals represent instances of NoteTrees based around a given Heuristic present in the population.
An individual implements Function in the sense that it has access to the "play()" function in the NoteSequence represented by the individual. Theoretically, this permits individuals and their sequences to be chained together.

DECLARATION

public class Individual
extends java.lang.Object
implements Function

CONSTRUCTORS

- *Individual*
  `public` **Individual**`( net.parallaxed.bluejam.NoteSequence` **notes**`, net.parallaxed.bluejam.Heuristic` **heuristic** `)`
  - **Usage**
    - For creating an individual outside a population in crossover
  - **Parameters**
    - `notes` - The NoteSequence to initialize the Individual with
    - `heuristic` - The heuristic used to build this individual.
- *Individual*
  `public` **Individual**`( net.parallaxed.bluejam.NoteSequence` **notes**`, net.parallaxed.bluejam.PopulationParameters` **popParams** `)`
  - **Usage**
    - For creating an individual outside a population (mostly for testing)
  - **Parameters**
    - `notes` - The NoteSequence to initialize the Individual with
    - `popParams` - The parameters the individual should take.

- *Individual*
  `public` **Individual**`( net.parallaxed.bluejam.NoteSequence` **notes**`,`
  `net.parallaxed.bluejam.PopulationParameters` **popParams**`,`
  `net.parallaxed.bluejam.Heuristic` **heuristic** `)`
  - **Usage**
    - For creating an individual outside a population (mostly for testing) with a heuristic.
  - **Parameters**
    - `notes` - The NoteSequence to initialize the Individual with
    - `popParams` - The parameters the individual should take.
    - `heuristic` - The heuristic to use for initialize()

- *Individual*
  `public` **Individual**`( net.parallaxed.bluejam.Population` **population** `)`
  - **Usage**
    - Creates an individual with no given heuristic.
  - **Parameters**
    - `population` - The population that this individual belongs to.

- *Individual*
  `public` **Individual**`( net.parallaxed.bluejam.Population` **population**`,`
  `net.parallaxed.bluejam.Heuristic` **heuristic** `)`
  - **Usage**
    - Creates this individual with a given Heuristic.
      The heuristic passed will be used to create the base note sequence for this individual.
  - **Parameters**
    - `population` - The population that this individual belongs to.
    - `heuristic` - A heuristic for evolving this individual's NoteSequence.

- *Individual*
  `public` **Individual**`( net.parallaxed.bluejam.PopulationParameters` **popParams**`,`
  `net.parallaxed.bluejam.Heuristic` **heuristic** `)`
  - **Usage**
    - For creating an individual outside a population (mostly for testing) with a heuristic.
  - **Parameters**
    - `popParams` - The parameters the individual should take.
    - `heuristic` - The heuristic to use for initialize()

Methods

- *evaluate*
  `public double` **evaluate**`( )`
  - **Usage**

    * Gives the fitness of this individual or throws an exception if the individual is not evaluated yet.
  - **Returns** - The fitness of the individual

- *getHeuristic*
  `public Heuristic` **getHeuristic**`( )`
  - **Usage**
    * Returns a reference to the Heuristic used to create the individual, or null if no Heuristic was used.
  - **Returns** - This individual's Heuristic, or null.

- *getNoteSequence*
  `public NoteSequence` **getNoteSequence**`( )`
  - **Usage**
    * {@inheritDoc}

- *getParameters*
  `public PopulationParameters` **getParameters**`( )`
  - **Returns** - This population's parameter object.

- *initialize*
  `public void` **initialize**`( )`
  - **Usage**
    * Initialises the individual by parsing the heuristic into a note tree. For optimisation reasons this might not always occur at instantiation.

- *invalidate*
  `public void` **invalidate**`( )`
  - **Usage**
    * Sets the internal state of this individual such that the next call to evaluate() will re-evaluate the fitness of the Individual.

- *population*
  `public void` **population**`( net.parallaxed.bluejam.Population` **population** `)`
  - **Usage**
    * Reassigns this individual's population membership.
  - **Parameters**
    * `population` - The population to place this individual in.

- *setParameter*
  `public void` **setParameter**`( java.lang.String` **name**`, java.lang.Object` **value** `)`
  - **Usage**
    * Will create and IndividualParameters instance for this object if it doesn't already have one, and set the given individual parameter.

- **Parameters**
  - `name` - The parameter name.
  - `value` - The parameter value.
- **See Also**
  - `net.parallaxed.bluejam.IndividualParameters` (in 3.2.11, page 29)

3.2.7 Class **IndividualParameters**

If the individual desires different parameters from those defined in the population by default, it can override the population's decision by creating a class of IndividualParameters.
This only works for some parameters.

Declaration

public class IndividualParameters
extends net.parallaxed.bluejam.PopulationParameters

Constructors

- *IndividualParameters*
  `public` **IndividualParameters**`( net.parallaxed.bluejam.PopulationParameters` **params** `)`
  - **Usage**
    - If and individual can't resolve a parameter, it will retrieve it from the parent.
  - **Parameters**
    - `params` -

Methods

- *_checkType*
  `protected boolean` **_checkType**`( java.lang.String` **name**`, java.lang.Object` **value** `)`
  - **Usage**
    - Ensures that only parameters specific to individuals can be overridden.

- *getParameter*
  `public Object` **getParameter**`( java.lang.String` **name** `)`
  - **Usage**
    - If the value is not found in the individual parameters, will return them from the parent.

3.2.8 Class **JamParamters**

The ParameterCollection class is used by BlueJam to specify the properties of a cycle of evolution. Each ParameterCollection is passed down to a population, which uses it to configure it's individuals. Parameter collections are immutable, since the individuals need access to read the values, but are prohibited from changing them.
Separate configurations may evolve in different populations simultaneously to produce different solos.

Declaration

public class JamParamters
extends java.lang.Object

Constructors

- *JamParamters*
  `public` **JamParamters**`( )`
  - **Usage**
    - Initialises a parameter collection with defaults.

- *JamParamters*
  `public` **JamParamters**`( net.parallaxed.bluejam.Pitch` **pitch**`, net.parallaxed.bluejam.Scale` **scale**`, java.lang.Integer` **tempo** `)`
  - **Usage**
    - Initializes a Jam with the given parameterss
  - **Parameters**
    - `pitch` - The root pitch (one of the Pitch enum)
    - `scale` - The scale to use (one of the Scale enum)
    - `tempo` - An integer representing the beats per minute (BPM)

Methods

- *getParameter*
  `public Object` **getParameter**`( net.parallaxed.bluejam.JamParamters.Config` **c** `)`
  - **Usage**
    - This method retrieves a typed parameter from the collection.
  - **Parameters**
    - `c` - The configuration key to return
  - **Returns** - An untyped object that can be casted to an expected type
  - **Exceptions**
    - `java.lang.Exception` - If the parameter does not exist in this ParameterCollection

- *getParameter*
  `public String` **getParameter**`( java.lang.String  s )`
  - **Usage**
    - Retrieves a parameter stored in stringParams, or throws an exception if the parameter is not found.
  - **Parameters**
    - `s` - The parameter name to get
  - **Returns** - The value of parameter s
  - **Exceptions**
    - `java.lang.Exception` -

- *getScaledSet*
  `public ScaledSet` **getScaledSet**`( )`
  - **Usage**
    - Terminals for the current Jam can be pulled from this ScaledSet.
  - **Returns** - A reference to this JamParameters ScaledSet.

- *maxOctave*
  `public int` **maxOctave**`( )`

- *minOctave*
  `public int` **minOctave**`( )`

- *readConfig*
  `public void` **readConfig**`( java.io.File  file )`
  - **Usage**
    - Reads in the supplied file and configures the ParameterCollection
  - **Parameters**
    - `file` - A java.io.File object to read.

- *readConfig*
  `public void` **readConfig**`( java.lang.String  path )`
  - **Usage**
    - Reads in a config file at the specified (absolute) path.
  - **Parameters**
    - `path` - The path to the config file.

- *rootPitch*
  `public Pitch` **rootPitch**`( )`
  - **Usage**
    - Returns the root pitch of this configuration as a type Pitch.
  - **Returns** - The root Pitch.

- *rootPitch*
  `public void` **rootPitch**`( net.parallaxed.bluejam.Pitch  rootPitch )`
  - **Usage**

    - Sets the root pitch of this Jam
  - **Parameters**
    - `rootPitch` - The rootPitch of the Jam (cannot be Pitch.R)

- *scale*
  `public Scale` **scale**`( )`
  - **Returns** - The scale being used in this Jam

- *scale*
  `public void` **scale**`( net.parallaxed.bluejam.Scale` **scale** `)`
  - **Usage**
    - Sets the scale of this Jam
  - **Parameters**
    - `scale` - A reference to the singleton Scale instance.

- *setParameter*
  `public void` **setParameter**`( java.lang.String` **s**`, java.lang.String` **value** `)`

- *setRange*
  `public void` **setRange**`( int` **minOctave**`, int` **maxOctave** `)`
  - **Usage**
    - Sets the range of the Jam, in octaves.
      Both values must be -1 <x <10.
      Also, (minOctave <= maxOctave) must be true, otherwise no action will be taken.

- *tempo*
  `public void` **tempo**`( int` **tempo** `)`
  - **Usage**
    - Sets the tempo of this Jam
  - **Parameters**
    - `tempo` - A BPM value between 1-240

3.2.9 Class **JamParamters.Config**

Defines the typed configuration values in a ParameterCollection

Declaration

public static final class JamParamters.Config **extends** java.lang.Enum

FIELDS

- public static final JamParamters.Config ROOT_PITCH
  - (Pitch) The root pitch around which to evolve. Default = A440
- public static final JamParamters.Config TEMPO
  - (Integer) The tempo (in beats per minute - BPM). Default = 120bpm
- public static final JamParamters.Config SCALE
  - (Scale) One of the supported "Scales" enumeration values. Default = Blues.

METHODS

- *toString*
  `public String` **toString**`( )`
  - **Usage**
    - The string value of each enum.
- *valueOf*
  `public static JamParamters.Config` **valueOf**`( java.lang.String` **name** `)`
- *values*
  `public static final JamParamters.Config` **values**`( )`

3.2.10 CLASS Mutable

Utility Class defining which properties of a note can be changed. Assigns bitmasks to deal with permissions under NoteLeaf.

DECLARATION

public final class Mutable
extends java.lang.Object

FIELDS

- public static final byte NONE
  - Indicates none of the note properties can be changed.
- public static final byte RHYTHM
  - This note can have it's duration increased/decreased (in the note tree implementation, this note can change it's level in the tree).
- public static final byte PITCH

  - The pitchClass/noteValue can change. The note can also toggleRest()

- public static final byte ALL
  - Both pitch and rhythm can change (sum of other flag values).

CONSTRUCTORS

- *Mutable*
  `public` **Mutable**`( )`

3.2.11 CLASS Note

This class is currently under consideration for the "abstract" modifier.
This class fully abstracts the properties of a note from the possible implementations of notes in the evolution framework.
Since BlueJam is entirely tree based, the default extension of this class is NoteLeaf. Notes can belong to more than one collection. A NoteLeaf can also belong to more that one collection, but only one NoteTree.
Once stored in a structure, the only safe way to manipulate the note is through the methods defined on the type specific to that data structure -i.e. it's not recommended to typecast to Note in order to bypass things like locking protection.
All subclasses of Note involved in a GP algorithm should also implement Terminal.
TODO Event-driven pattern for Note validation to inform children.

