# The Complete Guide to **JFugue**

## Programming Music in Java™

*Second Edition*
*Updated for JFugue 5*

# The Complete Guide to JFugue

## Programming Music in Java™

Second Edition

# The Complete Guide to JFugue: Programming Music in Java™

Second Edition.
*Book Version 2.0.0 – April 2, 2016*



http://www.jfugue.org



## About the Cover Image

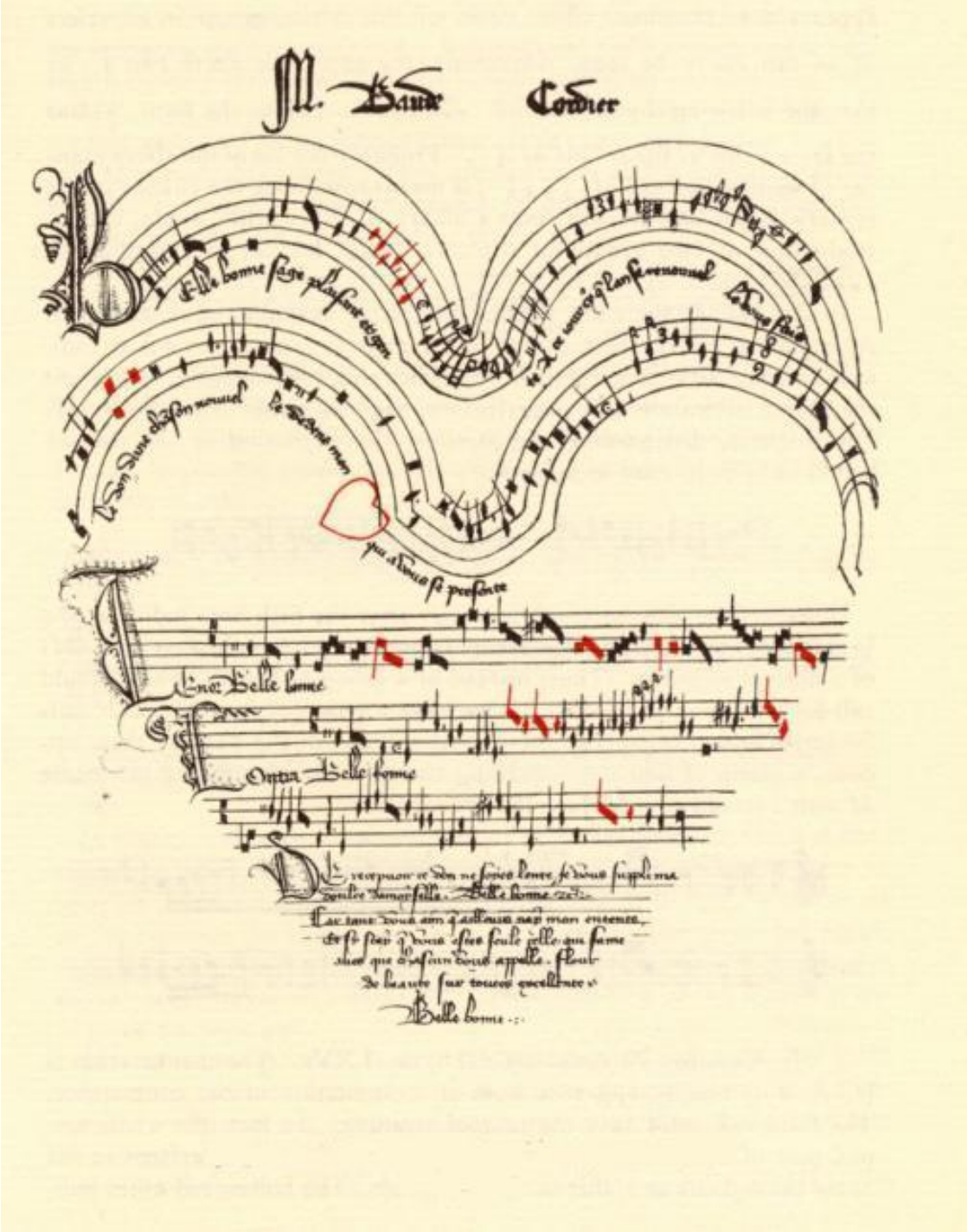
*Derivative work on Abraham Bosse's "Hearing" (original ca. 1630)*

The image used as source material for the cover of this book is in the public domain because its copyright has expired in the United States and those countries with a copyright term of no more than the life of the author plus 100 years. According to *Bridgeman Art Library v. Corel Corp.*, 36 F. Supp. 2d 191, photographic reproductions of visual works in the public domain are not copyrightable because the reproductions involve no originality. According to US Copyright Law, work in the public domain may be used for a derivative work. The image used as source material for the cover of this book is the full-resolution image linked from http://en.wikipedia.org/wiki/Image:ABosseHearing.jpg and labeled as being in the public domain, retrieved on April 4, 2008.

***For Joshua Gooding (skavookie)***

*A brilliant mind*
*whose final wishes*
*included contributing to*
*this humble project.*

*http://skavookie.blogspot.com/*




Score of “Belle, bonne, sage” by Baude Cordier. From *The Chantilly Manuscript*. The piece uses the red color to show notes that have a different duration than the corresponding notes in black.

# *Table of Contents*

# *Tables & Figures*

**Tables**

**Figures**

# *Forward to the Second Edition*

Thank you for your interest in JFugue, and thank you for supporting the development of JFugue by purchasing this book. Through your generosity, you have helped encourage the further development of a freely-available open-source software library that so many people have found useful, easy, and enjoyable.

I first became interested in programming music when, as a young and aspiring software developer, I discovered the musical capabilities of my Commodore® 128. I particularly enjoyed the Basic 7.0 "play" command, which let me program music in an easy and straightforward manner. I have always missed the ability to program music so easily using more recent programming languages. Eventually, I decided to create my own library for sharing that joy that I had in my youth. JFugue makes it easy for developers to express their musical ideas.

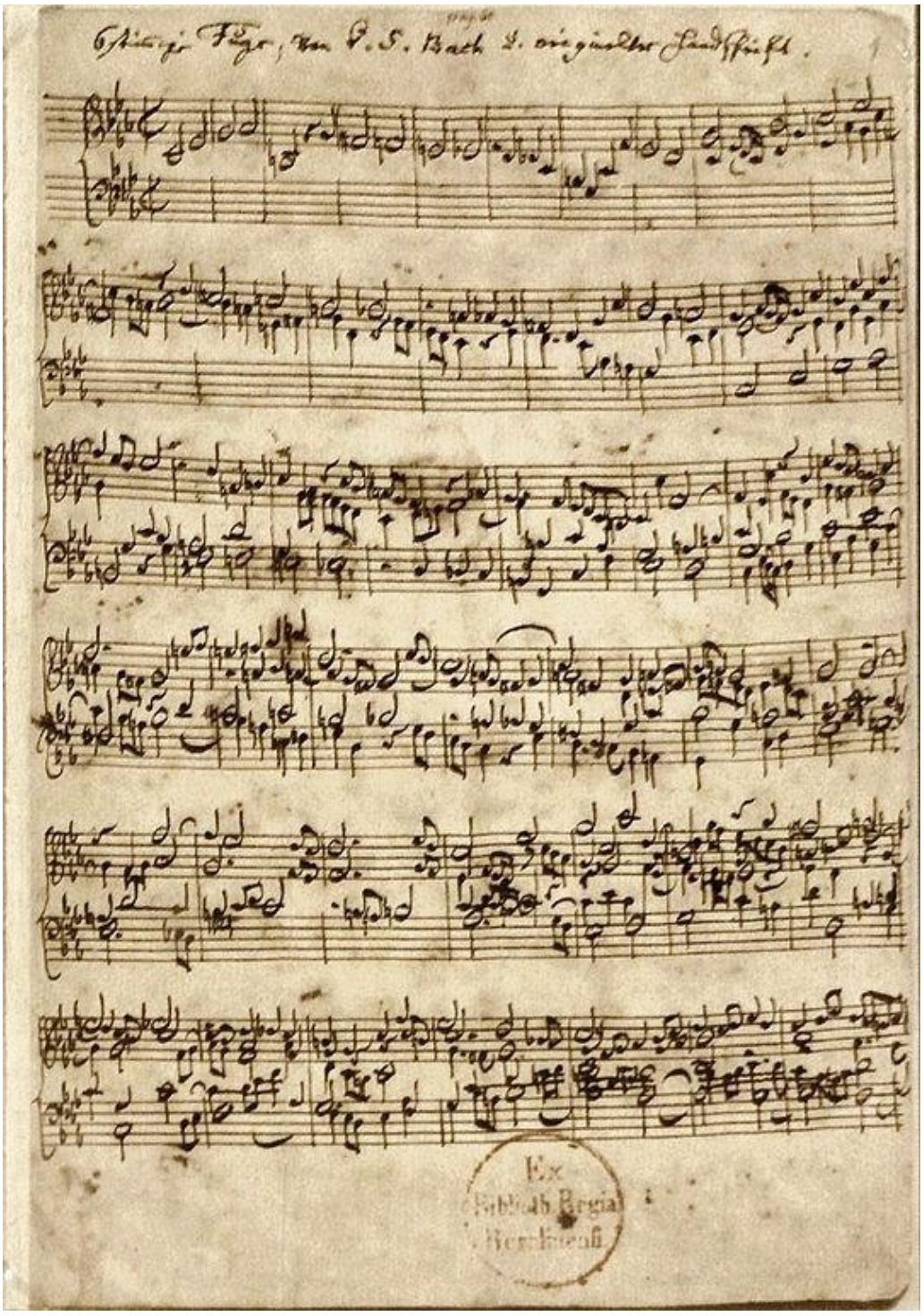
While fixing issues in the fourth major version of JFugue, I realized that the code needed a rebirth. When I first wrote JFugue, I was a mid-level developer and an inexperienced music theorist. Since then, I had grown immensely in my profession as a developer, and I also vastly increased by knowledge of music theory. I was also dogged by some insidious timing bugs that should not have been as hard to fix as they were. I re-wrote the code from the ground up. I also took the opportunity to introduce new ideas, such as the use of intervals and the development of a fluent API

(where methods that would normally return void instead return `this` so you can chain commands together: `new ChordProgression().setRoot().getNotes();`). There were also many ideas from the user community that I wanted to incorporate, such as new capabilities in the Music String and new ways to think about parsers. After re-writing the code, the central core of this version of JFugue contains fewer classes and slightly over half of the original code size (when measured in kilobytes of Java files) and has made it easy to introduce new capabilities (e.g., intervals, chord progressions, customizable chords) that work seamlessly together. The new version of JFugue provides more correct functionality (e.g., converting MIDI files to JFugue, real-time parsers) and introduces an independent music specification language based on the JFugue Music Strings called Staccato.

