Problem Set 1

Programming Music

Out: 27 August
Due: **Monday, 1 Sept**
(beginning of class)

Collaboration Policy - Read Carefully

For this problem set, you should work alone, but feel free to ask other students for help and offer help to other students. It is a violation of class pledge policy to use answers from previous years.

Purpose

- Learn basics of programming in Java
- Get familiar with the Eclipse integrated development environment (IDE)
- Provide some exposure to Java class interactions and object oriented design

Background

Part of the richness of Java is the large number of libraries and APIs that are available for free. In many ways the utility of Java is defined more by the extras that are available for it than the core language itself. One API that is fun to play with is the music generator JFugue. Its homepage can be found at: http://www.jfugue.org/index.html. JFugue is open source software, distributed under the LGPL license http://www.gnu.org/copyleft/lesser.html.

A nice feature of JFugue is that it allows a user to submit a pattern – a string – with a fairly simple syntax that is then converted to music. JFugue's features are fairly extensive, as you'll discover as you play with the code.

In this assignment you will be installing JFugue, exploring it, and then modifying it, all in the Eclipse integrated development environment.

Running jfugue

The following steps will create an Eclipse project containing source code for Problem Set 1:

1. Open Eclipse by clicking on `Start | SEAS | Java Apps | Eclipse`
2. Inside the Eclipse window, click on `File | New | Java Project`. The *Create a Java Project* dialog box will appear.
3. In the `Project name:` box enter, `jfugue`. In the `Project Layout` pane, select `Create separate folders for sources and class files.` Click `Next>` and then `Finish` on the next dialog box.

Now, you should see the *Package Explorer* view, containing the `jfugue` folder.

4. Click on + next to `jfugue` in the package explorer

You should see `src` and `JRE system library` in the `jfugue` directory

5. Right click on `src` and select `NEW | PACKAGE`
6. Name your package `org.jfugue`

7. Create a `J:\CS205` directory
8. Open a browser and navigate to: http://www.jfugue.org/index.html
9. Select `download` from the list on the left (Explore a little first)
10. Download `jfugue-4.0.2-src.zip` into `J:\CS205`.
11. Unzip `jfugue-4.0.2-src.zip` in place --a directory named `org` is created
12. Drag `org` and drop it on the `src` directory for `jfugue` in the package explorer.

You should see two directories (packages in this case) created in `src` :

```
org.jfugue
org.jfugue.extras
```

13. In your browser, navigate to http://www.xom.nu/
14. Download the third item listed, `Jar file`, (Core packages only) into `J:\CS205`. This action will create the file `xom-1.1.jar`
15. In the Eclipse package explorer, right click on project `jfugue`
16. Select `properties`
17. Select `Java Build Path` from the list in the left hand column.
18. Click on the tab `libraries`
19. Select the `Add External JARs` button
20. Navigate to `J:\CS205` and select `xom-1.1.jar`
21. Select `OPEN` in the file selection dialog, and then select `OK.` All of the little red x's in the `jfugue` directory should go away.
22. Click on the + to the left of the `src` directory for your project `jfugue` in the package explorer
23. Right click on the `org.jfugue` package directory within the src directory
24. Select `NEW | CLASS`
25. Name your class `MyMusicAPP` and select `FINISH`
26. In the editor window for `MyMusicAPP` enter the following text just after the line:
    `public class` `MyMusicApp {`

```
        public static void main(String[] args)
        {
              Player player = new Player();
              Pattern pattern = new Pattern("C D E F G A B C6w");
              player.play(pattern);
        }
```

27. Select `FILE | SAVE`

Whenever you make a change to a Java source file and save it, the changes are automatically compiled. If you want to force Eclipse to rebuild the entire project, select `Project | Clean`.

28. Select `RUN | OPEN RUN DIALOG...`
29. `Project` should be named `jfugue` and `Main class` should be named `org.jfugue.MyMusicAPP`
30. Plug in your headphones and select `RUN`

You should hear the eight notes listed in the player pattern (scale).

The jfugue music programmer

The implementation of the JFugue music programmer contains modules for parsing and interpreting input strings (e.g. "`C D E F G A B C6w`" in the example above), checking them for errors, setting up timing and parallel notes (e.g. chords) and sending them off to "parser listeners" which can turn them into music, create sheet music, create a graphical display, etc. Currently, the program only plays music. Here we provide a description of some of the more important classes. At this time you do not need to understand the details of how these classes are implemented.

`Player.java`

This is the top level class for the implementation of JFugue. It receives user specified input patterns denoting a musical sequence and sends them off to a parser and interpreter.

The author of JFugue, David Koelle, has published a book, "The Complete Guide to JFugue". Excerpts from the first two chapters can be found here: http://www.jfugue.org/book.html Take a look at the excerpts from chapter 2, "Using the JFugue Music String." There you will find a description of the grammar (rules for making legal strings) for patterns, and an overview of all of the actions that can be expressed in patterns.

`Parser.java`

Processing of input patterns takes place in the parser class hierarchy. The parser class is the root of this hierarchy. It defines a base set of actions (methods) that can be applied no matter what patterns are parsed to do (musical output, music notes, graphics, etc.). Such actions include management of debug tracing, setting up listeners to the parser's output, and sending inputs to the listeners.