DECLARATION

public class Note **extends** java.lang.Object

FIELDS

- public static final long serialVersionUID
  - This requires post-processing by an output class.

CONSTRUCTORS

- *Note*
  `public` **Note**`( )`
  - **Usage**
    * Trivial Constructor for later application of properties.

- *Note*
  `public` **Note**`( float` **noteValue** `)`

- *Note*
  `public` **Note**`( net.parallaxed.bluejam.Pitch` **pitchClass**`, int` **octave** `)`

  - **Usage**
    * Constructs a note with the given pitch class and octave
  - **Parameters**
    * `pitchClass` -
    * `octave` -

METHODS

- *duration*
  `public double` **duration**`( )`
  - **Usage**
    * Retrieves the duration of the note.
      Logs an exception if the note was not validated prior to playback.
  - **Returns** - The length of this note in milliseconds.

- *duration*
  `protected void` **duration**`( double` **milliseconds** `)`
  - **Usage**
    * Sets the duration of the note. This value is sent out along the MIDI line to tell the ouput device how long to hold the note for.
      Can only be called by it's subclasses that deal with how long the note should be played for.
  - **Parameters**
    * `milliseconds` - The new duration in milliseconds.

- *evaluatedPitch*
  `public Pitch` **evaluatedPitch**`( )`
  - **Returns** - A value for the calculated pitch.

- *evaluatePitch*
  `protected boolean` **evaluatePitch**`( net.parallaxed.bluejam.JamParamters` **params** `)`
  - **Usage**
    * Evaluates this Note in the given JamParameters context.
      JamParameters supplies the root pitch. The actual pitch of this note is supplied by 0 <pitchRelative <11, which is added as an offset to the root pitch to give the real pitch.
  - **Parameters**
    * `params` - The JamParameters context - passed from somewhere on high.
  - **Returns** - True on success, false on failure.

- *evaluatePitch*
  `public boolean` **evaluatePitch**`( net.parallaxed.bluejam.Pitch` **rootPitch** `)`
  - **Usage**

        * A note can be defined as "relative", in the context of a given pitch and scale. If a note is relative, it will remain invalid until this method is called with some notion of context (i.e. a root pitch and a scale).
        This method will resolve the relative position of this note into an absolute pitch value.
    - **Parameters**
        * `rootPitch` - The root pitch to evaluate against

- *evaluateRhythm*
`public boolean` **evaluateRhythm**`( net.parallaxed.bluejam.SequenceParameters` **params** `)`
    - **Usage**
        * Evaluates the rhythm of this note given the passed SequenceParameters. These need to be passed in from an implementation of Note.
        Error's will be sent to the default ErrorFeedback.
    - **Parameters**
        * `params` - The given SequenceParameters ("context")
    - **Returns** - Whether the evaluation succeeded or not.

- *invalidate*
`protected void` **invalidate**`( )`
    - **Usage**
        * Marks this note as invalid.

- *noteValue*
`public float` **noteValue**`( )`
    - **Usage**
        * The MIDI note number of this note.
    - **Returns** - A float between 0 and 127

- *octave*
`public int` **octave**`( )`
    - **Usage**
        * Returns the octave number.
    - **Returns** - The octave the note is in

- *octave*
`public void` **octave**`( int` **octave** `)`
    - **Usage**
        * Sets the octave of this note if it can be arbitrarily changed.
        Subclasses should override this to account for locks if necessary.
    - **Parameters**
        * `octave` - The new octave number

- *pitchClass*
`public Pitch` **pitchClass**`( )`

- **Returns** - The Pitch class of this note.

- *pitchClass*
`public void` **pitchClass**`( net.parallaxed.bluejam.Pitch` **p** `)`
  - **Usage**
    - Set the pitchClass of this note.
      Subclasses should override this fully to account for mutability of note properties.
  - **Parameters**
    - `p` - The desired pitchClass

- *pitchRelative*
`public int` **pitchRelative**`( )`
  - **Usage**
    - Used when pitch = Pitch.R to calculate the absolute pitch of the note.
  - **Returns** - The number of steps to this note from the root

---

- *pitchRelative*
`public void` **pitchRelative**`( int` **offset** `)`
  - **Usage**
    - Sets the pitch relative to the given root note + octave.
      Used when evaluating heuristic trees.
      If this number is negative, the root note is above the note If this number is positive, the root note is below the note
  - **Parameters**
    - `offset` - The number of steps (semitones) between the root note and this note.

- *rest*
`public boolean` **rest**`( )`

- *rhythm*
`public Rhythm` **rhythm**`( )`
  - **Usage**
    - Only accessed when initially adding this NoteLeaf to a NoteTree.
  - **Returns** - The rhythm of the note

- *rhythm*
`public void` **rhythm**`( net.parallaxed.bluejam.Rhythm` **r** `)`
  - **Usage**
    - Only accessed when initially adding this NoteLeaf to a NoteTree.
      This method should be overridden (not just hidden) by classes that use locking.

- *swingNote*
`public void` **swingNote**`( int` **swingPercent**`, net.parallaxed.bluejam.Note` **swingPartner** `)`
  - **Usage**

    * Adds swing to a given note.
      A swing of 100% is equivalent to a tied note.
      Swing is the act of pairing off two notes and then assigning a ratio to control their rhythmic value.
      This function will automatically assign the right swing value to the partner note.
  - **Parameters**
    * `swingPercent` - The amount of swing to assign to this note
    * `swingPartner` - The swing partner of this note

- *swingPartner*
  `public Note` **swingPartner**`( )`
  - **Returns** - The note partnered with this one.

- *swingPercent*
  `public int` **swingPercent**`( )`
  - **Returns** - The percentage swing on this note (if any).

- *toggleRest*
  `public void` **toggleRest**`( )`
  - **Usage**
    * Toggles rest on/off.

- *toggleRest*
  `public void` **toggleRest**`( boolean  rest )`
  - **Usage**
    * Sets this note to be a rest given the passed boolean.
      No information about the note is lost by turning it into a rest note. This can be undone at a later stage by calling toggleRest() again.
  - **Parameters**
    * `rest` - True turns this into a rest note, false reverts.

- *toString*
  `public String` **toString**`( )`
  - **Usage**
    * Returns true if this note is a rest

- *validateNotes*
  `public void` **validateNotes**`( )`
  - **Usage**
    * This method validates the note and throws the right kind of exception if anything is awry.
      For some reason this method uses lazy evaluation to optimize screen real-estate.
  - **Exceptions**
    * `net.parallaxed.bluejam.exceptions.ValidationException` -

- *validatePitch*
  `protected boolean` **validatePitch**`( )`
  - **Usage**
    - Checks that the pitchClass field has a value and that the value is not relative.
  - **Returns** - True if the criteria are met, false otherwise.

- *validateRhythm*
  `protected boolean` **validateRhythm**`( )`
  - **Usage**
    - Checks the duration of this note (in milliseconds) is >0.
  - **Returns** - True if the criteria are met, false otherwise.

3.2.12 Class **NoteCollection**

Simply contains a list of notes in their order of play.
Note that this class is recursively typed, even though it wraps a collection of <Note>, those elements added to it MUST also implement NoteSequence.
This class acts as a buffer between the playback classes and the evolution classes (i.e. note trees etc).

Declaration

public class NoteCollection
extends java.util.ArrayList
implements NoteSequence

Serializable Fields

- private HashMap _idMap
  –
- private SequenceParameters _sp
  –

Constructors

- *NoteCollection*
  `public` **NoteCollection**`( )`
  - **Usage**
    - Constructs a vanilla NoteCollection

- *NoteCollection*
  `public` **NoteCollection**`( net.parallaxed.bluejam.SequenceParameters` **sequenceParameters** `)`

  - **Usage**
    * Constructs a NoteCollection with the passed sequenceParameters
      This is useful for subclasses that require context in their Collection.
  - **Parameters**
    * `sequenceParameters` -

Methods

- *add*
  `public boolean` **add**`( net.parallaxed.bluejam.Note` **n** `)`

  - **Usage**
    * Will check if a passed note has an ID, if so it will also be added to an internal HashMap for later retrieval.

- *addNotes*
  `public boolean` **addNotes**`( net.parallaxed.bluejam.NoteSequence` **notes** `)`

  - **Usage**
    * Appends the passed NoteSeqeunce to the NoteCollection.
  - **Parameters**
    * `notes` - The notes to be appended.
  - **Returns** - Always true for this implementation.

- *clone*
  `public NoteCollection` **clone**`( )`

  - **Usage**
    * NoteCollections are note cloneable (...yet)

- *contains*
  `public boolean` **contains**`( net.parallaxed.bluejam.NoteSequence` **n** `)`

  - **Parameters**
    * `n` - The NoteSequence to search for
  - **Returns** - True if this collection contains the passed NoteSequence.

- *crop*
  `public NoteCollection` **crop**`( int` **index** `)`

  - **Usage**
    * Crops a noteCollection returning all elements from the specified index to the end of the collection.
  - **Parameters**
    * `index` - The note index where the crop begins
  - **Returns** - A cropped copy of this NoteCollection

- *crop*
  `public NoteCollection` **crop**`( int` **index**`, int` **end** `)`

- **Usage**
  - Crops a noteCollection returning all elements from the specified index to the point specified.
- **Parameters**
  - `index` - The note index where the crop begins
  - `end` - Where to stop the crop.
- **Returns** - A cropped copy of this NoteCollection

- *get*
  `public Note` **get**`( int` **index** `)`

- *getIndex*
  `public Note` **getIndex**`( int` **index** `)`
  - **Usage**
    - To preserve original functionality, this simply calls the superclass get()
  - **Parameters**
    - `index` - The index to retrieve.
  - **Returns** - The note at index.

- *getNotes*
  `public Iterator` **getNotes**`( )`
  - **Usage**
    - {@inheritDoc}
  - **Returns** - An iterator over this NoteCollection

- *removeNotes*
  `public void` **removeNotes**`( net.parallaxed.bluejam.NoteSequence` **notes** `)`
  - **Usage**
    - {@inheritDoc}
  - **See Also**
    - `net.parallaxed.bluejam.NoteSequence` (in 3.2.3, page 19)

- *sequenceParameters*
  `public SequenceParameters` **sequenceParameters**`( )`
  - **Returns** - A reference to this NoteCollection's SequenceParameters.
  - **See Also**
    - `net.parallaxed.bluejam.SequenceParameters` (in 3.1.1, page 13)

- *swapNotes*
  `public boolean` **swapNotes**`( net.parallaxed.bluejam.NoteSequence` **swapOut**`, net.parallaxed.bluejam.NoteSequence` **swapIn** `)`
  - **Usage**
    - {@inheritDoc}
  - **See Also**
    - `net.parallaxed.bluejam.NoteSequence` (in 3.2.3, page 19)

- *validateNotes*
  `public void` **validateNotes**`( )`
  - **Usage**
    - Attempts to call validateNotes() on all it's children.

3.2.13 Class NoteLeaf

Implements a unary note-sequence (only one note in the sequence)
NoteLeaf instances extend the functionality of Note, such that it can be added and manipulated within a NoteTree. It is part of the default BlueJam implementation.
Instances of this class can be manipulated in the same way as a note sequences, to add more notes (branch off) or swap itself.
This class also implements Terminal, inferring that it can be part of a terminal set, and that when added to a NoteTree, the leaf is only manipulable within the bounds given by it's lockMask.
Note that a Terminal on a NoteTree can only be condensed by having it's parent swapped out, it cannot be removed, only cleared (or "rested").
Consider NoteLeaf interning by using a HashMap over the heap.