It is my hope that JFugue will inspire other developers and encourage people of all ages to experiment with the expressive power of music. There are so many interesting musical things than one can create by using programs to generate musical data interactively or algorithmically. I love seeing what people invent using JFugue to help stretch the bounds of imagination!

# Part 1

## A Tour of JFugue



The *Ricecar a 6*, a six-part fugue from Johann Sebastian Bach's "The Musical Offering" (BWV 1079), written in Bach's hand.

# 1.1

## *Let's Make Music!*

Welcome to JFugue! Let's jump right into the code and start making music. Part 1 of this guide will introduce you to several examples of programming music with JFugue. Later parts of this guide provide in-depth details about the library. If you need help setting up your development environment, please see Appendix: Setting Up JFugue.

### Introducing "Hello, World," JFugue Style

Let's start by writing one of the simplest JFugue programs possible. The following example plays a scale of notes (`C` through `B`).

```
import org.jfugue.player.Player;

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

Turn on your speakers or put on your headphones, run the program, and let the music play!

The Player class is responsible for playing music that you enter as strings. Those strings are written in a format called *Staccato*, which had its origins in the JFugue project but is now an open standard and may be used by any music software. Staccato is designed to be easy for people to write and read. There are other music formats, such as MusicXML and ABC Notation, some of which focus on detailed, high-fidelity representation of music notation, but Staccato is geared *specifically* towards ease of writing and reading by everyday people.

One of the ways that Staccato accomplishes this is by making several common-sense assumptions. For example, you may notice that you did not indicate a duration or octave for any of the notes, nor a tempo for the music. Staccato used default values (fourth octave, quarter-note duration, 120 beats per minute) that make it easy for you to play music without being overwhelmed with the details. Of course, when you are ready to delve into the details, there are plenty of details to learn; yes, you can set your own durations, octaves, or tempo. There are also some fun and powerful things you can do in Staccato, but let's get the basics down first.

This next example is a step up from "Hello, World." This example plays the first few notes of one musical line from the crab canon in Johann Sebastian Bach's "The Musical Offering" (BWV 1079). In this example, the numbers you see (`5` and `6`) are octaves, `h` and `q` represent half and quarter durations, and you can probably figure out which notes are flats and sharps. More details about octaves, durations, and accidentals are provided in Section 2.2, Notes, Rests, and Chords.

```
player.play("D5h E5h A5h Bb5h C#5h Rq A5q A5q Ab5h G5q G5q");
```

Suppose you want to play your notes in an instrument besides the piano. Since this is a crab canon, there is a second musical line that is the reverse of the first, and the two lines are played in harmony. In the following example, you can see the use of voices (`V`) (also known as tracks or channels) and instruments (`I`), and now some eighth notes (`i` next to a note, since 'e' is already used as a note). Let's also add some bar lines (`|`) to make the music easier to read, and slow down the tempo (`T`) from the default of 120 BPM.

```
player.play("T100 V0 I[Violin] D5h E5h | A5h Bb5h | " +
            "C#5h Rq A5q | A5q Ab5h | G5q G5q" +
            "V1 I[Flute] D5q F5q A5q D6q | " +
            "C#6i D6i E6i F6i G6i F6i E6i D6i | " +
            "E6i A5i E6i G6i F6i E6i D6i C#6i | " +
            "B5i C#6i D6i F6i E6i D6i C#6i B5i | A5i Bb5i");
```

There are other things you can do with notes—ties, triplets, dynamics (on/off velocities), melodies and harmonies—which are all described in detail in Section 2.2, Notes, Rests, and Chords.

## Using Chords and Chord Progressions

It is easy to play chords with JFugue.

```
player.play("C3MAJq F3MAJq G3MAJq C3MAJq");
```

And, of course, chords can also be played in different voices, with different instruments, at different octaves, and for different durations, just as notes can.

Suppose you know something about music theory and would really like to play a I-IV-V-I chord progression, which is what the above example actually is, but you would rather think of it as I-IV-V-I. Meet the ChordProgression class:

```
Player player = new Player();
ChordProgression cp = new ChordProgression("I IV V I");
player.play(cp);
```

This plays the same thing!

In the example above, `"I IV V I"` is not a Staccato string; instead, it is a string passed to the ChordProgression constructor. You can get the Staccato representation of this class by using the following method:

```
Pattern pattern = cp.getPattern();
```

This returns the Staccato representation `"C3MAJ F3MAJ G3MAJ C3MAJ"` – four chords using the default C in the 3rd octave as the root.

Chords can also be played with different root notes. The following two lines play the same music – a I-IV-V-I progression with a root of B-flat.

```
1. player.play(cp.setKey("Bb")); // using cp from above

2. player.play("Bb3MAJq Eb4MAJq F4MAJq Bb3MAJq");
```

## Creating Rhythms

Are you ready to really get down and groove? JFugue makes it easy to create rhythms. In this example, each of the letters is associated with a percussion sound – drums, cymbals, hand claps, and so on.

```
Rhythm rhythm = new Rhythm()
     .addLayer("O..oO...O..oOO..")
     .addLayer("..S...S...S...S.")
     .addLayer("`````````````````")
```

```
    .addLayer("...............+");
Player player = new Player();
player.play(rhythm.getPattern().repeat(4));
```

This seven-line program is probably the coolest piece of code you'll write today – have a listen! And you *could* actually write this as one long line of code, if you were really ambitious:

```
new Player().play(new Rhythm("O..oO...O..oOO..",
  "..S...S...S...S.","````````````````","...............+")
  .getPattern().repeat(4));
```

Most things in JFugue that produce music implement `PatternProducer`, and the `play()` method can play music from any `PatternProducer`. Let's see this in action by combining this rhythm with a chord progression. Use the seven-line rhythm example, but remove the `player.play` line and replace it with the following two lines:

```
ChordProgression cp = new ChordProgression("I IV V I");
player.play(cp.setKey("AbMIN"), rhythm.getPattern().repeat(4));
```

## Understanding Patterns

As you worked through these examples, you started to see uses of the `Pattern` class, which is essentially a wrapper around a Staccato string. Patterns serve as one of the foundational objects in JFugue, and most classes that generate musical instructions implement `PatternProducer` and override the `getPattern()` method. The `Player.play()` method takes one or more implementations of `PatternProducer` and asks each `PatternProducer` for its pattern.

`Pattern`s can be split into `Token`s, and you can write programs that examine or recombine tokens in musically interesting ways. For example, you may be adventurous enough to create an algorithm that will create a Markov model of notes read in from a MIDI file, or create a way to reverse a `Pattern` while maintaining the correct Voice and Instrument settings for the reversed notes.

# 1.2

## *That's No Moon… That's an Awesome Music API!*

In the previous section, you saw how easy it is to create and manipulate music. Now, witness the power of this fully armed and operational music API.

Let's pick up right where we left off in Part 1.1

### Loading MIDI Files

Suppose you have a MIDI file for your favorite song. You can convert it into Staccato quite easily:

```
Pattern pattern = MidiFileManager.loadPatternFromMidi(new File
("my_favorite.mid");
```

You can now print the Pattern to see the notes, or pass the Pattern to `player.play()`. And now knowing that you can transform Patterns, you can have a lot of fun with your favorite music. What would your favorite song sound like backwards? Or with a portion of the song extracted and used as the theme in a fugue?

## Converting from One Music Format to Another

You have a MIDI file for your favorite song and you need to convert that file to MusicXML. No problem:

```
MidiParser parser = new MidiParser();
MusicXmlParserListener listener = new MusicXmlParserListener();
parser.addParserListener(listener);
parser.parse(MidiSystem.getSequence(new File(filename)));
Element musicXmlElement = listener.getRootElement();
```

Done.

JFugue uses a simple yet powerful Parser/Listener paradigm that allows any `ParserListener` (in the past, also called a Renderer) to listen to musical events sent from any `Parser`. You can easily create your own instances of `Parser` and `ParserListener` and use them with the existing instances of `Parser` and `ParserListener` in JFugue. This means that you can*:

- Convert MIDI to a light show
- Take a song in ABC Notation and turn it into LilyPond sheet music
- Write a parser that converts spam email messages and turns your electronic trash bin into beautiful musical odes (some imagination required)
- Convert DNA sequences to MIDI

The possibilities are endless. Have fun and create great stuff!

(* - Each example includes at least one parser or parser listener that is left as an exercise for you, dear reader.)

## Developing Musical Tools

The same Parser/Listener paradigm that makes it easy to convert between two musical formats can also be used to create musical analysis and manipulation tools with little effort.

For example, suppose you have a piece of music (in any format) and you want to know all of the instruments that are used in the song. To do this, create a `ParserListener` that does something when instrument events are parsed. (You can do this by extending `ParserListenerAdapter`, which provides empty implementations for all of the events that a `ParserListener` receives.)

