**Instructions for the end-user regarding the custom function API, located in /src/customapi/**

1. ***CustomFunctions.java***

This is the primary class to be used for creating user-defined functions. This class is roughly divided into three sections:

1. Utility Functions

These functions occur under the heading “Begin utility methods”, towards the end of the above-referenced file. These are (mostly) private utility methods, which you may use as helpers in creating your custom functions. Note that utility methods can also be treated as normal custom methods (see section b below) and may be displayed in the GUI if you wish, but caution should be taken in doing so, or in modifying any utility methods for that matter, due to the following reasons:

- Some utility methods invoke methods from classes outside of this package. While invoking other classes’ methods is allowed for utility methods, it is not recommended that the end user modify classes outside of the /customapi package.

- Some utility methods are called by other classes, such as runCommand() which is invoked by the CustomFunctionsPane class, in order to execute the custom methods from the GUI interface. Modifying these utility methods would require modifying these other classes which, again, is not recommended.

If you still wish to treat a utility method as a custom method, then just make it public, move it to the custom functions section, and follow the directions in section c below to make that method visible in the GUI.

1. Custom Functions

These functions occur under the heading “Begin custom methods”. These should be public, and the end user may create / modify functions for computational compositions / experimentation. A few custom methods have been created here for demonstrative purposes. For simplicity and consistency, right now every custom function should take in zero or more parameters of type int only. Caution: when creating custom methods here, you should take care not to invoke methods from classes other than those in the package /customapi/, as doing so may result in unexpected behavior. If you MUST invoke methods from outside classes, consider writing a utility function instead and making it private.

In order to make your custom functions available for viewing and executing in the GUI, see section c below.

1. Templates

Look for “Begin custom template” section towards the top of the class. Here, you may create (mostly copy/paste) a block of code that will cause one or more of your custom functions to be displayed on the GUI. An example for a custom method called rampVolumeGivenMeasureRange() is shown below:

|  |
| --- |
| str = "rampVolumeGivenMeasureRange(int startMeasureIdx, int endMeasureIdx, int volumeStart, int volumeEnd)";  **this**.commandsStrHM.put(index++, str);  **this**.commandsHM.put(str.substring(0, str.indexOf("(")), () -> {  **int**[] params = **this**.getAndValidateIntParamArr(4);  **this**.rampVolumeGivenMeasureRange(params[0], params[1], params[2], params[3]);  }); |

The first line contains the full name and signature of your custom method. You should edit this line for your own custom function. The second line should be left alone. The third line and the block of code should be left alone, except that for the method getAndValidateIntParamArr(x), you should set x = the no. of parameters in your custom function, and the line after that should contain an invocation of your actual custom function, with the right number of parameters, i.e. (params[0], params[1], … , params[x-1]). Remember that every parameter should be of type int.

This class contains a main() function. Running this class will automatically call the graphical user interface corresponding to the class PianoRollGUI.java. Alternatively, you may also execute the PianoRollGUI class directly; there is no difference. Once you see the interface, you may open the custom user dialog under the “View” menu, or alternatively by pressing F10. The dialog looks like this:



The list view contains the name and signature of the custom methods, including parameters if any, that you have chosen to display. Clicking on one of them automatically populates the textbox located 2nd-from-bottom with the method name (You may also manually type in the method name). The textbox at the bottom should then be filled in manually with the corresponding parameter values (separated by spaces) that you wish to pass into the method. Pressing OK will execute that method. If you do not pass in the correct number of parameters, or if they’re not the right type, etc., expect an error to be thrown.

Execution of custom functions now work in real-time, even during playback. For example, before opening the custom dialog, you can start playback from the beginning of a score by pressing Ctrl-p, and as the music is playing, open the custom dialog (F10) and execute a function of your choosing as explained above. Doing so will make the changes in real-time and update the sequencer GUI, and the changes will be instantly recognizable visually and audibly as the playback continues (this assumes, of course, that the custom function is efficient enough to not cause significant computation time / lags).

1. ***NoteFeatures.java***

Intended to be accessed primarily by CustomFunctions, NoteFeatures is a class that represents a given note’s attributes, e.g. its (col, row) location in the sequencer GUI, duration, pitch value, color, volume, which midi channel the note belongs to, and so on. The class contains methods for representing the aforementioned attributes in the form of a BitSet object via the getBitSet() method, as well as in a plain String format using the getBitSetBinaryString() method. This makes it possible to experiment with manipulating notes via bit operations, e.g. XOR between two notes to produce a new note, and to create custom functions that exploit this information. (To help with this, the CustomFunctions class contains a utility method called notateFromBitSet() method, to write the note represented by a given BitSet object on to the sequencer.)

The NoteFeatures class has a number of overloaded constructors. It can be constructed via passing in either a RectangleNote object (see some utility methods in the CustomFunctions class for examples of this type of construction), or a BitSet object.

The former construction method is relatively problem-free, because the RectangleNote object parameter is expected to meet all the constraints of a valid note (e.g. its pitch value, column index, row index, etc. are within the valid range). However, the latter construction method can be problematic in the event that the BitSet parameter is either arbitrarily created, or created by experimenting with bit operations (such as XOR between two valid notes, which may output a BitSet whose values may imply one or more attribute values that are no longer valid).

To help protect against the above scenario, the constructor that takes in a BitSet object has a number of safeguards, such as performing the mod operation in the event that its row or column index would otherwise be out of valid range. These safeguards may not be ideal under all circumstances (for example, when the user would rather treat an “invalid” BitSet as a “null” note instead of constructing it as a valid note), and the end user is invited to experiment with different methods of dealing with the problems that can occur while working with bit operations.

You should exercise caution in modifying this class. It is recommended that you work with the CustomFunctions class to the extent possible in creating / modifying functions therein, and invoke the NoteFeatures class when needed.

This class contains a main() function which was used for testing some of the methods and making sure that the conversion to / from BitSet and corresponding binary String works correctly. The main() function may be deleted or commented out.

1. ***BitSetUtil.java***

This is a utility class with a number of static methods that are invoked by the NoteFeatures and CustomFunctions classes, for converting amongst BitSet, binary String, and integer representations. This makes it possible to convert to / from a note’s attribute values, which are in int or boolean format. Feel free to add new relevant static methods that you think are useful here. Exercise caution if modifying existing methods.

1. ***MyClassLoader.java / SuperCustomFunctions.java***

These classes work together to make it possible to dynamically re-load the CustomFunctions class, without having to re-start the entire PianoRollGUI interface. While you have the GUI open, you may edit the CustomFunctions class as per Sec. I, and save / build it. Once you’ve done so, you can simply close and re-open the custom user dialog, without having to restart the program. Once you do, any new / changed methods will be visible in the list view, and the deleted / commented out methods will be invisible. These classes should not be modified.

1. ***ColorEnum.java***

This class is currently unused. It was originally thought of as a way to represent color values (int values) in an easily readable enum format. Right now, however, the /pianoroll/ColorIntMap class contains thousands of different colors.