Declaration

public class NoteLeaf
extends net.parallaxed.bluejam.Note
implements NoteSequence, java.lang.Cloneable, Terminal,
net.parallaxed.bluejam.playback.Saveable

Fields

- public int id
  - This public identifier is only used when building note trees to describe notes that may be paired into swing partners, or have trails between them.
    Can also be used in NoteCollections to identify notes.
- public NoteTree _parent
  - This should be set by the add() methods in NoteTree.

Constructors

- *NoteLeaf*
  `public` **NoteLeaf**`( float` **noteValue**`, int` **duration** `)`
  - **Usage**
    - Shouldn't be used in this implementation - this is for reconstructing playback melodies only.
  - **Parameters**
    - `noteValue` - The floatValue (MIDI number) of this note.

- *NoteLeaf*
  `public` **NoteLeaf**`( net.parallaxed.bluejam.Pitch` **pitchClass** `)`
  - **Usage**
    - For initialization - no parent or octave is needed yet (these are contextually added).
  - **Parameters**
    - `pitchClass` - The pitchClass of this note

- *NoteLeaf*
  `public` **NoteLeaf**`( net.parallaxed.bluejam.Pitch` **pitchClass**`, int` **octave** `)`
  - **Usage**
    - For construction in a set - no parent is needed.
  - **Parameters**
    - `pitchClass` - The pitchClass of this note
    - `octave` - The octave in which this note occurs.

- *NoteLeaf*
  `public` **NoteLeaf**`( net.parallaxed.bluejam.Pitch` **pitchClass**`,`
  `net.parallaxed.bluejam.Rhythm` **rhythm**`, int` **octave** `)`
  - **Usage**
    - For construction at runtime.
  - **Parameters**
    - `pitchClass` - The pitchClass of this note
    - `rhythm` - The rhythm of this note
    - `octave` - The octave in which this note occurs

- *NoteLeaf*
  `public` **NoteLeaf**`( net.parallaxed.bluejam.Rhythm` **rhythm** `)`
  - **Usage**
    - Constructs a NoteLeaf initialized with blank values (for building trees from files).

Methods

- *addNotes*
  `public boolean` **addNotes**`( net.parallaxed.bluejam.NoteSequence` **notes** `)`
  - **Usage**
    - Transforms this NoteLeaf into a NoteTree.
      Supports only the addition of NoteLeafs, therefore NoteSequences of size 1 only.
      The note currently on this leaf will split and accept one note in the given NoteSequence as it's child, if the rhythm for that NoteSequence is valid at this depth.
  - **Returns** - True on success, false on failure.

- *clone*
  `public NoteLeaf` **clone**`( )`
  - **Usage**

        * {@inheritDoc} NB The proper way to do this would involve a call to super.clone()

- *contains*
`public boolean` **contains**`( net.parallaxed.bluejam.NoteSequence` **n** `)`
  - **Usage**
    * {@inheritDoc}

- *getNote*
`public Note` **getNote**`( )`

- *getNotes*
`public Iterator` **getNotes**`( )`
  - **Usage**
    * Returns an iterator that simply wraps this note to keep in line with the Iterator pattern.
      Somewhat expensive, possibly a candidate for optimization.

- *getValue*
`public int` **getValue**`( )`
  - **Usage**
    * Returns a unique signed integer for this note (the MIDI note number), or -1 if this note is relative.
      This method is used for building TerminalSets

- *isRelative*
`public boolean` **isRelative**`( )`
  - **Returns** - True if this NoteLeaf bears relative pitch.

- *lockAll*
`protected void` **lockAll**`( )`
  - **Usage**
    * Attempts to lock all note properties and make the note immutable - if this fails, an exception thrown explaining which properties are invalid.
      This is arbitrary, no validation takes place here.
      Subclasses must override these methods to provide correct validation.
  - **Exceptions**
    * `net.parallaxed.bluejam.exceptions.ValidationException` - if the note cannot be locked

- *lockMask*
`public void` **lockMask**`( int` **lockMask** `)`
  - **Usage**
    * Will perform the appropriate lock operations given the passed mask.
      If the operation cannot be performed due to validation inconsistencies, this method will swallow the exception and forward it to an active display.
  - **Parameters**

* `lockMask` - A non-negative integer

- *lockMask*
`public void` **lockMask**`( int` **lockMask**`, boolean` **force** `)`
  - **Usage**
    * TreeParser needs to be able to force a lock mash when building trees from files.
  - **Parameters**
    * `lockMask` - A non-negative integer
    * `force` - Whether to force the lock mask application or not
  - **Exceptions**
    * `net.parallaxed.bluejam.exceptions.ValidationException` -

- *lockPitch*
`protected void` **lockPitch**`( )`
  - **Usage**
    * This is arbitrary, no validation takes place here.
      Subclasses must override these methods to provide correct validation.

- *lockRhythm*
`protected void` **lockRhythm**`( )`
  - **Usage**
    * This is arbitrary, no validation takes place here.
      Subclasses must override these methods to provide correct validation.

- *mutable*
`public byte` **mutable**`( )`
  - **Usage**
    * Returns a bitmask indicating which parts of the note are mutable.
  - **Returns** - a masked value from (0-3)

- *octave*
`public void` **octave**`( int` **octave** `)`
  - **Usage**
    * {@inheritDoc}
  - **See Also**
    * `net.parallaxed.bluejam.Note` (in 1.2.1, page 5)

- *pitchClass*
`public void` **pitchClass**`( net.parallaxed.bluejam.Pitch` **p** `)`
  - **Usage**
    * {@inheritDoc}
  - **See Also**
    * `net.parallaxed.bluejam.Note` (in 1.2.1, page 5)

- *removeNotes*
`public void` **removeNotes**`( net.parallaxed.bluejam.NoteSequence` **notes** `)`

- **Usage**
  - This method simply sets the parent to null, if the passed NoteSequence == this. NB: the parent should erase this child (leaving a null gap in the children array). The function call passed to this node should simply be a signal to get ready and GC the note.
- **Parameters**
  - `notes` - The NoteSequence to remove

- *rhythm*
`public void` **rhythm**`( net.parallaxed.bluejam.Rhythm` **r** `)`
  - **Usage**
    - To honour locking.

- *sequenceParameters*
`public SequenceParameters` **sequenceParameters**`( )`
  - **Usage**
    - {@inheritDoc}

- *stringify*
`public String` **stringify**`( )`
  - **Usage**
    - Returns a stringified version of this NoteLeaf {@inheritDoc}

- *swapNotes*
`public boolean` **swapNotes**`( net.parallaxed.bluejam.NoteSequence` **swapOut**`, net.parallaxed.bluejam.NoteSequence` **swapIn** `)`
  - **Usage**
    - NoteLeaves do not support swapping - this should be done by NoteTree's (i.e. the _parent)
  - **See Also**
    - `net.parallaxed.bluejam.NoteTree` (in 3.2.18, page 54)

- *swingNote*
`public void` **swingNote**`( int` **swingPercent**`, int` **swingPartnerId** `)`
  - **Usage**
    - Used when building trees to assign a swing partner to this NoteLeaf at a later time. swingPartnerId will be used to assign this.swingPartner
  - **Parameters**
    - `swingPercent` - The amount to swing this note
    - `swingPartnerId` - The ID of the paired note

- *swingPartnerId*
`public int` **swingPartnerId**`( )`
  - **Returns** - The ID of the swingPartner if this note leaf has one

---

- *validateNotes*
  `public void` **validateNotes**`( )`
  - **Usage**
    - Validates this NoteSequence. {@inheritDoc}

- *validatePitch*
  `protected boolean` **validatePitch**`( )`
  - **Usage**
    - Validates that this NoteLeaf is ready to play inside it's current context (assuming it has a parent), or is configured to play standalone.

- *validateRhythm*
  `protected boolean` **validateRhythm**`( )`
  - **Usage**
    - {@inheritDoc}

3.2.14 Class **NoteLeafSet**

The note set of all possible pitch classes in the octave range supplied to the constructor.
TODO Unit test for this...

Declaration

public class NoteLeafSet
extends net.parallaxed.bluejam.TerminalSet

Constructors

- *NoteLeafSet*
  `public` **NoteLeafSet**`( )`
  - **Usage**
    - Trivial constructor does nothing if we're using one of the subclasses.

- *NoteLeafSet*
  `public` **NoteLeafSet**`( int` **minOctave**`, int` **maxOctave** `)`
  - **Usage**
    - Constructs a NoteLeafSet constrained by octave range.
      Each range starts on A and ends on Ab, so a range of 4-5 yields all notes between A4-Ab5.
  - **Parameters**
    - `minOctave` -
    - `maxOctave` -

METHODS

- *add*
  `public boolean` **add**`( net.parallaxed.bluejam.Terminal` **t** `)`
  - **Usage**
    - Overrides the superclass' add function to ensure only NoteLeaf instances are added to this set.

- *getRandom*
  `public NoteLeaf` **getRandom**`( net.parallaxed.bluejam.Note` **n** `)`

- *getRandom*
  `public NoteLeaf` **getRandom**`( net.parallaxed.bluejam.NoteSequence` **n** `)`
  - **Usage**
    - Returns a random terminal that is likely to fit with the current model, knowing that the last terminal is of a given pitch.
      Note that this only takes into consideration the very last note of the note sequence.
      If no model has been set, simply returns a terminal from vanilla getRandom().
  - **Parameters**
    - `n` - The NoteSequence preceding the current note.

- *setModel*
  `public void` **setModel**`( net.parallaxed.bluejam.grammar.PitchModel` **model** `)`
  - **Usage**
    - Sets the model that this NoteLeaf set should use.
  - **Parameters**
    - `model` -

- *setRange*
  `public void` **setRange**`( int` **minOctave**`, int` **maxOctave** `)`
  - **Usage**
    - Sets the maximum range of the notes being played.
      maxOctave must be >= minOctave
  - **Parameters**
    - `minOctave` -
    - `maxOctave` -

3.2.15 CLASS NoteTree

Note trees are a data structure representing an unordered tree of NoteSequences, itself forming a note sequence.
Trees are walked and flattened into note sequences, representing a series of musical notes that make up a phrase.
Depending on the player, the tree may be walked in different ways, so the order of the notes in the tree is fixed, but that fixed structure may be interpreted in all directions.
Trees are a function, and by most definitions that function is isomorphic to "playing" the tree. When

If the parent of a note tree is null, then the current node is determined to be the root of the tree. No other node in the tree should have a null parent.
There are several operations that can happen to a NoteTree, remove(note), add(note) and swap(notesOut,notesIn). Add and remove are self explanatory.
Notes cannot be added in arbitrary positions yet, all added notes are appended to the node that executes the add operation.
NoteTree instances may also be Heuristics, effectively providing the "teaching" basics for the program, guiding the solo in a certain direction. Evolution cycles may occur with or without heuristic trees.
Node arity is not limited, each add will create a new branch and the arity will increase.
The arity of the note tree determines the rhythm of the notes underneath it, so the position of node in a tree defines the duration of that node (not including alterations for properties such as swing).
As a result of this, "rhythm" is not a property of notes, it is prescribed by the structure of the note sequence and evaluated at play time into the duration (long millisecs) property.
TODO Change locking mechanism so the Heuristic locks are preserved.

Declaration

public class NoteTree
extends java.lang.Object
implements NoteSequence, Heuristic, Function, java.lang.Cloneable

Fields

- public static final int MAX_DEPTH
  - Rhythms below HEMIQUAVER are not supported, so the MAX_DEPTH for any NoteTree is 8.

