Design Proposal-Revised
6.005 Project 1
Team: ahaidar, azizkag, ishabir

Summary of Changes

- In our initial design we were thinking of the header tokens as normal tokens and that we will treat them exactly like we treat the other ones. After trying to implement them, we found that they need to be treated differently. For that we used the lookahead feature in regex and then created a new ABCHeader method in parser to get the header information and store it in the parser object. Our initial design did not pay close attention to collection the important information from the Header of the abc file.
- As a result of taking care of the header file now, we needed to toknize the information in it. So we added new tokens and its grammar to the list of tokens.
- While building the lexer it was difficult to represent the length as numerator and denominator tokens so instead we created one number token and one forwardslash token and in our parser we recognize which one is the numerator and which is the denominator.
- Added another object to handle accidentals at the beginning of Bar

Lexing

First, the lexer will go through the header of the file and create the following tokens:

- **C:** Name of the composer.
- **K:** Key.
- **L:** Default length or duration of a note.
- **M:** Meter.
- **Q:** Tempo.
- **T:** Title of the piece.
- **V:** title of voice
- **X:** Index number

Where each one of those tokens will contain the values attached to it. For example, if we get "C: Ahaidar" this will produce a token "C" with value "Ahaidar" and so on for the rest of the tokens. This is done using the lookbehind feature in regex.

As for the music part:

Tokenizer takes the string, and creates tokens. A first stage parser creates BAR objects that contain everything between bars. A second stage parser creates the voices and handles repetition. VOICE objects are then added to the sequence player via PLAYER, which adjusts based on the key signature, and everything is played. Essentially this is an i*nterpreter* way of doing it. Note that while adding things to the player we keep track of the tick that has been reached.

TOKENS:

Tokenizer will produce the following tokens:

- C("C:")
- DIGIT("[0-9]+")
- FORWARDSLASH("\\/")
- FIRSTREPEAT("(\\[1)")
- SECONDREPEAT("(\\[2)")
- BASENOTE("[A-Ga-gz]")
- OCTAVE("[\\,\\']+")

- ACCIDENTAL("[__\\^_\\=\\^^]")
- OPENBAR("\\[\\|")
- ENDBAR("(\\|\\|)|(\\|\\])")
- CHORDBEGIN("\\[")
- CHORDEND("\\]")
- TUPLET("\\([0-9]+")
- REPEATBEG("\\|:")
- REPEATEND(":\\|")
- BAR("\\|")
- COMMENT(%.*(///n|///r|///f)
- SPACE("[\\s]+")
- INVALID(“[.]*”)

Parser

Then a list of tokens is passed from the lexer to the parser. The parser is a three staged parser (appendix I) which produces the AST containing the following data structures.

DATA STRUCTURES:

Interface NOTESTRUCT
This will be implemented by NOTE, CHORD, and TUPLET

- int timeNumerator
- int timeDenominator

- int getDenom() - returns the timeDenominator (of any of its notes)

- int getNumTicks(ticksPerQuarter) : returns the number of ticks that this note (or, one of the tuplet notes) will occupy. This will only be called internally.
- int addToPlayer(startingTick, player): figures out how many ticks the note will occupy and adds it to the player. Returns the 'tick' that we are now at.

(Note: the time length will be in the form of timeNumerator / timeDenominator. For Tuplets, this will be for each individual note.)

NOTE implements NOTESTRUCT

- String basenote (a-gA-G | z (for rests))
- String accidental
- String (or int) octave (Alternative: turn basenote and octave into an int)
- int timeNumerator
- int timeDenominator

(Note: the time length will be in the form of timeNumerator / timeDenominator)
(invariant:gcd(timeDenominator,timeNumerator) = 1 to maintain simplest term)
mutable: for can change the length of the note later

CHORD implements NOTESTRUCT

- ArrayList<NOTE> notes
  mutable: for containing an exposed mutable type

TUPLET implements NOTESTRUCT

- int Value (2, 3, 4)
- ArrayList<NOTE> notes (note: the timelength of the notes will not ignore the fact that it is in a tuplet (it will not be correct)
  mutable: for containing an exposed mutable type

---

interface BARLINEOBJECT (a BAR or a bar-y token)

BARSIGNAL implement BARLINEOBJECT

- String value (or token value) (ex. |, |[, |[2, |:, etc. OR voiceChanger)
  immutable: they are staying the same throughout.