```
public class InstrumentListener extends ParserListenerAdapter {
      private List<String> instruments = new ArrayList<String>();

      @Override
      public void onInstrumentParsed(byte instrument) {

      instruments.add(MidiDictionary.INSTRUMENT_BYTE_TO_STRING
           .get(instrument));
      }

      public List<String> getInstruments() {
            return this.instruments;
      }
}
```

Now you can use this with any parser (e.g., MIDI, Staccato, MusicXML):

```
MidiParser parser = new MidiParser();
InstrumentListener listener = new InstrumentListener();
parser.addParserListener(listener);
parser.parse(MidiSystem.getSequence(new File(filename)));
List<String> instruments = listener.getInstruments();
```

In only a few lines of code, you now have the complete capability to find all of the instruments used in a musical piece. You can image using the same technique to count how many times a musician uses the same three-note sequences. Or, you could create a tool to calculate the Parsons Code for a piece of music to index its melodic contour.

## Measuring and Transforming Patterns

Recall that a pattern is JFugue's way of representing a fundamental unit of musical information. Patterns are written in Staccato. You can measure and transform patterns in two ways. The first is to use a `ParserListener`, similar to the previous example that returns a list of instruments on any piece of music. If you have a `Pattern`, you could pass the `InstrumentListener` directly to `pattern.measure()` and get your answer easily. If you have multiple changes that you want to make to a pattern – change all Piano instruments to Flute, change all half notes to whole notes, and more – you can either put those all into a single `ParserListener`, or you can chain `ParserListener`s together, which is nice because it ensures that each `ParserListener` is responsible for only one type of change.

The second way is to split the pattern into tokens. Each note, instrument change, MIDI command, and so on is represented by a single token, and tokens know what kind of musical instruction they represent. Here is a second way of finding all of the instruments in a pattern:

```
Pattern pattern = new Pattern("V0 I[Piano] A B V1 I[Flute] D E");
List<Token> tokens = pattern.getTokens();
for (Token token : tokens) {
    if (token.getType() == TokenType.INSTRUMENT) {
        System.out.println(token);
    }
}
```

## Microtones, ReplacementMaps, and Functions

Music is not limited to the 12-tone octave of Western tradition. There are "notes between the notes" that are used by music from non-Western cultures. Carnatic music is an excellent example. JFugue lets you express microtones quite easily. As long as you know the frequency, in Hertz, of the tone you would like to generate, you can use Staccato's 'm' command followed by the frequency, then followed by any other markup you would use with a traditional note (e.g., duration and note dynamics). For example, `m440.0w` would be a whole-duration standard Western 'A' note. Behind the scenes, JFugue converts the microtone to a series of pitch wheel and note events that are necessary for MIDI to render the correct sound.

But if you're interested in something like Carnatic music, you probably don't want to write your music as a bunch of frequencies. Instead of `m275.6220`, you would rather think of this as `R1`. JFugue has you covered. Using a `ReplacementMap`, you can tell JFugue how *you* would like to specify your music, and what Staccato string should be used to represent each of the things you'd like to say. JFugue even comes with a `CarnaticReplacementMap` that takes care of `S`, `R1`, `R2`, `P`, `D1`, and so on. If also comes with a SolfegeReplacementMap that lets you specify music as `DO`, `RE`, `MI` instead of `C`, `D`, `E` – which is great for children who are learning music.

Now what if your music has certain effects – trills, tremolos, slides – do you have to manually enter each note into your Staccato string? Not quite! JFugue lets you specify a function directly in the Staccato string, so you can say something like `"A B C :TRILL(Dq) E F"`. You'll need to define the function (actually, `TrillFunction` is provided in JFugue), but your Staccato string remains nicely readable.

You will find a lot of cool features in JFugue that have been the result of a lot of thinking about how to make music programming as easy and as enjoyable as possible. Enjoy this fully armed and operational API!

# 1.3

## *The Magic of JFugue*

Now that you have a taste for JFugue's capabilities, I would like to take a step back and reflect on *The Magic of JFugue* – what it is about JFugue makes it fun and exciting to program music. In particular, there are four capabilities that I believe are unique to JFugue and that deliver a quintessentially different music programming library than other packages I have seen. I offer these thoughts in the hope that they inspire your own exploration of computational magic through music – and musical magic through computation.

The first and most obvious bit of magic is ease of use – specifically, a combination of Staccato strings to represent music in a human-readable format, and the simplicity of writing a musical program. The ability to say `player.play("C D E F G A B");` and just have that *work* is extremely satisfying and extremely simple. Programming music could not get any simpler than this. And the fact that this extends to let you say `player.play("T[Adagio] V0 I[Piano] CmajW V1 I[Flute] C5q E5q C6q G5q");` – a bit more complex, but still simple – is also impressive.

The second piece of magic is that JFugue calls musical things Patterns, and anything in JFugue (or your own code!) that can produce playable music is a `PatternProducer`, and that Patterns can be manipulated in

music-centric ways that Java Strings cannot. This ensures that any code that generates JFugue music can do so in a slightly more interesting way than just passing Strings around.

The third piece of magic is in the architectural underpinnings of JFugue, and here the piece of magic has two subparts: the event-based musical event architecture, and the architecture of the Staccato parser itself. With the architecture centered on musical events, you can use JFugue to convert *from* any music format *to* any music format. Staccato-to-MIDI happens to be the default. But you can also use JFugue to convert from MusicXML to Lilypond, or MIDI to MusicXML, or MusicXML to Staccato – any combination of `Parser` and `ParserListener`. And the architecture of the Staccato parser lets you create your own musical tokens that you can inject into a Staccato string, so you can create a `CarnaticTokenSubparser` and pass in tokens of Indian music, and have those tokens converted to microtone notations. Oh, and then you can still have that music played as MIDI, or exported to MusicXML or Lilypond. And you can represent your new Carnatic tokens in a Pattern that can be manipulated in ways that make musical sense. Wow!

The fourth piece of magic is that once the basic things become easy to express, we can have lots of fun doing more unique things that are not found in your typical music API. JFugue's `Rhythm` class is the perfect example of this: you can create a hard rock beat in four lines of creative code. You can add new chords to JFugue's chord map, which defines a set of intervals for each chord, and have them immediately available in your Staccato strings. You can get all of the `Chords` in a `ChordProgression` and have each of them played like four rapid-fire eighth notes. I didn't intend for `ReplacementMapProcessor` to have the ability to create musical fractals, but it can be used that way, as you'll see. The API encourages and facilitates creative interaction with music and programming.

But wait, there's more! I'll even throw two extra pieces of magic. The first is that all of this is built on a solid bed of music theory. Chords and Scales are defined by Intervals, ChordIntervals contain Chords and Chords contain Notes, and all of this works seamlessly together. The second extra bit of magic is that the chaining syntax of the JFugue API makes all of these things very expressive. It's easy to fluidly chain pieces together in a single line without the need for temporary variables. For example, if you want to find the notes to the second chord of a I-II-IV progression when played with a root of B-flat, you say: `Notes[] notes = new ChordProgression("I II IV").setRoot("Bb").getChords()[1] .getNotes();`

And now, let's get on with the music programming!

# 1.4

# *Transitioning from JFugue 4 to JFugue 5*

If you are already familiar with earlier versions of JFugue, particularly JFugue v4.0.3, you may be wondering how this version is different, and what you need to do to adjust between the two versions. This chapter provides the highlights.

## About JFugue 5

*JFugue 5 is a total, from-the-ground-up rewrite of JFugue.* Many of the original concepts of JFugue persist in JFugue 5 and many classes maintain the same names (e.g., `Player`, `Pattern`, `ParserListener`, `Note`, `Rhythm`), but the methods of these classes are different, and the classes have been reorganized into a new package structure. If you are using an integrated development environment (IDE) such as Eclipse or NetBeans, simply "organizing imports" in your existing code should resolve the correct package names for the basic classes like `Player` (which is in `org.jfugue.player`) and `Pattern` (which is in `org.jfugue.pattern`).

Some classes from previous versions of JFugue has been renamed in this version; for example, the former `DeviceThatWillReceiveMidi` is now simply `MidiReceiver`. Expect some classes with the same names as their JFugue 4 counterparts, like `Note` and `Rhythm`, to have substantial

differences in the methods they provide. There is also a lot of new and exciting functionality in JFugue 5, particularly around music theory, so please explore this guide and the API documentation to learn more!

## Terminology Change: From JFugue MusicString to Staccato

What had been called the JFugue MusicString in the past is now gaining its own traction and has a new name: Staccato. The intention is that, while Staccato had its origin as JFugue's MusicString, it has earned its own wings and is a music format that can be used on projects outside of JFugue. And now, JFugue 5 is a reference implementation for parsing Staccato music.

## Adjustment of Octave Range and Default Octaves

In the past, the MusicString allowed octaves in a range from -1 to 9; for example, `C-1` would be a C-note in the lowest octave. The octaves have changed: the allowable range is now 0 through 10, meaning that the lowest note in Staccato is `C0` and the highest note is `G10`. This is now consistent with MIDI note numbers.

This has the effect of making music sound an octave higher. If you previously composed a song with `C5`, you'll notice that `C5` in JFugue 5 sounds one octave higher than `C5` in JFugue 4. The new version is more consistent with the rest of the musical world; in particular, it ensures that `A5`, the A just above Middle-C, is the same as 440Hz, which is verified by this simple program (also in the code as `A440Test.java`):