Constructors

- *NoteTree*
  `public` **NoteTree**`( )`
- *NoteTree*
  `public` **NoteTree**`( net.parallaxed.bluejam.NoteSequence` **parent** `)`
- *NoteTree*
  `public` **NoteTree**`( net.parallaxed.bluejam.SequenceParameters` **params** `)`
  - **Usage**
    - For instantiating a tree with custom parameters.
  - **Parameters**
    - `params` -

Methods

- *acceptedRhythm*
  `public Rhythm` **acceptedRhythm**`( )`

  - **Returns** - The rhythm object that this tree will accept in an addNotes() call.

- *addNote*
  `protected boolean` **addNote**`( net.parallaxed.bluejam.NoteSequence` **note**`, int` **childIndex** `)`
  - **Usage**
    * Allows overriding behaviour, to add a note to a particular childIndex of a NoteTree. This method should only ever be called by the parent, _knowing_ that the NoteTree is newly instantiated.
      This forces out other children and can leave the tree unbalanced if used improperly.
  - **Parameters**
    * `note` - The NoteLeaf to insert as a child of this node.
    * `childIndex` - The index at which to insert this note.

- *addNotes*
  `public boolean` **addNotes**`( net.parallaxed.bluejam.NoteSequence` **notes** `)`
  - **Usage**
    * {@inheritDoc}
      This function will add notes until the tree is full.
      Currently optimized to a lookahead depth of 2.
      // NB: THIS METHOD _SHOULD_ ONLY CALL ITSELF WITH A SINGLE NOTE - but it doesn't matter if there's more than 1...

- *clone*
  `public NoteTree` **clone**`( )`
  - **Usage**
    * Returns a cloned NoteTree.
      Every node in the note tree will be replaced by copies of those nodes (this method is recursive).
  - **Returns** - A full copy of the note tree.

- *contains*
  `public boolean` **contains**`( net.parallaxed.bluejam.NoteSequence` **n** `)`

- *depth*
  `public int` **depth**`( )`
  - **Returns** - The depth at which this NoteTree node exists relative to the root

---

- *getChild*
  `public NoteSequence` **getChild**`( int` **index** `)`
  - **Parameters**
    * `index` - The index at which to retrieve the child
  - **Returns** - The child at the passed index

- *getCrossoverReferences*
  `public NoteSequence` **getCrossoverReferences**`( )`
  - **Usage**

    * Used during evolution to get all the references to NoteTree nodes that can undergo Crossover using the swapNotes() method.
  - **Returns** - An array of NoteSequenceInstances that are candidates for crossover.

- *getIncompleteReferences*
  `public NoteSequence` **getIncompleteReferences**`( )`
  - **Usage**
    * This method returns an array of references to nodes in the NoteTree with null children.
      A reference is returned for each null child, so for NoteTrees with multiple empty children, multiple references will be returned.
  - **Returns** - An array of references to nodes with null children.

- *getMutationReferences*
  `public NoteLeaf` **getMutationReferences**`( )`
  - **Usage**
    * Used during evolution to get all the references we need to find nodes that can be mutated.
  - **Returns** - An array of NoteLeaf instances

- *getName*
  `public String` **getName**`( )`
  - **Returns** - The name of this NoteTree (use toString())

- *getNotes*
  `public Iterator` **getNotes**`( )`
  - **Usage**
    * {@inheritDoc}
      This determines how the note will be played.

- *getNoteSequence*
  `public NoteSequence` **getNoteSequence**`( )`
  - **Usage**
    * Returns the contents of the note tree (all notes that extend from the tree as children included). Included to satisfy implementation of Function
  - **See Also**
    * `net.parallaxed.bluejam.Function` (in 3.2.1, page 16)

- *getNumChildren*
  `public int` **getNumChildren**`( )`
  - **Returns** - The number of children this node has

- *getSubTrees*
  `public NoteSequence` **getSubTrees**`( )`

- *parent*
  `public NoteSequence` **parent**`( )`

  - **Returns** - The parent of this NoteSequence

- *removeNotes*
`public void` **removeNotes**`( net.parallaxed.bluejam.NoteSequence` **notes** `)`
  - **Usage**
    - Deletes a branch from this Node and left-shifts all remaining children.

- *rhythmDepth*
`protected int` **rhythmDepth**`( net.parallaxed.bluejam.Rhythm` **r** `)`
  - **Usage**
    - Tree depth is calculated by taking the logarithm of the rhythmic fraction to the base 2, and adding 1.
      i.e. 2∧3 = 8 (giving us depth 3 for a quaver).
      We add 1 to compensate for the total length of the phrase starting at depth 0 (tree root).
      This way, a whole note always starts at depth 1, not depth zero, since:
      2∧0 = 1 (giving us depth 0 for a whole note)
  - **Parameters**
    - `r` - A rhythm enum representing the value of the
  - **Returns** - An integer representing the tree depth.

- *sequenceParameters*
`public SequenceParameters` **sequenceParameters**`( )`
  - **Usage**
    - {@inheritDoc}

- *setSequenceParameters*
`public void` **setSequenceParameters**`(`
`net.parallaxed.bluejam.SequenceParameters` **sequenceParameters** `)`
  - **Usage**
    - To allow this to be a heuristic, we must be able to override it's SequenceParameters object (on construction), since NoteTrees are cloned from this and need to have a reference to it after cloning.

- *swapNotes*
`public boolean` **swapNotes**`( net.parallaxed.bluejam.NoteSequence` **swapOut**`,`
`net.parallaxed.bluejam.NoteSequence` **swapIn** `)`
  - **Usage**
    - Functions as a basic find/replace for tree mutations and other genetic operators like 1-point crossover.
      The first argument is searched for in the children of this node, if that object it found, it is detached from it's parent and the swapIn parameter replaces it.
      Whatever method calls this should take care of setting placing the swapOut NoteSequence back in the right place.
      swapOut should be a reference to some node in this note tree.
      swapIn will be converted to a NoteTree if it is not already.

  - **Parameters**
    - `swapOut` - The NoteSequence to replace in the tree.
    - `swapIn` - The new NoteSequence to place in the position of swapIn.

- *validateNotes*
  `public void` **validateNotes**`( )`
  - **Usage**
    - Simply calls validateNotes on all children and ensures their rhythms add up to _sp.length whole notes.

3.2.16 Class **Pitch**

This enumeration represents the pitches and can be used to determine enharmonic equivalence between pitch classes.

Declaration

public final class Pitch
extends java.lang.Enum

Fields

- public static final Pitch C
  –
- public static final Pitch Cs
  –
- public static final Pitch Db
  –
- public static final Pitch D
  –
- public static final Pitch Ds
  –
- public static final Pitch Eb
  –
- public static final Pitch E
  –
- public static final Pitch F
  –

- public static final Pitch Fs
  –
- public static final Pitch Gb
  –
- public static final Pitch G
  –
- public static final Pitch Gs
  –
- public static final Pitch Ab
  –
- public static final Pitch A
  –
- public static final Pitch As
  –
- public static final Pitch Bb
  –
- public static final Pitch B
  –
- public static final Pitch R
  – R is a special kind of parameter, defining relative pitch. It is used when loading up heuristic trees.

    Notes can be locked at intervals from a given root pitch and, scale, but these are not known until the Heuristic is paired with an individual in a population and initialized.

Methods

- *equals*
  `public final boolean` **equals**`( net.parallaxed.bluejam.Pitch` **p** `)`
  – **Usage**
    * Determines enharmonic equivalence a given pitch.
  – **Parameters**
    * `p` - The pitch to compare with
  – **Returns** - true if in the same pitch class i.e. (Cs,Db) = true.

- *eval*
  `public final Accidental` **eval**`( )`
  – **Usage**
    * Evaluates which accidental is present on a given pitch.

  - **Returns** - Accidental.SHARP for all Pitch.Xs and Accidental.FLAT for all Pitch.Xb

- *getName*
  `public static final String` **getName**`( net.parallaxed.bluejam.Pitch` **pitch** `)`
- *getPitch*
  `public static final Pitch` **getPitch**`( java.lang.String` **pitch** `)`
- *valueOf*
  `public static Pitch` **valueOf**`( java.lang.String` **name** `)`
- *values*
  `public static final Pitch` **values**`( )`

3.2.17 Class **Population**

The population class holds a collection of individuals and evolves them with or without a set of heuristics. Each evolution produces a new population of individuals, which are the product of mating that takes place between selected individuals in the prior population.
Each new population can be thought of as another "generation". The best individuals (as evaluated by the fitness algorithms) should survive and in the final stages, producing a small set of candidates. One should be selected for output, and the others will be placed into the mating pool to produce the next generation. In the case of increased evaluation time on the second generation, another of the chosen solos from the first can be chosen.
A population may be instructed to destroy itself, in which case it returns a reference to the parent population and continues from there (see documentation on the interfaces).

Declaration

public class Population **extends** java.lang.Object

Fields

- public int memberCount
  - Limit on the number of individuals in this population.
- public Individual populous
  - An array of fixed length, with a slot for each individual in the population.

Constructors

- *Population*
  `public` **Population**`( net.parallaxed.bluejam.Population` **p** `)`
  - **Usage**

    * Constructs a skeleton population given the passed population as a template. The constructed population will contain no individuals, but will reflect the parameters of the passed population.
      NB: Does not preserve the _changed status of the given population.
  - **Parameters**
    * `p` - The population to take as a template.

- *Population*
  `public` **Population**`( net.parallaxed.bluejam.SequenceParameters` **sequenceParameters** `)`
  - **Usage**
    * Constructs a population.
  - **Parameters**
    * `sequenceParameters` - The sequenceParameters to use in construction.

- *Population*
  `public` **Population**`( net.parallaxed.bluejam.SequenceParameters` **sequenceParameters**`, int` **memberCount** `)`
  - **Usage**
    * Constructs a population.
  - **Parameters**
    * `memberCount` - Maximum number of individuals in this population

---

- *Population*
  `public` **Population**`( net.parallaxed.bluejam.SequenceParameters` **sequenceParameters**`, int` **memberCount**`,`
  `net.parallaxed.bluejam.HeuristicCollection` **heuristics** `)`
  - **Usage**
    * Constructs a population.
  - **Parameters**
    * `memberCount` - Maximum number of individuals in this population
    * `heuristics` - Collection of heuristics to use while evolving this population

Methods

- *addHeuristic*
  `public void` **addHeuristic**`( net.parallaxed.bluejam.Heuristic` **heuristic** `)`
  - **Usage**
    * Adds a heuristic to the population.
  - **Parameters**
    * `heuristic` - The heuristic to be added

- *addIndividual*
  `public void` **addIndividual**`( net.parallaxed.bluejam.Individual` **i** `)`

- *addIndividuals*
  `public void` **addIndividuals**`( java.util.List` **individuals** `)`
  - **Usage**
    - Adds a list of individuals to the population.
      Throws IndividualAddException with any failed adds.
  - **Parameters**
    - `individuals` - The individuals to add, in the order to add them.
  - **Exceptions**
    - `net.parallaxed.bluejam.exceptions.IndividualAddException` -

- *buildHeuristic*
  `public Heuristic` **buildHeuristic**`( )`
  - **Usage**
    - Should be refactored to a utility method in EvolveHeuristic.
      Will take the MatingPool of candidates, and cycle through each candidate, taking a node from each. - Extend to first crossover point in the first tree, grab and add it - Extend to second crossover point in the second tree, grab it ... - ...
      Alternatively, just take the highest fitness individual in the pool.
      Finally, all notes should be changed to RELATIVE pitches, before serializing out the file.
  - **Returns** - A new Heuristic.