---

interface BARSTRUCT (a BAR or VOICE)

- int getMinTicksPerQuarter() : returns smallest minTicksPerQuarter of all the notes under it.
- int addToPlayer(startingTick, player):
  - calls the PLAYER with everything it represents, in order. Returns the tick it reaches after adding itself.

  immutable: they are staying the same throughout.

BAR implements BARSTRUCT, BARLINEOBJECT

- ArrayList<NOTESTRUCT> *notestructs*- this will contain the NOTESTRUCTS within that bar
  immutable: they are staying the same throughout.

VOICE implements BARSTRUCT (mutable)

- ArrayList<BARSTRUCT>
- append(BARSTRUCT)
- String name
- int startPoint : represents an index (in the ArrayList) - everything after this is in the most recent 'group'.
- hitStart(): resets startPoint to the last index (= ArrayList.length)
- repeat(): adds to the ArrayList all the bars since the hitStart
- replaceLast(BAR): replaces the last bar with the received BAR argument

  mutable: we keep adding BARSTRUCTS after we create it.

stage1Parser(ArrayList<TOKEN>);

Create an ArrayList<BARLINEOBJECT> - either BARs or BARSIGNALs everything into BARs and BARLINEOBJECT

Parser is an iterator, and will keep track of the index *i* in the list of tokens. Parser will fill in an ArrayList<BARLINEOBJECT>.

Parser will ignore all voice-related tokens. Parser will turn all bar-like tokens into BARSIGNALs. After every BARSIGNAL, it will call barConstructer, which will return a BAR (representing all objects until the next barline, and is made up of NOTE, CHORD, and TUPLET objects, created by calling noteConstructer, chordConstructer, and tupletConstructer, respectively).

BAR barConstructer(i):

- takes index i of the Array that the parser is going through, moves through until the end of the bar, and constructs and returns the bar. i is updated in the process.

- create a new ArrayList<NOTESTRUCT> and add NOTESTRUCTS to it until the next BARSIGNAL is observed. It will also create:

- an ArrayList<String> naturals: list of basenotes with naturals.
- an ArrayList<String> sharps: list of basenotes with sharps.
- an ArrayList<String> flats: list of basenotes with flats.

- If the parser encounters a tuplet_start token:
  - it will call tupletConstructer(i),
- If the parser encounters a [:
  - it will call chordConstructer(i),
- Else (a normal note is reached)

- it will call noteConstructer(i)

- The returns of these methods will be added to the ArrayList<NOTESTRUCT>.

- When a new BARSIGNAL Is encountered, the parser creates a BAR with the NOTESTRUCTS it has gathered in the ArrayList, and adds this BAR to the ArrayList of BARLINEOBJECTS.

TUPLET tupletConstructer(i);

- Takes the current index and moves through tokenArray until a space is encountered. After saving the tuplet number (2,3,or 4) it calls noteConstructer until a space is reached, and will save the returned NOTEs in an ArrayList<NOTE>, which it will then use (along with the original tuplet number) to construct the TUPLET. The TUPLET is returned. i is updated in the process.

CHORD chordConstructer(i);

- takes the current index and moves through tokenArray until a] is reached. It will call noteConstructer until a space is found, adding the returned NOTEs to an ArrayList<NOTE>. It will then construct a CHORD with this list of NOTEs and will return the CHORD. i is updated in the process.

NOTE noteConstructer(i);

- takes the current index and moves through tokenArray until a space is reached. It will make a NOTE object with the tokens (accidentals, base notes, octaves) it went through, and returns the NOTE. i is updated in the process.
- If the note does not have an accidental, its basenote is looked for in the three accidental lists (naturals, sharps, flats). If it is found in any, the appropriate accidental is manually added to it.
- If the note has an accidental, its base_note are added to the appropriate accidentals list (and removed from other accidentals lists, if it is there).

VOID HeaderInfo():

This method parses the header of the abc file and populates the header variables in the parser with the appropriate values.

Once the ArrayList of BARLINEOBJECTS is created, it is passed on to the Stage2Parser.

Stage2Parser(ArrayList<BARLINEOBJECTS>);

int repeatStartPoint = 0;

- This will constantly hold the index in the array of the last object that is either a |] or a |:

ArrayList<VOICE> voices;
ArrayList<String> voiceNames;

- voices and voiceNames must be created at the very beginning, and must either be available as global variables, or must be retrievable via some method.