```
player.play("A5 m440"); // Make sure that A5 sounds like 440Hz
```

Eliminating the -1 octave also removes a parsing ambiguity when determining whether a note was in the -1 octave (e.g., `C-1`) or was the end of a tie (e.g., `C-w`).

The default octave – which allows one to say `C` and assume they would hear Middle-C – is 5. The default octave for a chord, such as `Cmaj`, had been 3 in past versions of JFugue but is now 4, since that is the typical octave for notes in the bass clef (just below Middle-C). The defaults for all note settings, which includes octave, duration, and on/off velocity, are now adjustable with the `DefaultNoteSettingsManager`.

Despite the changes you may have to make to your MusicStrings (now Staccato strings), hopefully your headache will be eased with the understanding that the new octave settings are more correct than the previous ones.

## Elimination of Brackets around Note Numbers

If you have programmed music using note numbers, such as `[74]w`, please remove the brackets from the note numbers, giving `74w`. Brackets should be used in cases where the contents of the brackets is a string representation of a value that needs to be looked up (e.g., `I[Piano]`, `T[Allegro]`). This includes percussive instruments in the 10th MIDI channel, so if you have `V9 [Bass_Drum]w`, that is a legitimate use of the bracket notation and those brackets must remain.

## Change in Expressing MIDI Commands in Music Strings

In previous versions of JFugue, MIDI commands such as Pitch Wheel or Controller Events were specified by starting a token with a special character, such as `&` or `^`. Those characters were hard to remember and did not provide for readability of the music string; in addition, there are only so many special characters that one may type, so they limited the number of "meta-musical expressions" (instructions about the music) that could be specified in a Music String.

In JFugue 5, MIDI commands are invoked through the Function system. JFugue 5's Functions allow an end-user to name a function that accepts a set of parameters in parentheses. JFugue pre-defines functions for Pitch Wheel, Controller Events, and other MIDI capabilities. Now, these look like `:PitchWheel(x,y)` or `:PW(x,y)`, or `:Controller(x,y)` or `:CON(x,y)`. The MIDI commands are still interpreted by the Parser and come through a `ParserListener` in specific methods, such as `onPitchWheelParsed` and `onControllerEventParsed`.

## New Chords and Altered Chord Names

Previously, some complicated chords (e.g., Dominant 7th 6/11) had names like DOM7<6>11. Now, the percent character is used in those chord names. In addition, JFugue 5 provides new chords, as well as a mechanism for you to add chords with one line of code.

## Elimination of Music Classes that Wrapped Basic Values

Previous versions of JFugue had classed like `Instrument` and `Tempo` that did little more than wrap a value representing an instrument or a tempo. Many of these classes do not exist in JFugue 5; it is sufficient to share information about an instrument by simply passing along a byte value representing an instrument, since the former `Instrument` class provided little else. This also impacts the methods in `ParserListener`, as you will see in the next section. For constants representing instrument names (e.g, PIANO = 0), see `MidiDictionary`.

## More and Different Methods in ParserListener

`ParserListener` is probably the most commonly-used interface in JFugue. In JFugue 5, the methods in `ParserListener` are significantly different from previous versions. First, most of the methods start with the word 'on' (e.g., `onTempoChanged`, `onNoteParsed`). Parameters for methods like `onInstrumentParsed` on `onControllerEventParsed` are now the byte values representing the changed or parsed values rather than classes that wrap these values, as discussed in the previous section. New methods have been added to support new functionality (e.g., `onLyricParsed`, `onMarkerParsed`). Two new workflow methods, `beforeParsingStarts` and `afterParsingFinished`, let your implementation of a `ParserListener` know when to do important set-up and tear-down activities.

## Updates to Rhythm API and Introduction of DefaultRhythmKit

The `Rhythm` class is simpler to use in JFugue 5. Substitutions, which allow you to say what percussion sound is represented by what character, is now handled by a simple `Map<Character, String>`, and a default map called the `DefaultRhythmKit` comes pre-loaded with commonly-used characters and their corresponding percussion sounds. At the same time that the `Rhythm` class has been made simpler, it has also been made more powerful, and it now features Z-ordering or layers and a hierarchy of different types of "alternate" replacements, which allow you to create a long-duration rhythm that has switch-ups at points during the duration. `Rhythm` no longer supports chromatic notes.

## New Intervals Class and Removal of IntervalNotation Class

Previous versions of JFugue had a class called `IntervalNotation`, which was a separate notation style for specifying notes. That concept no longer exists. JFugue 5 has a different class, `Intervals`, which is consistent with intervals in music theory and can be used to create chords, scales, keys, and melodies.

## Removal of PatternTransformer and PatternTool Classes

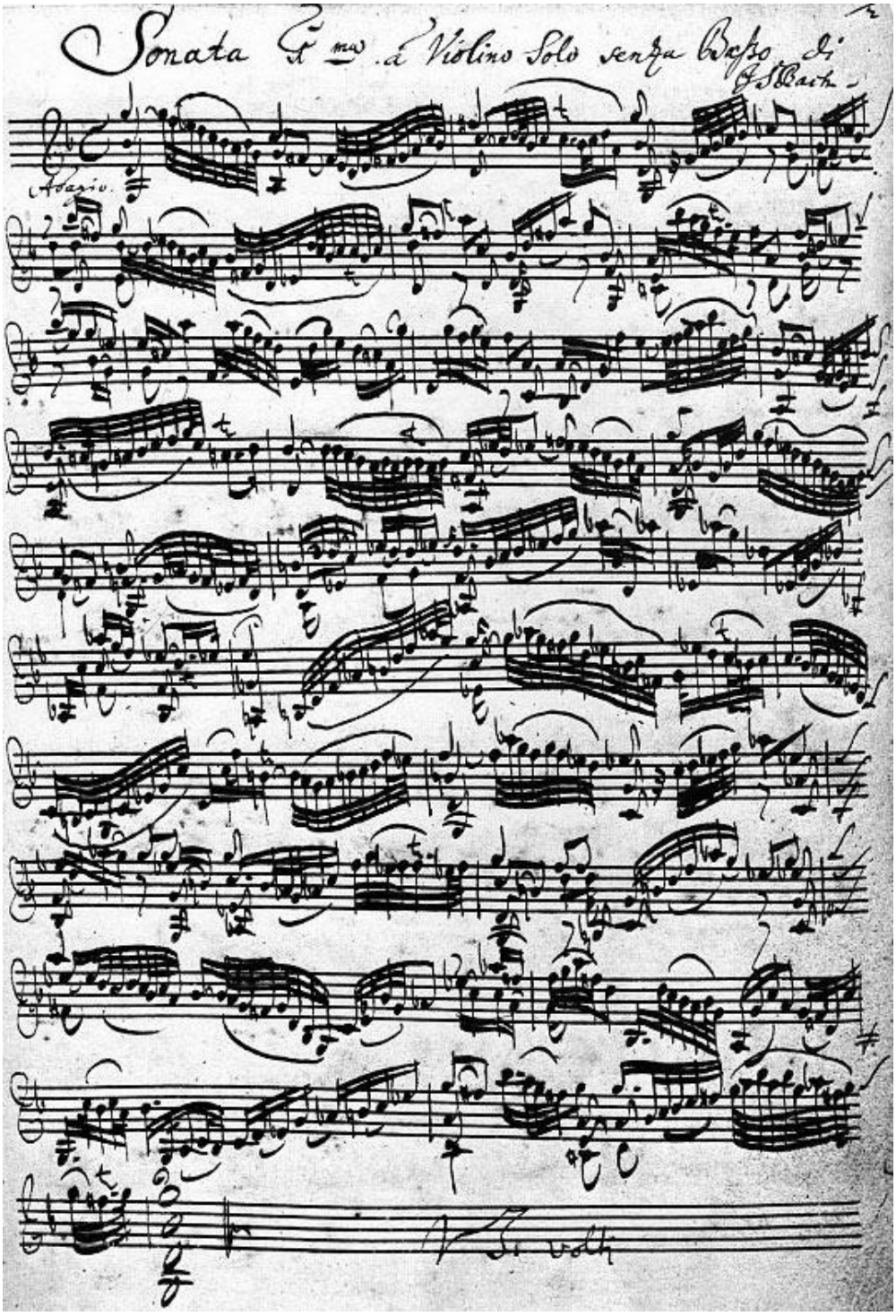
JFugue 5 does not provide a `PatternTransformer` and `PatternTool` class. However, you can still write code that transforms a pattern using a `ParserListener`, and the `Pattern` class now has `transform()` and `measure()` methods that take a `ParserListener`.

## Update to Key Signature and New Time Signature

JFugue now has a time signature command, and an updated key signature command (`KEY:` instead of `K`).

# Part 2

## Staccato: Writing Music for JFugue




Front page of the autograph for J. S. Bach's
"Sonata for single violin #1 in G minor" (BWV 1001)

# 2.1

# *Introduction to Staccato*

In this chapter, you will learn most of what you need to know to start creating music with JFugue. Specifically, you will learn about the features of Staccato, which is the format used to specify music in strings. Staccato lets you to create music with notes of varying octaves, durations, and instruments. You'll also learn all about chords, tuplets, tempo, controllers, key signatures, and more. Finally, you'll learn how to transcribe sheet music into a Staccato string.

## Introducing Staccato

Staccato is the name of the format that JFugue uses to specify music in strings. Staccato originated as the MusicString that you may be familiar with from past incarnations of JFugue. As a specific format now with its own name and separate existence, Staccato is a new standard in musical formats and is intended for use in other projects aside from JFugue. Perhaps JFugue is the reference implementation of Staccato.

*Staccato is optimized for use by people who write, read, and create music in software programs*. There are other music formats that are better for other tasks. For example, MusicXML, the *de facto* music representation standard for sharing musical data across applications, provides an extremely rich set of tags for specifying intricacies in music lithography

and style. The LilyPond format is used for music engraving. MIDI is a format for communicating music among musical devices, and is not intended to be user-facing. JFugue provides tools for converting music among these formats.

Staccato is one of the magical things about JFugue. It is the reason that JFugue is easy to use and allows a programmer to create music quickly. Staccato is designed to be easy to learn, easy to write, and easy to read.

The following example is the simplest demonstration of Staccato. This example plays a C note using the default octave and duration.

```
Player player = new Player();
player.play("C");
```

Staccato is not case-sensitive. You will see a consistent style of upper- and lowercase used in the examples below. While this style is designed to make Staccato music as readable as possible, adherence to this particular style is not required for JFugue to properly parse the string. Additional guidelines for Staccato style are presented towards the end of this chapter.

In the following sections, you will learn how to represent music using Staccato.

## Staccato by Example

Here are some examples of MusicStrings:

```
Player player = new Player();
player.play("C");
player.play("C7h");
player.play("C5maj7w");
player.play("G5h+B5h+C6q_D6q");
player.play("G5q G5q F5q E5q D5h");
player.play("T[Allegro] V0 I0 G6q A5q V1 A5q G6q");
player.play("V0 Cmajw V1 I[Flute] G4q E4q C4q E4q");
player.play("T120 V0 I[Piano] G5q G5q V9 [Hand_Clap]q Rq");
```

Each set of characters separated on either side by one or more spaces is called a *token*. A token represents a note, chord, or rest; an instrument change; a voice or layer change; a tempo indicator; a controller event; the definition of a constant; and more, as described in more detail in this chapter. In the example above, the first four examples each contain one token, and the last four examples each contain eight tokens.

# 2.2

## *Notes, Rests, and Chords*

You can't have music without notes! Let's get right into it – there is a lot to cover, and notes are the most complex of the musical data objects in JFugue. This chapter starts from the most basic cases and moves on to more complex constructs. Two tips for reading this chapter: 1) this will be a lot more fun if you try some of the examples as you read; 2) you do not need to remember everything in here to get started with making beautiful music.

### Basic Notes and Rests

Specifying a note in Staccato is intuitive. In the simplest case, you simply provide the name of the tone: `C`, `D`, `E`, `F`, `G`, `A`, or `B`. By default, these notes are played in the fourth octave and for a quarter duration. Sharp notes are indicated with a hash mark `#`, and flat notes are indicated with a `b`; both of these follow the root, so `F#` or `Bb` would indicate an F-sharp or a B-flat. Use `R` to represent a rest, which also defaults to a quarter duration.

A melody of notes consists of notes separated by spaces, as in this example:

```
player.play("C C E E G G E"); // Twinkle, twinkle little star
```

To specify an octave for a note, append a number from 0 to 10 to the note. `C4` is Middle-C. MIDI supports 128 notes, from C in the zeroth octave (`C0`) to G in the tenth octave (`G10`).

## Notes by Number

Notes can be specified using a note number instead of a note letter plus octave. The note number corresponds to the MIDI number that represents the note. For example, MIDI note 60 is Middle-C. To specify a note using its note number, simply use the number in the place of the note letter and octave:

```
player.play("60"); // Plays Middle-C, same as player.play("C")
```

MIDI note values are shown in Table 1.

| Octave | C | C# | D | E*b* | E | F | F# | G | G# | A | B*b* | B |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| **1** | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| **2** | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 |
| **3** | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 |
| **4** | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 |
| **5** | **60** | 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 | 70 | 71 |
| **6** | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 | 80 | 81 | 82 | 83 |
| **7** | 84 | 85 | 86 | 87 | 88 | 89 | 90 | 91 | 92 | 93 | 94 | 95 |
| **8** | 96 | 97 | 98 | 99 | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 |
| **9** | 108 | 109 | 110 | 111 | 112 | 113 | 114 | 115 | 116 | 117 | 118 | 119 |
| **10** | 120 | 121 | 122 | 123 | 124 | 125 | 126 | 127 | | | | |

**Table 1. MIDI note values. Middle-C (C in the 5th octave; MIDI note value 60) is marked in green on the left side of the table**

Specifying a note numerically results in a less readable Staccato string, but the ability to specify notes numerically can be useful if you are calculating note values in your application and do not want to convert the note values to letters – although the JFugue API does provide methods that convert note values to letters, as explained in Section 4.2.

In previous versions of JFugue, note numbers were placed in square brackets. Square brackets are no longer used to indicate notes by number. They *are* still used to indicate a note by name, as described next.

## Notes by Name, and Percussion Note Names

Wherever you can use a numeric value in Staccato, you can also use a string surrounded by square brackets, and the name will be looked up in a map maintained by the Staccato parser. For notes, this is most commonly used to refer to percussion instruments.

MIDI has 16 channels, and the tenth channel is set aside specifically to represent percussion sounds. The percussion sounds played by the tenth channel are not chromatic (i.e., they do not have a range of tones). To make the sounds, the percussion channel uses the same note values that are used for chromatic notes in other channels. This is why you see images of percussion instruments above the piano keys of many MIDI keyboards – if you are in the percussion channel, pressing those keys will make the percussion sound indicated by the image instead of the regular tone of the key.

Whereas 60 is a Middle-C in most channels, in the percussion channel 60 plays a "hi bongo" drum.

When writing music for the percussion channel, it makes more sense to write the music using the intended percussion instrument. Instead of writing the following line:

```
player.play("V9 60q"); // V9 is the 10th channel
```

It makes more sense to write:

```
player.play("V[Percussion] [HI_BONGO]q");
```

The Staccato parser will look up the value of `HI_BONGO`, find it to have a value of 60, and will play the corresponding drum sound. This example skips ahead a bit to introduce the Voice (i.e., channel) command while also demonstrating that voices can also be specified by a name in square brackets.

## Sharps, Flats, and Naturals

You can indicate that a note is sharp or flat by using the `#` character to represent a sharp, and the `b` character to represent a flat. Place the `#` or `b` character immediately after the note name; for example, a B-flat would be represented as `Bb`.

Staccato also supports double-sharps and double-flats, which are indicated by using `##` and `bb`, respectively.

If you use a key signature in your music string (see Section 2.6), you can indicate a natural note by using the `n` character after the note. For example, a B-natural would be represented as `Bn`. If you use a key signature and do not mark a note as a natural, the note value will be automatically changed based on the key signature. For example, if you specified an F-major key signature, a B note would be played like a B-flat automatically.

## Internal Intervals

Later, you will learn that Intervals are one of the foundational classes in JFugue. They make it easy to specify chords – for example, a major chord is represented by a `1 3 5` interval, where `1` is the root. A minor chord would be represented by the interval `1 b3 5`, in which the second note is a flat third.

An internal interval lets you specify an interval relative to a specific note. To do this, use the single-quote character, followed by an interval value. For example, we know that `C` will give a C note. Well, `C'b3`, which is the `b3` interval of C, is E-flat. In fact, anywhere you can say `Eb`, you can equivalently say `C'b3` if you cared to – JFugue considers them to be the same note. Since octaves modify the note itself, if you are using an octave number, place it next to the note and before the single-quote: `C5'b3`.

## Chords

A chord is a combination of notes played simultaneously (i.e., in harmony). Chords have a root note and a phrase that expresses the intervals for the harmonic notes. For example, a C-Major chord has a root note of C and is a major triad. To specify a chord in Staccato, you first provide the root note (including sharps, flats, and octaves, or expressing the note by number or name) and, immediately following, specify the intervals using the name of the chord. For example, to play a C-Major chord, you would use the following call:

```
player.play("Cmaj"); // Plays a C Major chord
```

JFugue provides many chord names by default. You may also add new chord names and intervals; this capability is explored in more detail in Section 4.3. The chord names that JFugue provides are shown in Table 2. (As a rule of thumb, where you might expect to see a slash in the chord name (e.g., “Minor 6/9”), JFugue places a percent sign; otherwise, the slash would be ambiguous with the use of the slash to indicate a decimal duration, as you will learn about shortly.)

To specify an octave with a chord, follow the chord root with the octave number. For example, an E-flat, 6th octave, major chord would be `Eb6maj`. An easy way to remember where to place the octave is that the octave describes the root note in more detail, so it should be next to the root. If a number follows the chord name, then the number is associated with the chord itself: for example, `Cmaj7` describes a C-Major seventh chord, not a C-Major chord in the seventh octave. The default octave for a chord is the 4th octave, which is slightly lower than the default octave for a note (5th octave).

The rest of this chapter introduces durations, triplets, ties, and dynamics; all of these items apply to chords as well as notes. In later sections, JFugue's support for music theory, including additional details about chords and an introduction of the `ChordProgression` and `Interval` classes, provide much more detail about chords. This chapter is limited to introducing the ability to specify chords in a Staccato string.

## Chord Inversions

Chords have a root note, and in most cases the root is also the bass (or lowest) tone in the chord. A chord inversion is a rearrangement of the chord that makes a different note the bass. For example, a C Major chord in the 3rd octave consists of the notes C3, E3, and G3; the first inversion of that chord puts E3 in the bass, and the chord is played as E3 G3 C4. See Figure 1.

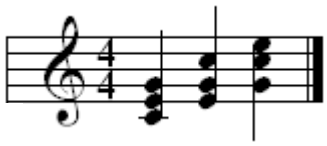


**Figure 1. Chord inversions of C Major: no inversion, first inversion, and second inversion**

Staccato provides two ways to express chord inversions. In both cases, the inversion indicator immediately follows the chord name. First, you may use the caret symbol (`^`) as many times as inversions. For example, a first inversion would use a single caret; a second inversion would use two. This call plays the three inversions of C-Major:

```
player.play("C3maj C3maj^ C3maj^^"); // Inversions of C-Major
```

Second, you may use a single caret followed by the new bass of the chord. The following call plays the same music as the previous example:

```
player.play("C3maj C3maj^E C3maj^G"); // Inversions of C-Major
```

When you see sheet music that places the chord names over the staff and you see a chord with a slash followed by another letter (for example, "C/E"), that is an inversion: in this case, you are being asked to play a C-Major chord with E as the bass note. In Staccato, this is `Cmaj^E`.

| Common Name | JFugue Name | Intervals |
|---|---|---|
| **Major Chords** | | |
| Major | `MAJ` | 1 3 5 |
| Major 6th | `MAJ6` | 1 3 5 6 |
| Major 7th | `MAJ7` | 1 3 5 7 |
| Major 9th | `MAJ9` | 1 3 5 7 9 |
| Added 9th | `ADD9` | 1 3 5 9 |

| | | |
|---|---|---|
| 6/9 | MAJ6%9 | 1 3 5 6 9 |
| 7/6 | MAJ7%6 | 1 3 5 6 7 |
| Major 13th | MAJ13 | 1 3 5 7 9 13 |
| **Minor Chords** | | |
| Minor | MIN | 1 *b*3 5 |
| Minor 6th | MIN6 | 1 *b*3 5 6 |
| Minor 7th | MIN7 | 1 *b*3 5 *b*7 |
| Minor 9th | MIN9 | 1 *b*3 5 *b*7 9 |
| Minor 11th | MIN11 | 1 *b*3 5 *b*7 9 11 |
| 7/11 | MIN7%11 | 1 *b*3 5 *b*7 11 |
| Minor Added 9th | MINADD9 | 1 *b*3 5 9 |
| Minor 6/9 | MIN6%9 | 1 *b*3 5 6 |
| Minor Major 7th | MINMAJ7 | 1 *b*3 5 7 |
| Minor Major 9th | MINMAJ9 | 1 *b*3 5 7 9 |
| **Dominant Chords** | | |
| Dominant 7th | DOM7 | 1 3 5 *b*7 |
| Dominant 7/6 | DOM7%6 | 1 3 5 6 *b*7 |
| Dominant 7/11 | DOM7%11 | 1 3 5 *b*7 11 |
| Dom 7th Sus | DOM7SUS | 1 4 5 *b*7 |
| Dom 7/6 Sus | DOM7%6SUS | 1 4 5 6 *b*7 |
| Dominant 9th | DOM9 | 1 3 5 *b*7 9 |
| Dominant 11th | DOM11 | 1 3 5 *b*7 9 11 |
| Dominant 13th | DOM13 | 1 3 5 *b*7 9 13 |
| Dom 13th Sus | DOM13SUS | 1 3 5 *b*7 11 13 |
| Dom 7th 6/11 | DOM7%6%11 | 1 3 5 *b*7 9 11 13 |
| **Augmented Chords** | | |
| Augmented | AUG | 1 3 #5 |
| Augmented 7th | AUG7 | 1 3 #5 *b*7 |
| **Diminished Chords** | | |
| Diminished | DIM | 1 *b*3 *b*5 |
| Diminished 7th | DIM7 | 1 *b*3 *b*5 6 |
| **Suspended Chords** | | |
| Suspended 2nd | SUS2 | 1 2 5 |
| Suspended 4th | SUS4 | 1 4 5 |

**Table 2. Chord provided by JFugue**

## Duration

*Default duration if no duration is specified for a note: Quarter duration*

Durations are specified with either a letter or a decimal value that represents a fraction of a whole duration. Using a letter is quicker to write and easier to read once you remember the durations most commonly found in music (whole, half, quarter, eighth, sixteenth). Table 3 shows the durations that are used in Staccato.

| Duration | Character | Decimal value |
|---|---|---|
| **W**hole | W | /1.0 |
| **H**alf | H | /0.5 |
| **Q**uarter | Q | /0.25 |
| E**i**ghth | I | /0.125 |
| **S**ixteenth | S | /0.0625 |
| **T**hirty-second | T | /0.03125 |
| Si**x**ty-fourth | X | /0.015625 |
| **O**ne-twenty-eighth | O | /0.0078125 |

**Table 3. Durations that can be specified for a note**

The duration comes after the octave if an octave is specified, or the chord name and any inversions; otherwise it comes immediately after the note. For example, `C5w` is a whole C note, fourth octave (as is `Cw`, since the fifth octave is the default for a note). `F#5h` is a F-sharp in the fifth octave, half duration, `Rq` is a quarter duration rest, and `emini` is an E-minor chord, eighth duration, and a perfectly legal Staccato string but it might be better stylized as `EminI` (when using durations with chord names, the Staccato string is more readable if the chord name is printed in lowercase and the duration is printed in uppercase).

Staccato supports dotted durations (for example, `q.`), in which case the note is played for the original duration plus half of the original duration. A dotted quarter note is the same as combining a quarter note with an eighth note.

You can specify a duration as a group of letters to form an aggregate duration. For example, `Db6wwh` is a D-flat, sixth octave, that will play for a total of two whole durations plus a half duration. This is a single tone—the note will not stop and play again after any of the durations, it will simply sound for the duration of two and a half whole notes. You can group any duration letters together in any order. If you use a dotted duration, the dot applies to the single duration letter immediately to its left.

If you have many durations that you want to specify as an aggregate, you can use numbers as a shortcut for repeating the same duration character. For example, if you have a note that should play for 12 whole durations, you can say `Db6w12` instead of `Db6wwwwwwwwwwww`. You can also aggregate these numeric durations, so `Db6w12i3` will do exactly what you expect: play `Db6` for twelve whole durations plus three eighth durations.

Durations may also be specified as decimal values, where the decimal represents the portion of a whole note. To specify a decimal duration, you must first use the slash (`/`) character. For example, `Db6/2.5` plays the

same thing as `Db6wwh`. The decimal notation may be easier to use if you create programs that generate algorithmic music.

Here are some additional examples of durations:

```
player.play("Aw");          // A5 whole note
player.play("E7h");         // E7 half note
player.play("60wq");        // Middle-C whole+quarter note
player.play("60w5");        // Middle-C, 5 whole durations
player.play("G8i.");        // G8 dotted-eighth note
player.play("Bb6/0.5");     // B-flat, 6th octave, half note
player.play("C7maj^^q");    // C-major chord, 2nd inversion,
                            // 7th octave, quarter note
```

## Triplets and Other Tuplets

Tuplets are groups of notes that have a different rhythmic structure than would be expected based on the time signature. For example, in a musical piece with a 4/4 time signature, a triplet of three quarter notes will be played such that the three notes fit inside the duration of a half note (a half duration being the next larger duration of the quarter duration). This means that instead of being played at 0.25 times the duration of a whole note, these notes are played at 0.1667 times the duration of a whole note (0.5, the next higher note value, divided across three notes). Figure 2 shows the music notation used for triplets.

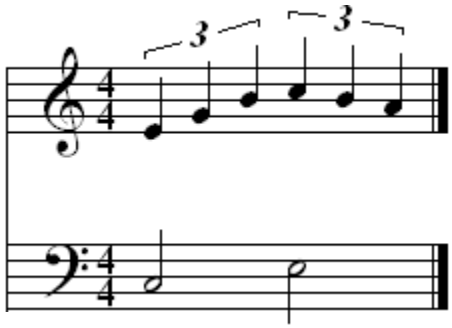



**Figure 2. Two triplets (also known as 3-tuplets) of quarter notes. Notice how each triplet has the same duration of the half note in the bass staff.**

Triplets are a special case of tuplets in which there are three notes in the group. Triplets are the most common tuplet, although other tuplets are possible (in both music theory and JFugue).

For a triplet, three notes are played with the same duration of the next greater duration; this is a 3:2 tuplet. In other words, this is three notes played in two beats. Another example is the 5:4 tuplet, a quintuplet, in which five notes are played in four beats.

In JFugue, a tuplet is indicated using an asterisk immediately following the duration for the note, followed by the type of tuplet, such as 3:2 or 5:4. For example, `E6q*3:2` plays an E note, sixth octave, as part of a 3:2 tuplet. This would be similar to saying `E6/0.1667`. `F4q*5:4` is identical to `F4/0.2` and would play one of a quintuplet of notes. Of course, if you are specifying a tuplet, you probably have multiple notes that you want as part of that tuplet. While you can specify each note individually, you might prefer to show the notes grouped together. For this, you can surround the group of notes in parentheses and add the duration immediately after the closing parenthesis. Thus, `(E G B)q*3:2` plays the E, G, and B notes in melody, with these three notes all being played in the duration of a half note. This use of parentheses is an example of "Collected Notes," which will be discussed shortly.

JFugue provides a default tuplet value. Since the 3:2 tuplet is so common, simply saying `Cq*` will give a C note with a 3:2 tuplet. And since `q` is the default duration for a note anyway, you can simply say `C*` in this case to get a C note with a duration of `0.16666666666666666`. Three of those will give the duration of a single half note.

## Ties

In sheet music, a tie connects two notes of the same pitch[1], and indicates that the two notes are to be played as one note, with the total duration equal to the sum of the durations of the tied notes. Ties are often used in sheet music to depict a note that has a duration which stretches across the bar line between two measures (Figure 3). Ties may also be used to connect notes to create a combined duration that cannot otherwise be indicated by note symbols, such as a half note plus an eighth note (Figure 4).

**Figure 3. Tying two notes across a measure**

**Figure 4. Tying two notes to achieve a combined duration**

[1] A line or curve connecting notes of different pitches is a slur, which indicates that the transitions between notes are to be played fluidly. Slurs are not currently supported in Staccato.

In Staccato, the dash symbol, -, is used to indicate ties. For a note that is at the beginning of a tie, append the dash to the end of the duration. For a note that is at the end of a tie, prepend the dash to the beginning of the duration. If a note is in the middle of a series of notes that are all tied together, two dashes are used: one before the duration, and one after. In each case, think of the dash as indicating whether the tie "follows" the duration of a note, whether it "continues" the duration of a note, or whether the note is in the middle of a tie, in which case the tie both "follows" and "continues" the duration. Each of these cases is shown in Figure 5, which uses the bar symbol (the vertical line or pipe character, |) to separate measures.

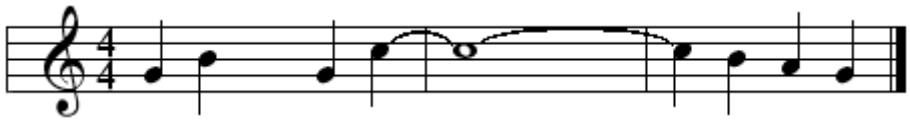


**Figure 5. Examples of ties in a Staccato string. The string for this sequence of notes is** `G5q B5q G5q C6q- | C6-w- | C6-q B5q A5q G5q`

Ties can also be used with decimal durations. To use a tie in this case, you must first indicate the use of a decimal duration with the slash (/) character. Then specify the decimal duration, using the tie dash before, after, or both before and after the duration. This string plays the same music as the string in Figure 5, but uses decimal durations: `G5/0.25 B5/0.25 G5/0.25 C6/0.25- | C6/-1.0- | C6/-.025 B5/0.25 A5/0.25 G5/0.25`.

## Notes That Continue Until Explicitly Stopped

The dashes used in ties can also be used to start a note that should play until it is explicitly stopped. For example, if you wish to implement a user interface for an organ, you may want to start a note when the user presses the organ key, and stop the note only when the user releases the key.

To start a note with undetermined duration, use the shortest possible duration followed by a dash; for example, `C6o-`. The duration is necessary because the parser needs to know whether a note is starting or stopping, and the placement of the dash relative to the duration (whether it comes after or before the duration) is used to determine whether the note is starting or ending. (I could have written JFugue to just know whether a note were starting or ending based on whether it had already been started, but I thought that could be too much automation that could lead to confusion about a note's state.)

To end a note, again use the shortest possible duration, this time preceded with a dash; for example, `C6-o`. Note that when you end a note,

it only ends the same note that was started. If you start `C6o-` and `B6o-` and then call `C6-o`, only `C6` stops; `B6` keeps going until `B6-o` is used.

## Note Dynamics: Note On and Note Off Velocities

Notes may be played with a specified attack and decay velocity. "Note On" velocity is how quickly the note reaches its full volume. "Note Off" velocity is how quickly the note decreases from its peak volume after it has played for its full duration. A note with a large value for "note on" velocity sounds like it takes a while to build up, and is commonly used for a "pad" instrument (e.g., Synth Pad). Notes with a large "note off" velocity continue to resonate after the note has been struck, like a bell or a guitar string.

Velocity for notes may be specified using the `a` for "note on" velocity and `d` for "note off" velocity. The letters harken back to an older version of JFugue where these are called "attack velocity" and "decay velocity", which is not an accurate set of terms but replacement single characters that clearly indicate "note on" and "note off" are hard to invent (forward slash and backslash had been considered since they are visual analogies to a note increasing to its full volume then decreasing at the end, but the forward slash is ambiguous with using a forward slash for decimal duration (e.g., `C4/5` could be C4 for 5 whole notes or C4 with a "note on" velocity of 5), and a backslash in a Java string would require a second backslash to escape the character (e.g., `C4/5\\10` instead of `C4/5\10`)). Each character is followed by a value of 0 through 127, which is a MIDI value that represents the velocity. If you do not specify velocity, a default value of 64 will be used for both dynamics, and the default value sounds good – you will not need to tweak this value unless you are doing special work to, say, simulate a foot pedal or evoke a particularly ethereal (large values) or choppy (small values) sounds using the velocity dynamics.

For any note, you can specify any or neither of the velocity values, but if you are specifying both values, the "note on" velocity needs to come before the "note off" velocity.

The following are value notes with attack and decay velocities set:

```
player.play("C5qa0d127");      // Sharp on vel., long off vel
player.play("E3w2d0");         // Default on vel, sharp off vel
player.play("C7maja30");       // C7, E7, and G7 (components of
                               // C7maj) will all play with a
                               // "note on" velocity of 30
```

## Notes Played in Melody and in Harmony

Notes that are to be played in melody – that is, one after another – are indicated by individual tokens separated by spaces, as shown in Figure

6. So far, all of the Staccato examples shown so far in this book have played notes in melody.

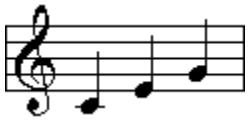


**Figure 6. Three notes in melody; the Staccato string is** `C4q E4q G4q`

Notes may also be played in harmony with other notes. The most likely case is that you want to play different staffs of music, such as a treble clef and a bass, or a piano, guitar, and flute playing different melodies. In these cases, you should use the Staccato Voice command, which plays music on different MIDI channels. The Voice command is discussed in the next section. A second likely case is that you want to play a chord, in which case you should use chords. If neither of these cases is true – if you really have some notes in the same staff that you want to play together – then read on!

You can use the plus character, `+`, to connect two or more notes that should play in harmony in the same channel. The plus character replaces a space between notes and indicates that the notes are part of one harmonic unit. Figure 7 shows an example.

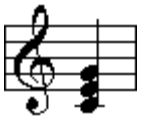


**Figure 7. Three notes in harmony; the Staccato string is** `C4q+E4q+G4q`

You may also find some occasions when a note is to be played in harmony while two or more notes are played in melody. To indicate notes that should be played together while played in harmony with other notes, use the underscore character, `_`, to connect the notes that should be played together. This is much clearer in a picture than in words, so please take a look at Figure 8. In this example, the `C4` note is played continuously (it has a half duration) while the `E4` and `G4` notes (each with a quarter duration) are played as a melody.

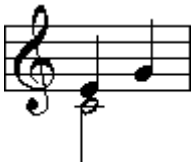


**Figure 8. A harmony and a melody played together; the Staccato string is** `C4h+E4q_G4q`

Like notes, rests and chords may also be played in harmony or in combined harmony/melody using the plus and underscore characters as connectors. Only notes, rests, and chords can take advantage of the + and _ characters.

## Collected Notes: Notes that Share Durations or Dynamics

As a shorthand notation, Staccato provides the ability to specify a series of notes, rests, and chords that might all have the same duration or dynamics (attack or decay velocities). To do this, group the notes in parentheses, and immediately after the closing parenthesis, provide the duration or dynamic information. Staccato will append the information at the end of the closing parenthesis to each of the elements inside parentheses.

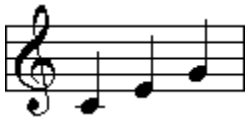


**Figure 9. Sharing durations across notes:** `(C E G)q`

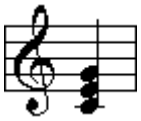


**Figure 10. Sharing durations across notes in harmony and mixing additional dynamics:** `(C+E+G)4/0.25a60d15`

## Summary

Notes are the most complex element in Staccato. If you've gotten to this point, step away and take a breath. Things get easier moving forward!

# 2.3

# *Voices and Layers*

This chapter discusses voices, also known as channels or tracks, which let you represent different staffs or instruments in harmony, and layers. Layers are a JFugue-specific way of creating polyphony within a single voice, which is particularly useful for playing percussion sounds in MIDI Channel 10 (or, as you would call it in Staccato, `V9` or `V[PERCUSSION]`).

## Voices

Staccato uses the word *voice* to refer to a channel or track. The MIDI specification allows for sixteen channels. A variety of MIDI functions, such as selecting an instrument, apply to a specific channel.

To specify a voice in Staccato, use the `V` character followed by the number of the desired voice, or a value in brackets that has been defined to represent a voice number (`[PERCUSSION]` is already defined to mean 9). Voice numbers are zero-based, so valid values are 0 through 15. The following example plays three perfect fifths – C and G, D and A, and E and B – by playing one set of notes in Voice 0 and the other in Voice 1.

```
player.play("V0 C D E  V1 G A B");     // Three perfect fifths
```

Since instrument selections are per voice, you will need to specify instruments after you indicate a voice:

```
player.play("V0 I[Piano] C D E  V1 I[Flute] G A B");
```

## The Percussion Voice and Layers

The tenth channel, of `V9` in Staccato, is special: it is used to represent percussive sounds. In `V9`, each "note" actually represents a non-chromatic percussion sound, such as drums, cymbals, tambourines, and woodblocks. For example, whereas Note 60 means "Middle-C" in the other voices, it means "Hi Bongo" in `V9`.

Fortunately, you can use predefined values for these percussion sounds instead of using the raw note numbers or note letters. The following example uses predefined values along with note durations to show that these values are treated just the same as notes:

```
player.play("V9 [Hi_Bongo]q [Hand_Clap]q Rq [Bass_Drum]/0.5");
```

As with regular notes, you can create "chords" by using the plus character between notes. For example, if you want to hear a bongo and hand clap at the same time, you could use the following:

```
player.play("V[Percussion] [Hi_Bongo]q+[Hand_Clap]q");
```

However, it can be tedious to specify all of your percussion sounds with plus and underscore characters. If you wanted to create a multi-layered percussion background for a song, your `V9` channel would quickly be the hardest thing to set up correctly. Wouldn't it be better if each percussion instrument could be represented in its own layer?

This is where Staccato's *layer* comes in. A layer makes it easier to specify "tracks" of percussion sounds. Layers only work with `V9`, and they work just like voices – the only difference is that there are no MIDI elements that work on a per-layer basis, because layers are purely a Staccato invention.

You can specify any instruments sounds within any layer (unlike a voice, you are not restricted to a single instrument per layer), although as a best practice, you may find it useful to use one layer per percussion instrument.

You may use up to sixteen layers (zero-based, so 0-15), and a layer is specified with the `L` character followed by the layer number or a value in brackets that has been defined to represent a layer number.

In the following example, the bongo and the hand clap will sound at the same time:

```
player.play("V9 L0 [Hi_Bongo]q L1 [Hand_Clap]q");
```

You can create "chords" of percussion instruments, just like you can with regular notes. For example, `"V9 [Hand_Clap]q+[Crash_Cymbal_1]q"` will play a hand clap and a cymbal crash at the same time, both for a quarter duration.

| MIDI Note | Predefined Value |
|---|---|
| 35 | ACOUSTIC_BASS_DRUM |
| 36 | BASS_DRUM |
| 37 | SIDE_STICK |
| 38 | ACOUSTIC_SNARE |
| 39 | HAND_CLAP |
| 40 | ELECTRIC_SNARE |
| 41 | LO_FLOOR_TOM |
| 42 | CLOSED_HI_HAT |
| 43 | HIGH_FLOOR_TOM |
| 44 | PEDAL_HI_HAT |
| 45 | LO_TOM |
| 46 | OPEN_HI_HAT |
| 47 | LO_MID_TOM |
| 48 | HI_MID_TOM |
| 49 | CRASH_CYMBAL_1 |
| 50 | HI_TOM |
| 51 | RIDE_CYMBAL_1 |
| 52 | CHINESE_CYMBAL |
| 53 | RIDE_BELL |
| 54 | TAMBOURINE |
| 55 | SPLASH_CYMBAL |
| 56 | COWBELL |
| 57 | CRASH_CYMBAL_2 |
| 58 | VIBRASLAP |
| 59 | RIDE_CYMBAL_2 |
| 60 | HI_BONGO |
| 61 | LO_BONGO |
| 62 | MUTE_HI_CONGA |
| 63 | OPEN_HI_CONGA |
| 64 | LO_CONGA |
| 65 | HI_TIMBALE |
| 66 | LO_TIMBALE |
| 67 | HI_AGOGO |
| 68 | LO_AGOGO |
| 69 | CABASA |
| 70 | MARACAS |
| 71 | SHORT_WHISTLE |
| 72 | LONG_WHISTLE |
| 73 | SHORT_GUIRO |
| 74 | LONG_GUIRO |
| 75 | CLAVES |
| 76 | HI_WOOD_BLOCK |
| 77 | LO_WOOD_BLOCK |
| 78 | MUTE_CUICA |
| 79 | OPEN_CUICA |
| 80 | MUTE_TRIANGLE |
| 81 | OPEN_TRIANGLE |

**Table 4. Predefined percussion note values**

# 2.4

## *Instruments*

For each voice, you may specify an instrument – and you can change instruments for a given voice during a song. To specify an instrument, use the `I` character followed by an instrument number or a value in brackets that has been defined to an instrument number.