- *evolve*
  `public Population` **evolve**`( )`
  - **Usage**
    - Returns a reference to the evolved population
  - **Returns** - A reference to the evolving thread.

- *getEmptySlots*
  `public ArrayList` **getEmptySlots**`( )`
  - **Usage**
    - Returns an ArrayList of integers with one entry for every empty slot in the populous array.
  - **Returns** - An ArrayList of [i] gaps in the populous, where populous[i] = null.

- *getFittestIndividual*
  `public Individual` **getFittestIndividual**`( )`

- *getIndividual*
  `public Individual` **getIndividual**`( int` **i** `)`
  - **Usage**
    - Returns an individual given it's index in the population.
  - **Parameters**
    - `i` - The index of this individual
  - **Returns** - The individual at index i, or null.

- *getParameters*
  `public PopulationParameters` **getParameters**`( )`
  - **Usage**
    - Retrieves a reference to the parameters of this population
  - **Returns** - A ParameterCollection for this population
  - **See Also**
    - `net.parallaxed.bluejam.JamParamters` (in 3.2.12, page 34)

- *getRandomIndividual*
  `public Individual` **getRandomIndividual**`( )`

- *initialize*
  `public void` **initialize**`( )`
  - **Usage**
    - Initialises the population by creating the individuals and setting them with random values.
      Before calling this method, you must set the parameters for this population through setParameter(), or the hardcoded defaults will be used.
  - **See Also**
    - `net.parallaxed.bluejam.PopulationParameters` (in 3.2.21, page 59)

- *populationSize*
  `public int` **populationSize**`( )`

- *setParameter*
  `public void` **setParameter**`( java.lang.String` **name**`, java.lang.Object` **value** `)`
  - **Usage**
    - Sets a parameter for this population.
  - **Parameters**
    - `name` - The name of the parameter to set
    - `value` - An object of the correct type for this parameter
  - **Exceptions**
    - `net.parallaxed.bluejam.exceptions.ParameterException` -
  - **See Also**
    - `net.parallaxed.bluejam.PopulationParameters` (in 3.2.21, page 59)

3.2.18 Class PopulationParameters

Provides a moderately strongly typed parameter collection for the population properties and constants.

Declaration

public class PopulationParameters
extends java.lang.Object

Fields

- public static final String SEQUENCE
  - Default SequenceParams
- public static final String SELECTION_PRESSURE
  - SelectionPressure parameter name.
    Value of this parameter must be an integer. This also defines the tournament size (if present).
- public static final String SELECTION_TYPE
  - SelectionType parameter name.
- public static final String HEURISTIC_SELECTION_TYPE
  - SelectionType parameter name.
- public static final String INITIALIZATION_TYPE
  - InitializationType parameter name.
- public static final String FITNESS_TYPE
  - InitializationType parameter name.
- public static final String GENOTYPE
  - Genotype (Individual representation) parameter name.

Constructors

- *PopulationParameters*
  `public` **PopulationParameters**`( )`
  - **Usage**
    - Trivial Constructor

Methods

- *_checkType*
  `protected boolean` **_checkType**`( java.lang.String` **name**`, java.lang.Object` **value** `)`
- *getFitnessType*
  `public FitnessType` **getFitnessType**`( )`
  - **Returns** - The FitnessType for this population.
  - **See Also**
    - `net.parallaxed.bluejam.evolution.FitnessType` (in 5.2.18, page 87)
- *getGenotype*
  `public Genotype` **getGenotype**`( )`

- **Returns** - The Genotype for this population
- **See Also**
  - `net.parallaxed.bluejam.evolution.Genotype` (in 3.2.8, page 25)

- *getInitializationType*
  `public InitializationType` **getInitializationType**`( )`
  - **Returns** - The InitializationType for this population.
  - **See Also**
    - `net.parallaxed.bluejam.evolution.InitializationType` (in 3.2.8, page 25)

- *getParameter*
  `public Object` **getParameter**`( java.lang.String` **parameter** `)`
  - **Returns** - An object of the value of that parameter, or null if not found.

- *getSelectionPressure*
  `public Integer` **getSelectionPressure**`( )`
  - **Returns** - The SelectionPressure for this population.
  - **See Also**
    - `net.parallaxed.bluejam.evolution.SelectTournament` (in 3.2.8, page 26)

- *getSelectionType*
  `public SelectionType` **getSelectionType**`( )`
  - **Returns** - The SelectionType for this population.
  - **See Also**
    - `net.parallaxed.bluejam.evolution.SelectionType` (in 3.2.8, page 26)

- *getSequenceParameters*
  `public SequenceParameters` **getSequenceParameters**`( )`
  - **Usage**
    - JamParameters can also be obtained through here.
  - **Returns** - The SequenceParameters instance used in this population.

- *setParameter*
  `public void` **setParameter**`( java.lang.String` **name**`, java.lang.Object` **value** `)`
  - **Usage**
    - Sets a parameter using moderately strict type checking.
      Value and name must both be non-null and supported by the Parameter collection type (either Individual or Population).
  - **Parameters**
    - `name` - The name of the parameter to set
    - `value` - The object value to set the parameter to.
  - **Exceptions**
    - `net.parallaxed.bluejam.exceptions.ParameterException` -

3.2.19 Class **Rhythm**

This Enum contains all the rhythm's BlueJam supports.
Consequently, the maximum depth of any NoteTree (in the default implementation) = the number of constants in this enum, plus 1 (currently, 8).

Declaration

public final class Rhythm
extends java.lang.Enum

Fields

- public static final Rhythm SEMIBREVE
  –
- public static final Rhythm MINIM
  –
- public static final Rhythm CROTCHET
  –
- public static final Rhythm QUAVER
  –
- public static final Rhythm SEMIQUAVER
  –
- public static final Rhythm DEMIQUAVER
  –
- public static final Rhythm HEMIQUAVER
  –

Methods

- *eval*
  `public int eval( )`
  - **Usage**
    - Return the reciprocal of the fraction represented by this rhythmic value. The reciprocal is 1/evalR().
  - **Returns** - The denominator of the rhythmic fraction

- *evalR*
  `public float evalR( )`

  - **Usage**
    - Return the decimal fraction of a bar represented by this note in regular notation and a 4/4 time signature.
      Some hacks may be needed if using this to represent an odd or irrational time signature, all others can be calculated relative to these values.
  - **Returns** - A fraction representing that note's value.
  - **Exceptions**
    - `java.lang.RuntimeException` - if there is no hardcoded value for that rhythm

- *getRhythm*
  `public static Rhythm` **getRhythm**`( int` **number** `)`
  - **Usage**
    - Gets a rhythm given it's eval() value.
  - **Parameters**
    - `number` - The eval() value of the rhythm (1/x)
  - **Returns** - The Rhythm enumeration for the given number.

- *getRhythm*
  `public static Rhythm` **getRhythm**`( java.lang.String` **s** `)`
  - **Usage**
    - Returns the rhythm matching the supplied name, e.g. "Semibreve", "Demiquaver" etc.
      Case Insensitive.
  - **Parameters**
    - `s` - The name of the rhythm
  - **Returns** - The rhythm enumeration for the given name

- *increase*
  `public Rhythm` **increase**`( int` **steps** `)`
  - **Parameters**
    - `steps` - The factor to increase the rhythm by (i.e. 1 doubles the rhythm, 2 triples, etc)
  - **Returns** - a new (augmented) rhythm, based on the number of steps.

- *valueOf*
  `public static Rhythm` **valueOf**`( java.lang.String` **name** `)`

- *values*
  `public static final Rhythm` **values**`( )`

3.2.20 Class **Scale**

This Enum describes how to produce each scale using stepped jumps over pitch classes.
Each encoding is minimal, and describes how to navigate from the root (I) through II-IV in the scale. VIII is the root repeated.
In the blues (I, bIII, IV, bV, V bVII), the flattened fifth seems traditionally noted as a "sharp" or natural note. I don't know why this occurs but I see it everywhere. Therefore the accidental map for the blues notes the fourth as flat.

Since bV (flat fifth) is enharmonic to IV# (sharp fourth) in all cases I see no problem with re-implementing this using flats only if desired.
This enum can also map accidentals.

Declaration

public abstract class Scale
extends java.lang.Object

Constructors

- *Scale*
  `public` **Scale**`( )`

Methods

- *eval*
  `public final int` **eval**`( )`
- *eval*
  `public abstract Accidental` **eval**`( net.parallaxed.bluejam.Pitch [] pitchClass, net.parallaxed.bluejam.Pitch rootPitch, int noteNumber )`
  - **Usage**
    - This method provides a definite function call to find out which accidental should be present on a note.
      noteNumber is not strictly needed for this.
  - **Parameters**
    - `pitchClass` - The pitch class of the note.
    - `noteNumber` - The note number, n, corresponds to accident[n]. The nth note in that scale.
    - `rootPitch` - Passing in the root pitch reduces the calculation time.
  - **Returns** - The accidental of the note.
- *getInstance*
  `protected static Scale` **getInstance**`( )`
  - **Usage**
    - Singleton instance returning method which all subclasses must implement and make visible.
  - **Returns** - A scale of the type that has been instantiated.

3.2.21 Class Scale.BLUES

Defines a blues scale with hexatonic pitch representation.

DECLARATION

public static final class Scale.BLUES
extends net.parallaxed.bluejam.Scale

METHODS

- *eval*
  `public final Accidental` **eval**`( net.parallaxed.bluejam.Pitch [] ` **pitchClass**`, net.parallaxed.bluejam.Pitch ` **rootPitch**`, int ` **noteNumber**` )`
- *getInstance*
  `public static final Scale` **getInstance**`( )`

3.2.22 CLASS Scale.MAJOR

Defines a standard major scale.

DECLARATION

public static final class Scale.MAJOR
extends net.parallaxed.bluejam.Scale

CONSTRUCTORS

- *Scale.MAJOR*
  `public` **Scale.MAJOR**`( )`

METHODS

- *eval*
  `public final Accidental` **eval**`( net.parallaxed.bluejam.Pitch [] ` **pitchClass**`, net.parallaxed.bluejam.Pitch ` **rootPitch**`, int ` **noteNumber**` )`
- *getInstance*
  `public static final Scale` **getInstance**`( )`

3.2.23 CLASS Scale.MINOR

Defines a standard minor scale.

DECLARATION

public static final class Scale.MINOR
extends net.parallaxed.bluejam.Scale

CONSTRUCTORS

- *Scale.MINOR*
  `public` **Scale.MINOR**`( )`

METHODS

- *eval*
  `public final Accidental` **eval**`( net.parallaxed.bluejam.Pitch [] ` **pitchClass**`,`
  `net.parallaxed.bluejam.Pitch` **rootPitch**`, int` **noteNumber** `)`
- *getInstance*
  `public static final Scale` **getInstance**`( )`

3.2.24 CLASS ScaledSet

Given a set of rules for the scale, ScaledSet can work out the set of notes that fit that scale, over a given number of octaves. A set of the chromatic notes in that scale can also be produced.

DECLARATION

public class ScaledSet
extends net.parallaxed.bluejam.NoteLeafSet

FIELDS

- public Pitch pitchClass
  - This array describes pitch classes. Pitch classes encapsulate enharmonic notes and divide the pitches into orders of 12, which can then be used to calculate the given scale.