REPEAT repeatConstructer(i);

The Job of Stage2Parser is twofold:

- First, it constructs and builds up the voices.
- Second, it fills up the voices with BAR objects, which it creates.

This parser is a state machine, with the state being the number(index) of the voice it is currently filling up.

Parser iterates through the list of BARLINEOBJECTS. If it encounters a voice object, it switches the currentVoice. If it encounters a |, it just adds the previous bar to voices[currentVoice] (which is a VOICE object). If it encounters a |: or |], it calls VOICE.hitStart(). If it encounters a :|, it calls VOICE.repeat() after adding the bar. (and calls hitStart())

If it counters a bar with [1, it adds the bar, calls repeat, then calls replaceLast(next Bar), calls hitStart and continues.

Parser returns the list of VOICEs.

PLAYER

This class is what receives all play commands from the notes, mutates them as dictated by the key signature (adds sharps/flats if necessary) and then adds them to the actual SequencePlayer (a global object).

OVERALL: PLAYING

We end up with everything contained in a SONG object, which contains VOICE objects, which contain BAR objects, which contain NOTE, CHORD, and TUPLET objects (with the last two themselves containing NOTE objects).

Thus, everything boils down to NOTE objects.

When getMinTicksPerQuarter is called from Voice, it runs down recusively, and effectively returns the smallest number that is a common multiple of the denominators of every single NOTE (including those in chords, tuplets). This is a number that is guaranteed to be able to support the entire song.

Playing:

Music is added to the sequence player via a single myPlayer object. This object converts octave symbols into numbers, applies sharps/flats as defined by the key signature, etc.
How are things added to myPlayer? There are two levels:
First, every bar is added, one at a time. We go down from Song until we hit BARs, and each one creates an object of BarManager. This is sent down further to all the notes, which are added to the BarManager. The BarManager processes the note with bar-specific information (ie, any accidentals have an effect onwards till end of bar), and then adds the notes to the myPlayer (which then adds to sequence player, as described above).

Finally, we play the sequence player.

Testing Strategy

Throughout the implementation of this project, we will be following the model of test-first programming. So as the project grows, all the little methods and classes should be tested. Now the abc player has three major components that we need to check thoroughly for any bugs, and those are: the lexer, parser, player.

Lexer:

- Test of basic functionality
- Testing invalid input (this would require multiple tests)
- Testing if it tokenizes the spaces between note length correctly
- Taking a list of notes with no spaces and being able to tokenize the notes correctly

Parser:

- parsing multiple voices and being able to assign them correctly
- parse multiple accidentals
- recognize the key signature and differentiates it from normal accidentals.

Player:

- Notes with accidentals
- Notes with octaves
- Chords
- Tuplet with different length notes

We also have tests for chords, tuplets and notes.

And for an overall system testing we are planning on writing three abc files and playing them with our program and also borrow other (more complex) abc files from the internet and make sure they sound like they are supposed to.

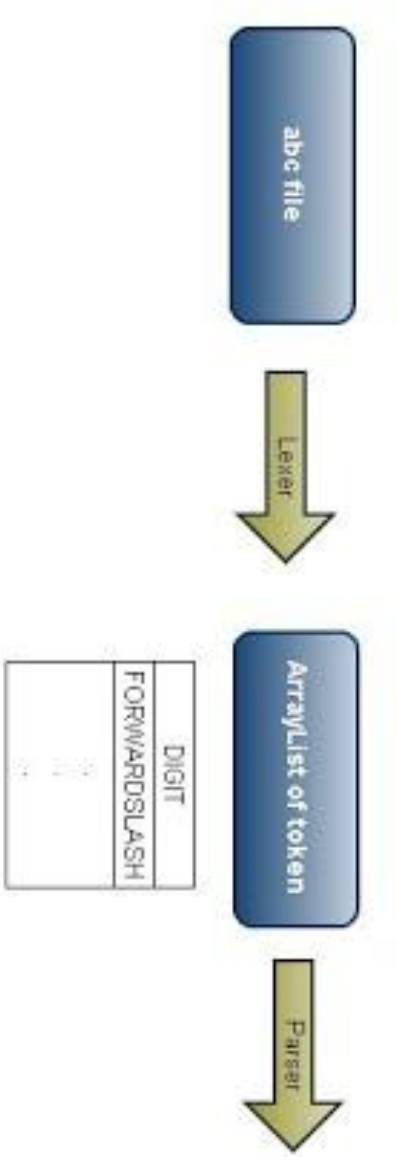
Additional files:

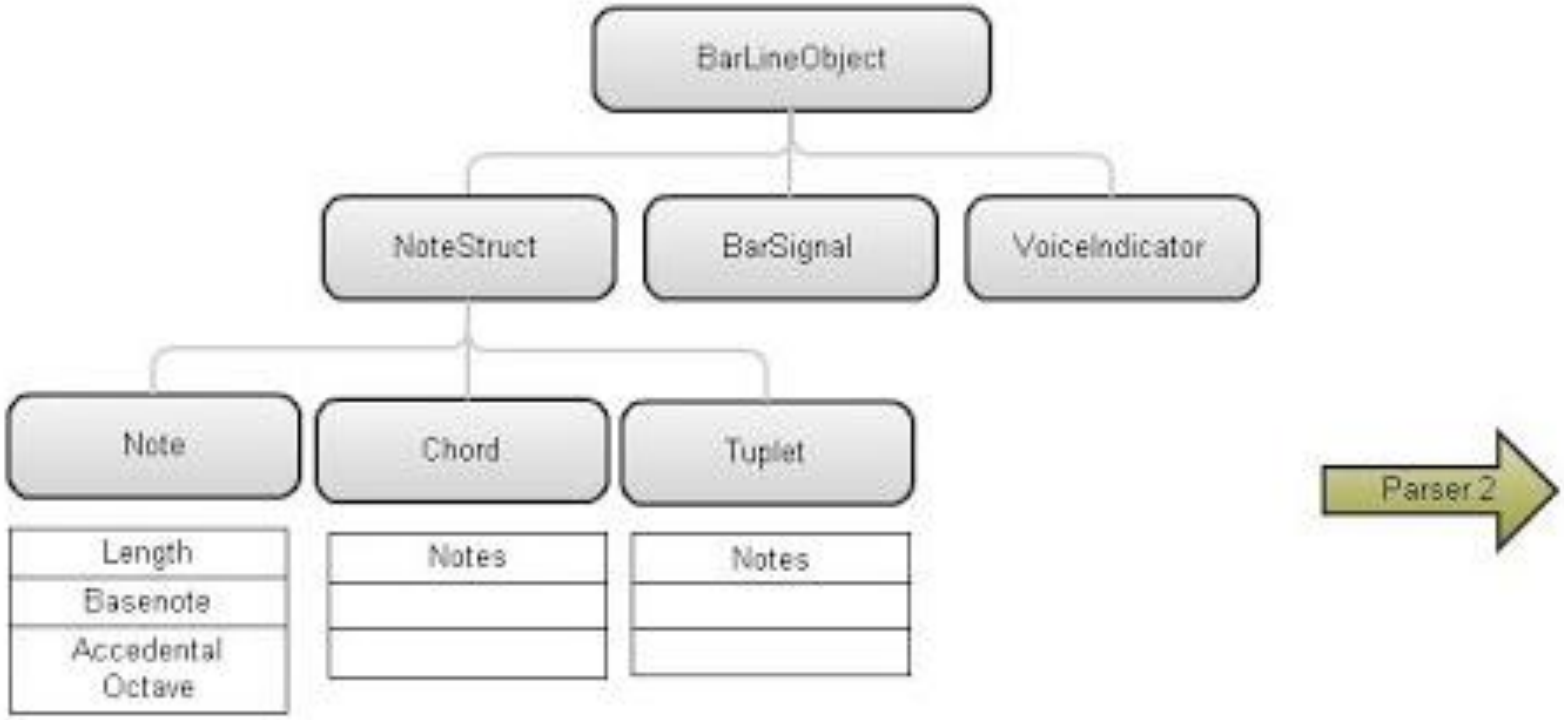
pluginbaby.abc: The intro riff and chorus of the popular song “Plug In Baby” by Muse, which we manually created. This is a general test, and was mainly done to prove to ourselves that we built something cool (fur_elise and other things didn’t quite do the magic). Some of the functionalities included are basic repeats, multiple voices, etc. We used notes that were very low (to simulate the bass guitar) and high (electric guitar), so the octaves were tested. It was also a special key signature (not C), so that was tested as well.