`MidiParser.java, MusicStringParser.java and MusicXmParser.java`

Using object-oriented methodology, which we'll discuss in detail this semester, JFugue creates extensions ("extends") the parser class in multiple directions. You'll find the MidiParser, MusicStringParser and MusicXmParser classes all extend the parser class, each providing specific capabilities for the functions suggested by their names. Our focus will be on MusicStringParser.

```
public final class MusicStringParser extends Parser
```

Open the MusicStringParser class in the Eclipse editor and browse through it. Having read the excerpts from chapter 2 of The Complete Guide to JFugue, you should be seeing some code that makes sense, in the context of the chapter 2 discussions. For example, this method:

```
private void parseToken(String s) throws JFugueException
{    {
        // If there are any spaces, get out
        if (s.indexOf(" ") != -1) {
                  throw new
      JFugueException(JFugueException.PARSER_SPACES_EXC,s,s);
        }
        s = s.toUpperCase();
        trace("--------Processing Token: ",s);

        switch(s.charAt(0))
        {
            case 'V' : parseVoiceElement(s);          break;
            case 'T' : parseTempoElement(s);          break;
            case 'I' : parseInstrumentElement(s);     break;
            case 'L' : parseLayerElement(s);          break;
            case 'K' : parseKeySignatureElement(s);   break;
            case 'X' : parseControllerElement(s);     break;
            case '@' : parseTimeElement(s);           break;
            case '*' : parsePolyPressureElement(s);   break;
            case '+' : parseChannelPressureElement(s); break;
            case '&' : parsePitchBendElement(s);      break;
            case '|' : parseMeasureElement(s);        break;
            case '$' : parseDictionaryElement(s);     break;
            case 'A' :
            case 'B' :
            case 'C' :
            case 'D' :
            case 'E' :
            case 'F' :
            case 'G' :
            case 'R' :
            case '[' : parseNoteElement(s); break;
            default  : break;  // Unknown characters are okay
        }
    }
```

receives tokens one at a time, checks them for simple error conditions (e.g. embedded spaces) and then calls the right method for processing depending on the token value. You can see that the eight notes, A-G, (as well as R and [ ) result in a call to "parseNoteElement". The `parse-X-Element` methods all lead to calls to a "fire event" which results in a parser listener being told about the musical event. A parser listener that is a musical note generator will cause you to hear a note through the computer speakers (or headphones).

Note, at the end of the `switch` statement, that if the user has placed an unknown character at the beginning of a token, the music parser ignores it without warning the user. Note also that Java does have a mechanism for reporting exceptions, as can be seen at the beginning of `parseToken.`

The parseNoteElement method is one of the more interesting ones because notes can be fairly complex. `parseNoteElement` contains a fairly consolidated view of all of the possibilities for a note:

```
    private void parseNoteElement(String s) throws JFugueException
    {
        NoteContext context = new NoteContext();

        while (context.existAnotherNote) {
            int index = 0;
            int slen = s.length();
            index = parseNoteRoot(s, slen, index, context);
            index = parseNoteOctave(s, slen, index, context);
            index = parseNoteChord(s, slen, index, context);
            computeNoteValue(context);
            index = parseNoteChordInversion(s, slen, index, context);
            index = parseNoteDuration(s, slen, index, context);
            index = parseNoteVelocity(s, slen, index, context);
            s = parseNoteConnector(s, slen, index, context);
            fireNoteEvents(context);
        }
    }

    class NoteContext
    {
        boolean isRest                    = false;
        boolean isNumericNote             = false;
        boolean isChord                   = false;
        boolean isFirstNote               = true;
        boolean isNatural                 = false;
        boolean existAnotherNote          = true;
        boolean anotherNoteIsSequential = false;
        boolean isStartOfTie              = false;
        boolean isEndOfTie                = false;
        byte[] halfsteps                  = new byte[5];
        byte numHalfsteps                 = 0;
        byte noteNumber                   = 0;
        int octaveNumber                  = 0;
        double decimalDuration            = 0.0;
        long duration                     = 0L;
        byte attackVelocity               = Note.DEFAULT_VELOCITY;
        byte decayVelocity                = Note.DEFAULT_VELOCITY;

        public NoteContext() {
            for (int i=0; i < 5; i++) {
                halfsteps[i] = 0;
            }
        }
    }
```

You can see from the NoteContext class that notes can be rests, they can be specified numerically or by letter, they may appear alone, or as a part of a sequence (where the first note has distinct characteristics), velocity can be specified, etc.

Playing with Notes

Users may want to know when they've specified an input pattern that contains illegal characters. As noted above, `MusicStringParser` does not report them.

1. Using the `MusicStringParser` code as a guiding example, change the last test in the switch statement in `parseToken` so that an exception is thrown if an unknown character is encountered. You can see from the exception raised earlier in `parseToken` that you need to throw a new `JFugueException` with appropriate parameters. In keeping with the way the rest of the exceptions in the program are defined, you should add a definition for your new exception to the set of definitions in the class `JFugueException`.

2. Modify the switch statement in `parseToken` to admit a colon as the first character in a token. The meaning of a colon is to repeat the action specified in the remainder of the token. That is, the remainder of the token should be performed twice.

3. For additional *optional* challenge, allow the addition of a colon followed by a number at the beginning of a token, and interpret the pair as a requirement to repeat the token <number> times. Be careful, because notes can be expressed using numbers also. That creates an ambiguity: Is a number near the beginning of a token a note, or a multiplier for the repeat operation (colon).

Turn-in Checklist: You should turn in your answers on paper at the beginning of class on Monday, 1 Sept. For question 1, turn in your parseToken.java code. For question 2, describe the changes you made and turn in all the code you changed. If you decide to try #3, then again, just turn in a written description of changes, and the code you actually changed.