CONSTRUCTORS

- *ScaledSet*
  `public` **ScaledSet**`( net.parallaxed.bluejam.Pitch` **rootPitch**`,`
  `net.parallaxed.bluejam.Scale` **scale**`, int` **minOctave**`, int` **maxOctave** `)`
  - **Usage**

    - Constructs a set of all notes of the right pitch over a given octave range and scale. Limits are always exclusive, so for Pitch.A on octaves 3-5 will include A3, and all subsequent notes up to A6 (but not A6).
  - **Parameters**
    - `rootPitch` - The root pitch of the scale
    - `scale` - The scale to use for generating notes
    - `minOctave` - The lower 8ve limit
    - `maxOctave` - The upper 8ve limit.

Methods

- *addChromaticNote*
  `public boolean` **addChromaticNote**`( net.parallaxed.bluejam.NoteLeaf` **n** `)`
  - **Usage**
    - Adds a chromatic note leaf to the set. TODO NOT IMPLEMENTED
  - **Parameters**
    - `n` - The Note to add to this scaled set
  - **Returns** - Whether the operation succeded or not.

- *getRandom*
  `public NoteLeaf` **getRandom**`( net.parallaxed.bluejam.NoteSequence` **n** `)`

- *octaveChangeProbability*
  `public int` **octaveChangeProbability**`( )`
  - **Returns** - The probability we will change from the current octave.

- *octaveChangeProbability*
  `public void` **octaveChangeProbability**`( int` **probability** `)`
  - **Usage**
    - If we're at the cusp of the next octave, here's the chance that we have of going up or down. TODO Include threshold for cusp.
  - **Parameters**
    - `probability` - The probability of changes

3.2.25 Class SequenceParameters

Currently, custom tree parameters aren't supported, so this default instantiation should cover the basics.

Declaration

public class SequenceParameters **extends** java.lang.Object

Fields

- public volatile boolean Changed
  - Has this SequenceParameters object been changed recently?
- public JamParamters Jam
  - The JamParameters for the current sequence. TODO Work on dynamic changing at runtime.
- public int length
  - The number of whole notes in this Sequence.
    i.e. 4 whole notes = 4 bars, so length=4

Constructors

- *SequenceParameters*
  `public` **SequenceParameters**`( )`
  - **Usage**
    - Trivial SequenceParameters constructor.

- *SequenceParameters*
  `public` **SequenceParameters**`( net.parallaxed.bluejam.JamParamters` **jam** `)`
  - **Usage**
    - Initializes a set of SequenceParameters with the passed JamParameters
  - **Parameters**
    - `jam` - The JamParameters defining rootPitch, scale, etc.

Methods

- *beatCount*
  `public int` **beatCount**`( )`
  - **Usage**
    - The nominator of the time signature - the number of beat units per bar.
  - **Returns** - The number of beats per beat unit

- *beatUnit*
  `public int` **beatUnit**`( )`
  - **Usage**
    - The beat unit is the denominator of the time signature.
  - **Returns** - The note we measuring in (usually quarter-note, or Rhythm.CROTCHET)

- *timeSignature*
  `public String` **timeSignature**`( )`

  - **Returns** - The time signature of the sequence in string "x/y" format

---

- *timeSignature*
  `public void` **timeSignature**`( java.lang.String` **timeSignature** `)`
  - **Usage**
    - Time signature is parsed to set the private variables

3.2.26 Class TerminalSet

Wraps an ArrayList to provide set-like functionality (one unique instance per set).

Declaration

public abstract class TerminalSet **extends** java.util.AbstractSet

Constructors

- *TerminalSet*
  `public` **TerminalSet**`( )`

Methods

- *add*
  `public boolean` **add**`( net.parallaxed.bluejam.Terminal` **terminal** `)`
  - **Usage**
    - Returns false if either the terminal supplied is null or the collection already contains the terminal.
      Otherwise, adds the item to the collection.

- *contains*
  `public boolean` **contains**`( java.lang.Object` **o** `)`
  - **Parameters**
    - `o` - The Terminal to search for in this TerminalSet
  - **Returns** - True if this set contains the passed Object

---

- *get*
  `public Terminal` **get**`( int` **index** `)`
  - **Parameters**
    - `index` - The index at which to look for the terminal.
  - **Returns** - a terminal based on it's index in the set

- *getRandom*
  `public Terminal` **getRandom**`( )`

  - **Returns** - A random terminal from the set.

- *iterator*
  `public Iterator` **iterator**`( )`
  - **Usage**
    * Returns an Iterator over the elements in this TerminalSet.
  - **See Also**
    * `java.util.AbstractCollection.iterator()` (in 3.2.15, page 45)

- *size*
  `public int` **size**`( )`
  - **Returns** - The number of terminals in this set.

3.2.27 Class **TreeParser**

This class reads in information from passed tree files, which normally represent a serialized layout of a NoteSequence. These files come in two flavours, .heuristic and .tree, each specifying roughly the same parameters. The different extension is merely a semantic pragma to inform the user what the file contains (a heuristic, or just a tree).
.tree files may vary in what they contain, but they are normally serial stores of tree structures at a certain point in the evolution (they represent trees assigned to individuals) - or they are arbitrarily created files used for testing Tree parser provides only one method for accessing the parsed tree. The user can call getNoteTree() and cast the result to whatever is desirable. Errors during parsing are printed out to the terminal.

Declaration

public class TreeParser
extends java.lang.Object

Constructors

- *TreeParser*
  `public` **TreeParser**`( java.io.File  treeFile )`
  - **Usage**
    * Creates a TreeParser given a reference to a java.io.File object.
  - **Parameters**
    * `treeFile` -

- *TreeParser*
  `public` **TreeParser**`( java.lang.String  treeFile )`
  - **Usage**
    * Creates a tree parser given a relative path.
  - **Parameters**
    * `treeFile` - The file to parse.

Methods

- *getNoteTree*
  `public NoteTree` **`getNoteTree( )`**
  - **Returns** - The parsed note tree, can be cast to the desired type.

Chapter 4

Package net.parallaxed.bluejam.pd

Package Contents *Page*

Classes

Configure 68
Waits for a PD bang to kickstart the evolution and playback of evolved solos.

4.1 Classes

4.1.1 Class **Configure**

Waits for a PD bang to kickstart the evolution and playback of evolved solos. Also configures various parameters of the evolution before running the a generation.
This object supports two modes - evolution and single file. In single file mode, the object is initialized with an argument from PD, pointing to a tree file. The program loads this tree file and all subsequent playback will involve that tree file. The user can reload this single file after making changes to adjust the sound produced. This mode helps when designing heuristics, as the user can hear the programmed heuristic at any tempo and relative to any pitch.
In evolution mode, the program loads all heuristics present in ${pd_home}/extra/bluejam/config (all .heuristic files), along with all models if present (.../model-${PITCH}.m), and will run evolution cycles when triggered on or off.
Custom scales are not supported (yet). TODO Stop errors from killing instantiation.

Declaration

public class Configure
extends com.cycling74.max.MaxObject
implements com.cycling74.max.Executable, net.parallaxed.bluejam.playback.Listener

Constructors

- *Configure*
  `public` **Configure**`( )`
  - **Usage**
    * Instantiates the configuration to operate in evolution mode (no arguments).

- *Configure*
  `public` **Configure**`( com.cycling74.max.Atom [] args )`
  - **Usage**
    * If constructed with arguments, only the supplied filename is played (singleFileMode).
  - **Parameters**
    * `args` - A string array of all arguments passed to the program, only args[0] is read.

Methods

- *bang*
  `protected void` **bang**`( )`
  - **Usage**
    * This instructs the program to get ready and listen on the MIDI lines to detect a root pitch key.

- *execute*
  `public void` **execute**`( )`
  - **Usage**
    - Keeps the ball rolling; this function is called by the inner MaxClock. The delay of the MaxClock is set to the duration of the note.
      TODO - This is very primitive sequencing, could introduce separate clock for timing.

- *inlet*
  `protected void` **inlet**`( float  f )`
  - **Usage**
    - Deals with floats arriving at inlets; parameters like populationSize, numberOfGenerations etc.

- *listen*
  `public void` **listen**`( net.parallaxed.bluejam.NoteSequence  n )`
  - **Usage**
    - Queue's up a NoteSequence in the buffer and pre-meditatively evaluates it.

- *reload*
  `public void` **reload**`( )`
  - **Usage**
    - Reloads the file or completely restarts the evolution if running.

Chapter 5

Package net.parallaxed.bluejam.evolution

Package Contents *Page*

Interfaces

Breeder 69
Classes implementing this interface must fill a population to it's maximum capacity through the use of crossover and mutation to create new individuals.

IndividualEvaluator 69
Describes what arguments an implementing class (fitness function) must take.

IndividualSelector 69
Implementing classes should design selection algorithms that can accept a population as input, and produce another (skeleton) population as output.

NoteSequenceInitializer 69
Initializers are singletons that can accept a reference to a NoteSequence and fill it with notes using the algorithm they define.

Classes

FitnessContour 69
Computes a fitness value for the contour of the supplied NoteSequence.

FitnessDistance 69
Implements a measure of fitness by comparing the result of an individual to its original heuristic through means analysing each note in the sequence to see that it is sufficiently displaced from those around it.

FitnessInterval 69
This class selects a random fitness method to apply.

FitnessRandom 69
Picks a random fitness evaluation method and returns the result.

FitnessStacked 69
Tracks which individuals have already been measured by the function, and applies different fitness measures each time.

FitnessType 87
Defines which types of fitness are available to the system.

Genotype 25
Names and stored reference to the underlying implementations of Genomes available in the system.

HeuristicSelectionType 69

When initializing the population, the set of loaded heuristics can be assigned randomly or evenly.

InitializationType 25
Defines the possible initialization types for the evolution process.

InitializeGrow 69
This form of the grow algorithm psuedorandomly selects a rhythm with which to assign the note before adding it to the tree (having the effect of choosing all functions down to that depth, and the final terminal).

InitializeHeuristicTree 72
Initializes note sequences using crossover on the heuristics, as defined by the TreeBreeder method, breed(NoteTree,NoteTree).

InitializeRandom 72
Implements a random initialization.

NoteContext 73
The NoteContext class is a wrapper for NoteCollection that gathers some information about that particular collection of notes.

NoteContext.Contour 73
Contour defines the overall progression of a NoteSequence.

PropertyFactory 74
Defines which class sets properties for an Individual.

RhythmInitializer 75
Initializes Rhythm and other contextual properties for a passed note sequence using a static probabilistic model.

SelectionParameters 75
This class follows a generic parameter pattern where the user can pass in parameters for the selection algorithm.

SelectionType 26
An enum containing the possible selection algorithms that can be used.

SelectProportional 76
Implements a proportional (Roulette Wheel) selection algorithm

SelectTournament 26
Performs a variant of tournament selection.

TreeBreeder 77
TreeBreeder is the default implementation of the Breeder interface for recombining and mutating NoteTrees, given an initial skeletal population.

5.1 Interfaces

5.1.1 Interface **Breeder**

Classes implementing this interface must fill a population to it's maximum capacity through the use of crossover and mutation to create new individuals.
The initial population passed to the function represents the mating pool, so the algorithm should extract initial individuals from the referenced object. If no population is passed, this throws an exception. Nothing is returned by implementing classes, the population referenced is filled with children in-situ.

Declaration

public interface Breeder

Methods

- *breed*
  `public void` **breed**`( net.parallaxed.bluejam.Population` **population** `)`
  - **Usage**
    - Uses a skeletal population containing a collection of parents and fills up the population to the full memberCount quota using methods of crossover and mutation.
      Nothing should be returned by this method.
  - **Parameters**
    - `population` - The population to breed.