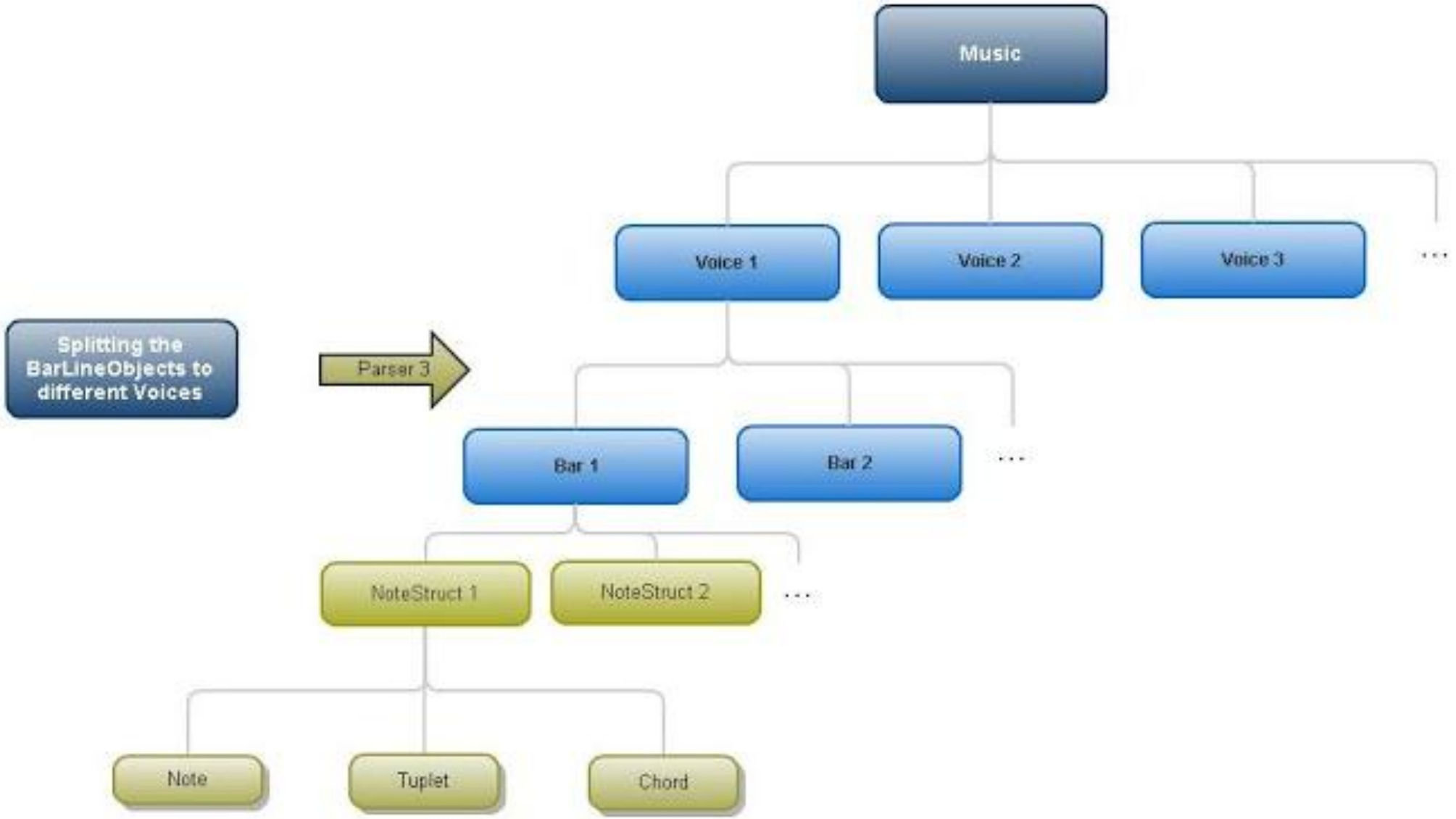
Next, we built azizlullaby.abc, which tests non-standard L (3/4), 2-stage repeats, as well as all tuplets (2,3,4). Pretty much, testing the nasty stuff.

Last, we built hastalavista.abc to test our catching of incomplete bars. Also tests commented lines.

Appendix I
ADT




BarLineObject
NoteStruct
BarSignal
VoiceIndicator
Note
Chord
Tuplet
Length
Basenote
Accedental
Octave
Notes
Notes
Parser 2


Music
Voice 1
Voice 2
Voice 3
...
Splitting the BarLineObjects to different Voices
Parser 3
Bar 1
Bar 2
...
NoteStruct 1
NoteStruct 2
...
Note
Tuplet
Chord

Appendix II

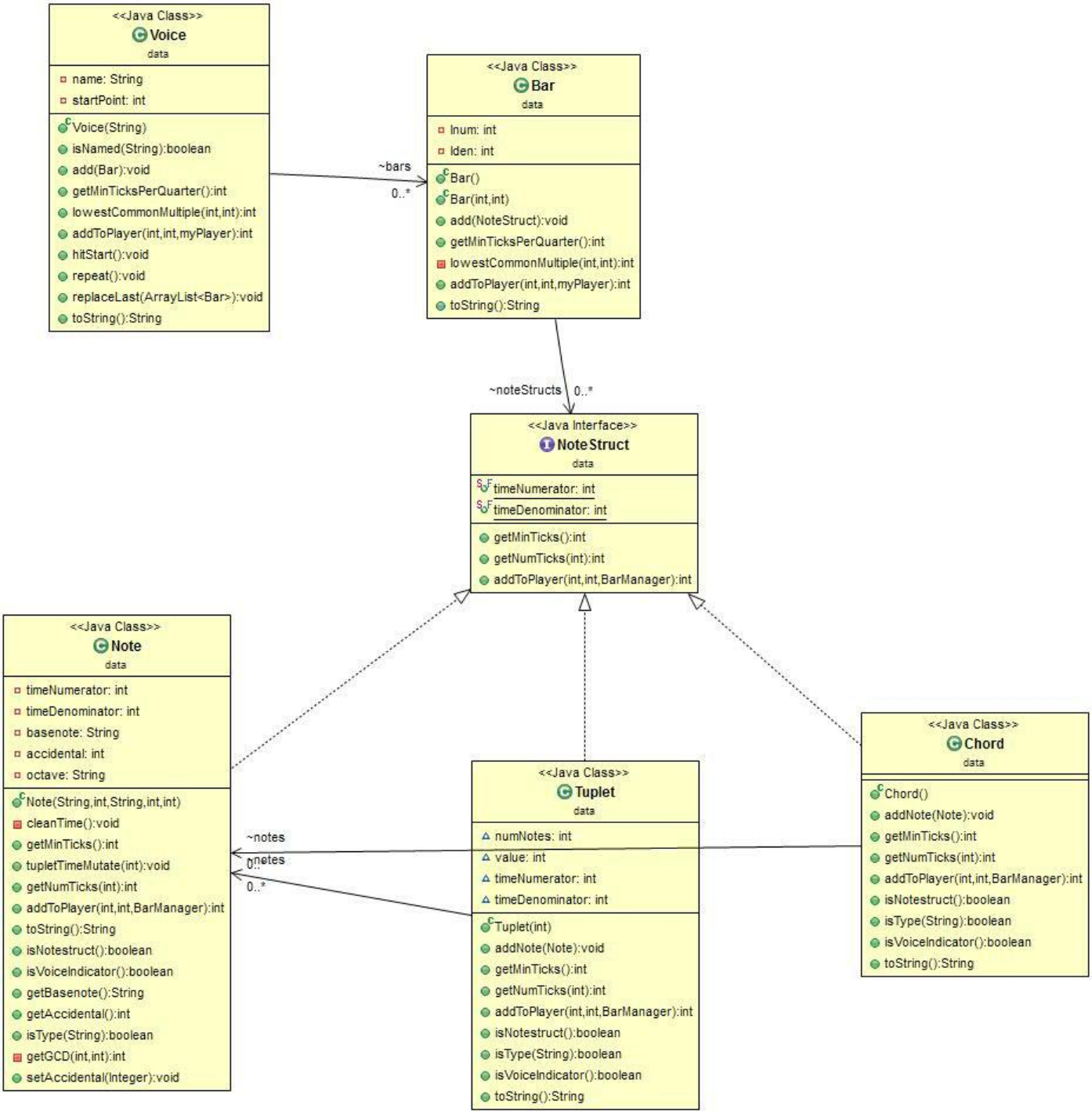


Figure 2: Constants and methods of the ADT