5.1.2 Interface **IndividualEvaluator**

Describes what arguments an implementing class (fitness function) must take. Like the NoteSequenceInitializer interface, the evaluators assume a singleton implementation, whereby the NoteSequences to be evaluated are passed in through the evaluate() function. Evaluators are singletons and should not store any volatile state in the class for ThreadSafety. NoteSequenceEvaluators may be optimized for different dataTypes, but the default implementation of evaluate() should return *some* value for fitness, no matter what the NoteSequence type is. In the default implementation, all fitness values are normalised (in the range0-1).
The implementing should throw an exception if required knowledge of the dataType is not found, or output a warning if knowledge was expected but not present.The Selector function however should be strictly compatible with the type of value returned.

Declaration

public interface IndividualEvaluator

METHODS

- *evaluate*
  `public double` **evaluate(** `net.parallaxed.bluejam.Individual` **individual)**
  - **Usage**
    - Evaluates the given NoteSequence in an individual, providing a double value for the fitness of that sequence.
  - **Parameters**
    - `individual` - The Individual to Evaluate
  - **Returns** - A value for the fitness of the NoteSequence
  - **See Also**
    - `net.parallaxed.bluejam.Individual` (in 3.2.10, page 29)

5.1.3 INTERFACE IndividualSelector

Implementing classes should design selection algorithms that can accept a population as input, and produce another (skeleton) population as output. These can be of an arbitrary size, but a method to alter the size of the skeletal population should be provided.
Various breeding methods can be called over this subpopulation to flesh it out, before the selection procedure is called again to produce another generation of individuals.

DECLARATION

```
public interface IndividualSelector
```

METHODS

- *select*
  `public Population` **select(** `net.parallaxed.bluejam.Population` **pop,** `int` **numberOfIndividuals)**
  - **Usage**
    - This function performs the selection, returning a smaller population as set by newPopulationCount(int).
  - **Parameters**
    - `pop` - The population to perform selection on.
    - `numberOfIndividuals` - The number of individuals to select from the population.
  - **Returns** - A new population, ready to go into breeding.

5.1.4 INTERFACE NoteSequenceInitializer

Initializers are singletons that can accept a reference to a NoteSequence and fill it with notes using the algorithm they define.
Use of this interface separates knowledge of the algorithm from the rest of the evolution.
*** NB Since Initializers are singletons, they should all provide a getInstance() method, which the framework will call to get a reference before it calls initialize() ***

DECLARATION

public interface NoteSequenceInitializer

METHODS

- *initialize*
  `public void` **initialize**`( net.parallaxed.bluejam.NoteSequence` **notes**`, net.parallaxed.bluejam.PopulationParameters` **params** `)`
  - **Usage**
    - This method should fill up the note sequence, adding notes as appropriate. The algorithm implementation should be specific to the Genotype structure, so the implementing class should check that params supplies the right value for Genotype. Note that this method does not return anything, the NoteSequence will be altered appropriately after method execution.
  - **Parameters**
    - `notes` - The NoteSequence to initialize.
    - `params` - The parameters specified for the individual that calls this method.
  - **See Also**
    - `net.parallaxed.bluejam.evolution.Genotype` (in 3.2.8, page 25)
    - `net.parallaxed.bluejam.NoteSequence` (in 3.2.3, page 19)
    - `net.parallaxed.bluejam.Note` (in 1.2.1, page 5)

5.2 Classes

5.2.1 CLASS FitnessContour

Computes a fitness value for the contour of the supplied NoteSequence.
This is optimised to work with NoteTree, but will return a value for sequences that are not NoteTrees.

DECLARATION

public class FitnessContour
extends java.lang.Object
implements IndividualEvaluator

METHODS

- *evaluate*
  `public double` **evaluate**`( net.parallaxed.bluejam.Individual` **individual** `)`
  - **Usage**
    - {@inheritDoc}

- *getInstance*
  `public static FitnessContour` **getInstance**`( )`

5.2.2 Class **FitnessDistance**

Implements a measure of fitness by comparing the result of an individual to its original heuristic through means analysing each note in the sequence to see that it is sufficiently displaced from those around it. The bracket of notes that are examined is defined by the DistanceThreshold, so for one note either side (the default) that's 3.
The final fitness score is given by adding up the measure of how similar the final output is, then taking the logarithm of that score before putting it into a function to get the final fitness value.
To give an idea of numbers, a similarity score of about 300-400 is virtually identical, 200-300 is too similar, 100-200 is similar, 50-100 is good, 10-50 is good but getting dissimilar, 0-10 is unrecognisable from the original heuristic. We should start generating new heuristics if this measure is returning the maximum possible fitness.

Declaration

public class FitnessDistance
extends java.lang.Object
implements IndividualEvaluator

Methods

- *evaluate*
  `public double` **evaluate**`( net.parallaxed.bluejam.Individual` **individual** `)`
  - **Usage**
    - Compare each note in the Sequence property for property.
      Use a minimal threshold distance to check if the properties are shared either side.
      If properties are shared, we increment the score. If the properties are shared at the midpoint of the window defined by DistanceThreshold, we increment the score again.
- *getInstance*
  `public static FitnessDistance` **getInstance**`( )`
  - **Returns** - The singleton instance of FitnessDistance

5.2.3 Class **FitnessInterval**

This class selects a random fitness method to apply.

DECLARATION

public class FitnessInterval
extends java.lang.Object
implements IndividualEvaluator

FIELDS

- public static int IntervalWindow
  - How many repetitions before we start to reduce the fitness?
- public static int BackoffPercent
  - Defines how many cases of repetition should be "let through"

METHODS

- *evaluate*
  `public double` **evaluate**`( net.parallaxed.bluejam.Individual` **individual** `)`
  - **Usage**
    - {@inheritDoc}
      The way to evaluate intervals is simply to skip over the noteSequence and find out how many notes have interval >3.
      This is acceptable in a few contexts, and octave jumps are permitted (in few amounts)
- *getInstance*
  `public static FitnessInterval` **getInstance**`( )`
  - **Returns** - An instance of the FitnessInterval Evaluator

5.2.4 CLASS **FitnessRandom**

Picks a random fitness evaluation method and returns the result.
This is not used in the default implementation of BlueJam.

DECLARATION

public class FitnessRandom
extends java.lang.Object
implements IndividualEvaluator

Methods

- *evaluate*
  `public double` **evaluate**`( net.parallaxed.bluejam.Individual` **individual** `)`
- *getInstance*
  `public static FitnessRandom` **getInstance**`( )`

5.2.5 Class FitnessStacked

Tracks which individuals have already been measured by the function, and applies different fitness measures each time.
This class compounds various fitness methods.
INTERVALS ->HEURISTIC-DIFFERENCE ->CONTOUR
One problem with this might be that individuals with a propensity for only one thing will be discarded, and never make it to the mating pool, where they may have very good genes for accomplishing a particular task.

Declaration

public class FitnessStacked **extends** java.lang.Object **implements** IndividualEvaluator

Methods

- *evaluate*
  `public double` **evaluate**`( net.parallaxed.bluejam.Individual` **individual** `)`
  - **Usage**
    - {@inheritDoc}
- *getInstance*
  `public static FitnessStacked` **getInstance**`( )`
  - **Returns** - An instance of FitnessStacked.

5.2.6 Class FitnessType

Defines which types of fitness are available to the system.
This parameter is also specified by default in PopulationParameters. Different Individuals can use distinct fitness measures if they wish.

Declaration

public final class FitnessType **extends** java.lang.Enum

Serializable Fields

- private Class _impl
  –

Fields

- public static final FitnessType STACKED
  –
- public static final FitnessType RANDOM
  –
- public static final FitnessType INTERVAL
  –
- public static final FitnessType CONTOUR
  –
- public static final FitnessType DISTANCE
  –

Methods

- *eval*
  `public Class` **eval**`( )`
  – **Returns** - The implementing class.
- *valueOf*
  `public static FitnessType` **valueOf**`( java.lang.String` **name** `)`
- *values*
  `public static final FitnessType` **values**`( )`

5.2.7 Class **Genotype**

Names and stored reference to the underlying implementations of Genomes available in the system. The default is a NoteTree.

Declaration

public final class Genotype **extends** java.lang.Enum

Serializable Fields

- private Class _genotypeImpl
  –

Fields

- public static final Genotype NOTE_TREE
  – The default genome implementation.

Methods

- *eval*
  `public Class eval( )`
  – **Usage**
    * Returns a refernece to the class for the implementing genome type. This type *must* implement NoteSequence, or undefined behaviour will occur.
  – **Returns** - A parameterizable Class instance for the enumerable genotype.
- *valueOf*
  `public static Genotype valueOf( java.lang.String name )`
- *values*
  `public static final Genotype values( )`

5.2.8 Class **HeuristicSelectionType**

When initializing the population, the set of loaded heuristics can be assigned randomly or evenly. HeuristicSelectionType.EVEN will iterate through the list, whereas HeuristicSelectionType.RANDOM will pick any from the list.

Declaration

public final class HeuristicSelectionType **extends** java.lang.Enum

Fields

- public static final HeuristicSelectionType EVEN
  –
- public static final HeuristicSelectionType RANDOM
  –

Methods

- *valueOf*
  `public static HeuristicSelectionType` **valueOf**`( java.lang.String` **name** `)`
- *values*
  `public static final HeuristicSelectionType` **values**`( )`

5.2.9 Class InitializationType

Defines the possible initialization types for the evolution process.
To add a new Initialization algorithm, create the class and define it here, then you can specify the enum value in PopulationParameters.

Declaration

public final class InitializationType
extends java.lang.Enum

Serializable Fields

- private Class _initImpl
  –

Fields

- public static final InitializationType RANDOM
  – Random initialization
- public static final InitializationType GROW
  – Grow initialization
- public static final InitializationType HEURISTIC
  – Contoured initialization

Methods

- *eval*
  `public Class` **eval**`( )`
  – **Returns** - The implementing class.
- *valueOf*
  `public static InitializationType` **valueOf**`( java.lang.String` **name** `)`
- *values*
  `public static final InitializationType` **values**`( )`

5.2.10 Class **InitializeGrow**

This form of the grow algorithm psuedorandomly selects a rhythm with which to assign the note before adding it to the tree (having the effect of choosing all functions down to that depth, and the final terminal).

Declaration

public class InitializeGrow
extends java.lang.Object
implements NoteSequenceInitializer

Methods

- *getInstance*
  `public static InitializeGrow` **getInstance**`( )`
  - **Returns** - An Instance of the "Grow" initialization algorithm.
- *initialize*
  `public void` **initialize**`( net.parallaxed.bluejam.NoteSequence` **notes**`, net.parallaxed.bluejam.PopulationParameters` **params** `)`
  - **Usage**
    - {@inheritDoc}

5.2.11 Class **InitializeHeuristicTree**

Initializes note sequences using crossover on the heuristics, as defined by the TreeBreeder method, breed(NoteTree,NoteTree).
This initializer performs crossover on the NoteSequences passed to it (which are clones of the Heuristics, if present).
On the first call of the function, the passed NoteSequence is stored in the Initializer. On the second call, crossover is performed and the memory of the Initializer is erased.

Declaration

public class InitializeHeuristicTree
extends java.lang.Object
implements NoteSequenceInitializer

Methods

- *getInstance*
  `public static InitializeHeuristicTree` **getInstance**`( )`

- *initialize*
  `public void` **initialize**`( net.parallaxed.bluejam.NoteSequence` **notes**`,`
  `net.parallaxed.bluejam.PopulationParameters` **params** `)`
  - **Usage**
    - {@inheritDoc}
      Satisfies implementation of NoteSequenceInitializer.

5.2.12 Class **InitializeRandom**

Implements a random initialization. This initially fills the tree out to crotchet depth, then randomly selects This is not a true FULL method, it is adapted to function better given the musical domain.
In this case we fill out to a LIMIT max-depth (normally Rhythm.CROTCHET), and grow the rest.

Declaration

public class InitializeRandom
extends java.lang.Object
implements NoteSequenceInitializer

Methods

- *getInstance*
  `public static InitializeRandom` **getInstance**`( )`
- *initialize*
  `public void` **initialize**`( net.parallaxed.bluejam.NoteSequence` **notes**`,`
  `net.parallaxed.bluejam.PopulationParameters` **params** `)`
  - **Usage**
    - {@inheritDoc}

5.2.13 Class **NoteContext**

The NoteContext class is a wrapper for NoteCollection that gathers some information about that particular collection of notes.
NoteContext also provides the Contour enum.

Declaration

public class NoteContext
extends net.parallaxed.bluejam.NoteCollection

Serializable Fields

- private boolean _changed
  - Has this context been changed?
- private int _contourThreshold
  -
- private NoteContext.Contour _contour
  -

Constructors

- *NoteContext*
  `public` **NoteContext**`( net.parallaxed.bluejam.SequenceParameters` **sequenceParameters** `)`
  - **Usage**
    - Instantiates a NoteContext.

Methods

- *add*
  `public void` **add**`( int` **index**`, net.parallaxed.bluejam.Note` **element** `)`
  - **Usage**
    - {@inheritDoc}

- *add*
  `public boolean` **add**`( net.parallaxed.bluejam.Note` **element** `)`
  - **Usage**
    - {@inheritDoc}

- *clear*
  `public void` **clear**`( )`
  - **Usage**
    - {@inheritDoc}

- *contour*
  `public NoteContext.Contour` **contour**`( )`

- *remove*
  `public Note` **remove**`( int` **index** `)`
  - **Usage**
    - {@inheritDoc}

- *remove*
  `public boolean` **remove**`( java.lang.Object` **o** `)`
  - **Usage**
    * {@inheritDoc}

5.2.14 Class NoteContext.Contour

Contour defines the overall progression of a NoteSequence. This is calculated using a minimum number of notes called the "contour threshold".
If the calculating context has less notes than the contour threshold, no contour can be assigned.

Declaration

public static final class NoteContext.Contour
extends java.lang.Enum

Fields

- public static final NoteContext.Contour UP
  - A NoteSequence that predominantly goes upward
- public static final NoteContext.Contour DOWN
  - A NoteSequence that predominantly goes downward
- public static final NoteContext.Contour NONE
  - A NoteSequence that does not have a predominant direction.

Methods

- *valueOf*
  `public static NoteContext.Contour` **valueOf**`( java.lang.String` **name** `)`
- *values*
  `public static final NoteContext.Contour` **values**`( )`

5.2.15 Class PropertyFactory

Defines which class sets properties for an Individual.
Other implementations may be devised and placed in the enum.

Declaration

public final class PropertyFactory
extends java.lang.Enum

Serializable Fields

- private Class _initImpl
  –

Fields

- public static final PropertyFactory RHYTHM_INITIALIZER
  –

Methods

- *eval*
  `public Class` **eval**`( )`
  - **Usage**
    - Returns the implementing class.
  - **Returns** -
- *valueOf*
  `public static PropertyFactory` **valueOf**`( java.lang.String` **name** `)`
- *values*
  `public static final PropertyFactory` **values**`( )`

5.2.16 Class **RhythmInitializer**

Initializes Rhythm and other contextual properties for a passed note sequence using a static probabilistic model.
This Initializer will track the context in which it is being initialized, which should aid the evolution by producing programs which are more likely to follow a musically "fit" pattern, rather than being completely random. This is because our initial population size is comparatively quite small, and we need to guarantee a short termination time.
This implementation keeps track of context (i.e. tracks the last notes that were added), assigning properties based on notes that have been processed since the last clearContext() call.
This implementation is BIASED towards returning Quavers. Other implementations can take on any other desirable bias.
Specialised to NoteTree implementation - will call getAcceptedRhythm() to validate the added rhythm.
TODO Dynamic rhythm probabilities.

Declaration

public class RhythmInitializer
extends java.lang.Object
implements NoteSequenceInitializer

Methods

- *getInstance*
  `public static RhythmInitializer` **getInstance**`( )`
  - **Returns** - An instance of the PropertyInitializer

- *getNextRhythm*
  `public static Rhythm` **getNextRhythm**`( )`
  - **Usage**
    - Returns a rhythm from the distribution specified by the static initializers of this class.
  - **Returns** - An instance of Rhythm
  - **See Also**
    - `net.parallaxed.bluejam.Rhythm` (in 3.2.22, page 59)

- *initialize*
  `public void` **initialize**`( net.parallaxed.bluejam.NoteSequence` **notes**`, net.parallaxed.bluejam.PopulationParameters` **params** `)`
  - **Usage**
    - Initializes properties of Rhythm.
      Default implementation initializes rhythm and swing on the passed NoteSequence.

- *initialize*
  `public void` **initialize**`( net.parallaxed.bluejam.NoteTree` **notes**`, net.parallaxed.bluejam.PopulationParameters` **params** `)`
  - **Usage**
    - This function does the same as a regular initialize() but does not assign arbitrary rhythms to the sequence. TODO Refactor duplicated code.
  - **Parameters**
    - `notes` - The notes to initialize
    - `params` - The PopulationParameters to use.

5.2.17 Class SelectionParameters

This class follows a generic parameter pattern where the user can pass in parameters for the selection algorithm.
In the default BlueJam implementation this is only used to pass in TournamentSize to SelectTournament.

Declaration

public class SelectionParameters
extends java.util.HashMap

Constructors

- *SelectionParameters*
  `public` **SelectionParameters**`( )`

5.2.18 Class SelectionType

An enum containing the possible selection algorithms that can be used. These can be set on a per-population basis using PopulationParameters

Declaration

public final class SelectionType
extends java.lang.Enum

Serializable Fields

- private Class _impl
  –

Fields

- public static final SelectionType PROPORTIONAL
  – A proportional selection algorithm
- public static final SelectionType TOURNAMENT
  – A Tournament-based Selection selection Algorithm

Methods

- *eval*
  `public Class` **eval**`( )`
  – **Returns** - The implementing class.
- *valueOf*
  `public static SelectionType` **valueOf**`( java.lang.String` **name** `)`
- *values*
  `public static final SelectionType` **values**`( )`

5.2.19 Class SelectProportional

Implements a proportional (Roulette Wheel) selection algorithm

Declaration

public class SelectProportional
extends java.lang.Object
implements IndividualSelector

Constructors

- *SelectProportional*
  `public` **SelectProportional**`( )`

Methods

- *select*
  `public Population` **select**`( net.parallaxed.bluejam.Population` **pop**`, int` **numberOfIndividuals** `)`

5.2.20 Class SelectTournament

Performs a variant of tournament selection. Unlike the classic tournament, this function is optimised to use FitnessStacked, so the tournament takes place in two rounds.
The first round is negative selection, where the weakest are assigned proportionally greater probabilities, and a roulette-wheel selection is made to knock out competitors.
The second round is positive selection, where the individuals are evaluated again, and the largest score wins.
Two are taken from each Tournament until the number of parents required for breeding are selected.
This will still work on fitness methods other than FitnessStacked.

Declaration

public class SelectTournament
extends java.lang.Object
implements IndividualSelector

Methods

- *getInstance*
  `public static final SelectTournament` **getInstance**`( )`
  - **Returns** - An instance of SelectTournament
- *round1*
  `protected void` **round1**`( java.util.ArrayList` **competitors**`, int` **firstRoundKnockouts** `)`
  - **Usage**

  - Round 1 performs a negative selection favouring the worst individuals. A total of *firstRoundKnockouts* individuals are selected and the passed competitors list is whittled down.
- **Parameters**
  - `competitors` - The competitors in the tournament
  - `firstRoundKnockouts` - The number of competitors to knock out in the first round.

- *round2*
`protected ArrayList` **round2**`( java.util.ArrayList` **competitors** `)`
  - **Usage**
    - Further compares the fitness of the individuals.
      This method is designed for FitnessStacked, but can still work on other fitness Functions - just becomes a regular TournamentSelection from here on in (fittest one wins).
  - **Parameters**
    - `competitors` - The competitors.
  - **Returns** - The winners.
  - **See Also**
    - `net.parallaxed.bluejam.evolution.FitnessStacked` (in 4.1.2, page 69)

- *select*
`public Population` **select**`( net.parallaxed.bluejam.Population` **pop**`, int` **numberOfIndividuals** `)`
  - **Usage**
    - {@inheritDoc}}
      Satisfies the implementation of IndividualSelector.
      See the class documentation for more coverage on how we execute the selection.

5.2.21 Class **TreeBreeder**

TreeBreeder is the default implementation of the Breeder interface for recombining and mutating NoteTrees, given an initial skeletal population.
Unlike other algorithms, this is not a singleton, since there may be more than one breeder available at any one time.
TODO Check working implementation of other NoteSequence representations.

Declaration

public class TreeBreeder
extends java.lang.Object
implements Breeder

Constructors

- *TreeBreeder*
  `public` **TreeBreeder**`( )`
  - **Usage**
    - Instantiates a TreeBuilder trivially.

- *TreeBreeder*
  `public` **TreeBreeder**`( int` **maxBreedCycles**`, double` **crossoverProbability** `)`
  - **Usage**
    - Instantiates a TreeBreeder with the passed parameters.
  - **Parameters**
    - `maxBreedCycles` -
    - `crossoverProbability` -

Methods

- *breed*
  `public void` **breed**`( net.parallaxed.bluejam.Population` **population** `)`
  - **Usage**
    - {@inheritDoc}

- *crossoverProbability*
  `public double` **crossoverProbability**`( )`
  - **Returns** - A value between 0 and 1 for the probability of doing crossover on the individual.

- *crossoverProbability*
  `public void` **crossoverProbability**`( double` **probability** `)`
  - **Usage**
    - NOTE: Setting this variable also sets the probability for mutation in this breeder. Sets the probability of crossover (between 0 an 1). The inverse of this value sets the probability for mutation. Setting this to a value less than 0.5 is not recommended.
  - **Parameters**
    - `probability` - A value between 0-1 (inclusive).

- *maxBreedCycles*
  `public int` **maxBreedCycles**`( )`
  - **Returns** - The maximum number of times we run breeding functions over any single or pair of individuals.

- *maxBreedCycles*
  `public void` **maxBreedCycles**`( int` **cycles** `)`
  - **Usage**

    * Sets the maximum number of times we run a breeding process for each Individual pair. Default = 5.
  - **Parameters**
    * `cycles` - The number of times to runs

- *recombine*
  `public void` **recombine**`( net.parallaxed.bluejam.NoteTree` **nt1**`,`
  `net.parallaxed.bluejam.NoteTree` **nt2** `)`
  - **Usage**
    * Performs crossover on two passed NoteTrees in-situ.
      This methods should receive two note trees that are to be crossed over. Nothing is returned, the trees are altered in-situ and should remain that way. If breeding NoteTrees through this method, always pass the clone().
  - **Parameters**
    * `nt1` - A note tree to recombine
    * `nt2` - The note tree to combine with.