```
player.play("V0   I[Piano] C   I[Flute] D   I[Guitar] E");
```

The MIDI specification defines 128 instruments, the names of which are listed in the table below. The actual sound that you will hear is generated by the MIDI synthesizer that the Java Virtual Machine (JVM) uses when playing the musical instructions in the MIDI data; different synthesizers have different sounds of varying quality. Often, the default soundbank that accompanies applications is of poor quality, and this has given MIDI files a bad rap and the stigma of something leftover from the 1980's. Other soundbanks, such as those from a company called Garritan, are nearly indistinguishable from real instruments.

| Instrument | Predefined Value |
|---|---|
| **Piano** | |
| 0 | PIANO |
| 1 | BRIGHT_ACOUSTIC |
| 2 | ELECTRIC_GRAND |
| 3 | HONKEY_TONK |
| 4 | ELECTRIC_PIANO |
| 5 | ELECTRIC_PIANO_2 |
| 6 | HARPSICHORD |
| 7 | CLAVINET |
| **Chromatic Percussion** | |
| 8 | CELESTA |
| 9 | GLOCKENSPIEL |
| 10 | MUSIC_BOX |
| 11 | VIBRAPHONE |
| 12 | MARIMBA |
| 13 | XYLOPHONE |
| 14 | TUBULAR_BELLS |
| 15 | DULCIMER |
| **Organ** | |
| 16 | DRAWBAR_ORGAN |
| 17 | PERCUSSIVE_ORGAN |
| 18 | ROCK_ORGAN |
| 19 | CHURCH_ORGAN |
| 20 | REED_ORGAN |
| 21 | ACCORDIAN |
| 22 | HARMONICA |
| 23 | TANGO_ACCORDIAN |
| **Guitar** | |
| 24 | GUITAR |
| 25 | STEEL_STRING_GUITAR |
| 26 | ELECTRIC_JAZZ_GUITAR |
| 27 | ELECTRIC_CLEAN_GUITAR |
| 28 | ELECTRIC_MUTED_GUITAR |
| 29 | OVERDRIVEN_GUITAR |
| 30 | DISTORTION_GUITAR |
| 31 | GUITAR_HARMONICS |

| Instrument | Predefined Value |
|---|---|
| **Bass** | |
| 32 | ACOUSTIC_BASS |
| 33 | ELECTRIC_BASS_FINGER |
| 34 | ELECTRIC_BASS_PICK |
| 35 | FRETLESS_BASS |
| 36 | SLAP_BASS_1 |
| 37 | SLAP_BASS_2 |
| 38 | SYNTH_BASS_1 |
| 39 | SYNTH_BASS_2 |
| **Strings** | |
| 40 | VIOLIN |
| 41 | VIOLA |
| 42 | CELLO |
| 43 | CONTRABASS |
| 44 | TREMOLO_STRINGS |
| 45 | PIZZICATO_STRINGS |
| 46 | ORCHESTRAL_STRINGS |
| 47 | TIMPANI |
| **Ensemble** | |
| 48 | STRING_ENSEMBLE_1 |
| 49 | STRING_ENSEMBLE_2 |
| 50 | SYNTH_STRINGS_1 |
| 51 | SYNTH_STRINGS_2 |
| 52 | CHOIR_AAHS |
| 53 | VOICE_OOHS |
| 54 | SYNTH_VOICE |
| 55 | ORCHESTRA_HIT |
| **Brass** | |
| 56 | TRUMPET |
| 57 | TROMBONE |
| 58 | TUBA |
| 59 | MUTED_TRUMPET |
| 60 | FRENCH_HORN |
| 61 | BRASS_SECTION |
| 62 | SYNTH_BRASS_1 |
| 63 | SYNTH_BRASS_2 |

**Table 5. Predefined instrument values**

| Instrument | Predefined Value |
|---|---|
| **Reed** | |
| 64 | SOPRANO_SAX |
| 65 | ALTO_SAX |
| 66 | TENOR_SAX |
| 67 | BARITONE_SAX |
| 68 | OBOE |
| 69 | ENGLISH_HORN |
| 70 | BASSOON |
| 71 | CLARINET |
| **Pipe** | |
| 72 | PICCOLO |
| 73 | FLUTE |
| 74 | RECORDER |
| 75 | PAN_FLUTE |
| 76 | BLOWN_BOTTLE |
| 77 | SKAKUHACHI |
| 78 | WHISTLE |
| 79 | OCARINA |
| **Synth Lead** | |
| 80 | SQUARE |
| 81 | SAWTOOTH |
| 82 | CALLIOPE |
| 83 | CHIFF |
| 84 | CHARANG |
| 85 | VOICE |
| 86 | FIFTHS |
| 87 | BASS_LEAD |
| **Synth Pad** | |
| 88 | NEW_AGE |
| 89 | WARM |
| 90 | POLY_SYNTH |
| 91 | CHOIR |
| 92 | BOWED |
| 93 | METALLIC |
| 94 | HALO |
| 95 | SWEEP |

| Instrument | Predefined Value |
|---|---|
| **Synth Effects** | |
| 96 | RAIN |
| 97 | SOUNDTRACK |
| 98 | CRYSTAL |
| 99 | ATMOSPHERE |
| 100 | BRIGHTNESS |
| 101 | GOBLINS |
| 102 | ECHOES |
| 103 | SCI_FI |
| **Ethnic** | |
| 104 | SITAR |
| 105 | BANJO |
| 106 | SHAMISEN |
| 107 | KOTO |
| 108 | KALIMBA |
| 109 | BAGPIPE |
| 110 | FIDDLE |
| 111 | SHANAI |
| **Percussive** | |
| 112 | TINKLE_BELL |
| 113 | AGOGO |
| 114 | STEEL_DRUMS |
| 115 | WOODBLOCK |
| 116 | TAIKO_DRUM |
| 117 | MELODIC_DRUM |
| 118 | SYNTH_DRUM |
| 119 | REVERSE_CYMBAL |
| **Sound Effects** | |
| 120 | GUITAR_FRET_NOISE |
| 121 | BREATH_NOISE |
| 122 | SEASHORE |
| 123 | BIRD_TWEET |
| 124 | TELEPHONE_RING |
| 125 | HELICOPTER |
| 126 | APPLAUSE |
| 127 | GUNSHOT |

# 2.5

## *Tempo and Time Navigation*

How fast do you want your music to play? How can you jump around in musical time? Let's find out!

### Tempo

Tempo indicates how quickly a song should be played. It is often expressed early in a Staccato string, since it applies to all musical events that follow the tempo command. Tempo is applied to the full song as opposed to individual voices. The Tempo command is a `T` followed by a value representing beats per minute (BPM). If no tempo is declared, the default is 120 BPM.

There are also several predefined values that you may use to make your tempo statement easier to read, as shown in Table 6. To use these values, use the `T` command followed by the name of the predefined value in square brackets, for example: `T[Adagio]`.

| Predefined value | BPM |
|---|---|
| GRAVE | 40 |
| LARGO | 45 |
| LARGHETTO | 50 |
| LENTO | 55 |
| ADAGIO | 60 |
| ADAGIETTO | 65 |
| ANDANTE | 70 |
| ADANTINO | 80 |
| MODERATO | 95 |
| ALLEGRETTO | 110 |
| ALLEGRO | 120 |
| VIVACE | 145 |
| PRESTO | 180 |
| PRETISSIMO | 220 |

**Table 6. Predefined tempo values**

The following examples show the use of tempo:

```
player.play("T120 V0 I[Piano] C D E  V1 I[Flute] G A B");

player.play("T[Largo] V0 I[Piano] E D C  V1 I[Flute] B A G");
```

## Beat Navigation

You can specify music to play at a specific point in the song by giving the number of beats into the song where the music should play. This is done using the at sign, `@`, followed by a value that, like a decimal duration for a note, consists of the number of whole notes in the integer portion, and a value representing shorter durations after the decimal point.

For example, if you want Voice 1 to play 1.5 beats into the song that Voice 0 is playing, you can express this as follows:

```
player.play("V0 Cq Dq Eq Cq | Eq Dq Fq Aq V1 @1.5 Bq Dq");
```

In this example, you can achieve the same effect by using `Rwq` or `R/1.5` instead of using `@1.5`. However, the beats passed to `@` is absolute, always using 0.0 as the beginning of the song; rests are relative to the other notes that have been played.

If you use JFugue to convert a MIDI song to Staccato using the `MidiParser` and `StaccatoParserListener` classes, you will see extensive use of the `@` command. This is because MIDI captures timestamps with musical events, and it does not capture rests explicitly. The use of `@` is necessary to make sure all music sounds at the correct time.

The values given to `@` do not have to be sequential during the song. For example, `"@100 Aq @50 Cq @200 Dq"` is a legitimate Staccato string. This would sound like `Cq Aq Dq`, with space in between each note.

Can you go back in time? Yes! Staccato music is fully parsed and turned into musical events before any music is played. If you go back in time, you are just changing the position where new musical events will be entered.

## Marker Navigation

In Section 2.6, you'll learn about lyrics and markers. Markers provide a way to place any text, such as comments, into a Staccato string. Markers can also be used to bookmark the beat time in one voice, which can then be accessed in other voices.

In the following example, `#mark` is used to define a marker, and `@#mark` is used to change the time to the same number of beats at which `#mark` was defined. `#mark` is the marker, and `@#mark` changes the time to match the given marker.

```
player.play("V0 Cq Dq #mark Fq Aq V1 @#mark Bq Dq");
```

This will sound like `Cq Dq` followed by `Fq` and `Bq` in harmony, followed by `Aq` and `Dq` in harmony. This is not a great example, as a clearer way to get this result would be `"V0 Cq Dq Fq Aq V1 Rq Rq Bq Dq"`, but the point is clear.

Markers need to be defined before the first reference to the marker in the Staccato string. For example, `"V0 @#mark Aq V1 Cq Dq #mark Fq"` will not work because `#mark` has not been declared by the time it is requested with `@#mark`.

# 2.6

# *Key Signatures, Time Signatures, Bar Lines, Markers, and Lyrics*

Staccato can represent key signatures and time signatures, and bar lines can be used to both visually separate measures and to provide indexes to measures. Staccato also provides markers that can be used to indicate sections of music, as well as lyrics that accompany the music.

## Key Signature

*Default: C-Major*

If you specify a key signature, JFugue will play the Staccato music in the key or scale that you indicate. For example, if you specify a key signature of F-major, any B notes that are in your Staccato music will be played as B-flat (unless you specify otherwise by using the natural using `n`, for example, `Bn`).

To specify a key signature, use the phrase `KEY:` (including the colon) followed by the root note of the key, then `maj` or `min` to indicate major or minor scale. For example, `KEY:Cbmaj` will set the key to C-flat major.

There is a second way to specify keys as well, and this way does not require you to translate the number of sharps or flats you see in a key signature in sheet music to a root and a scale. You can say `KEY:` followed simply by the number of sharps or flats that appear in the key signature. For example, `KEY:Cbmaj` is the same as `KEY:bbbbbbb` (see Figure 11), and `KEY:Gmaj` is the same as `KEY:#`.

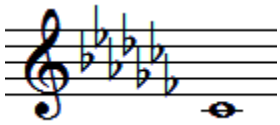


**Figure 11. This C-flat Major key signature can be represented as `KEY:Cbmaj` or `KEY:bbbbbbb`**

Note that key signatures are specified differently than in earlier versions of JFugue, in which key signatures were indicated with the letter `K` followed by the root and major or minor scale.

## Time Signature

*Default: 4/4*

Time signatures may be specified in Staccato, although JFugue will not alter the way that music is played when time signatures are present. However, time signature information will be maintained and is useful for other music formats, such as MIDI, MusicXML or LilyPond.

A time signature is specified by using the phrase `TIME:` (including the colon) followed by the beats per measure, a slash, and note that receives one beat. For example, a 6:8 time signature would be specified as `TIME:6/8`.

## Bar Lines

You can use bar lines in Staccato the same way you would expect to use them in music notation: to separate measures of notes. While the use of bar lines does not change how music is played, it does help make the Staccato music easier to read. And, since Staccto ignores whitespace between musical elements, you can use spaces to make bar lines appear at the same column of text when your application contains multiple pieces of music on different lines of code.

A bar line is simply the pipe character, `|`.

Although bar lines do not change how music is played, JFugue will parse bar lines and let any parser listeners know when bar line has been found.

Bar lines can optionally be followed by a number, which you can use as an index for the measure; for example, `|122`. These value of these indexes are not checked to make sure they follow a sequential order; you are free to use any number after a bar line. Alternatively, a bar line can be followed by a dictionary word. Dictionary items will be looked up and the corresponding numeric value will be sent in the `onBarLineParsed()` message to parser listeners. You may want to use indexes or dictionary words after a bar line for two reasons: to keep track of the measures as you write your Staccato music string, or to coordinate the activity of new parser listeners that you create.

## Markers

You can use a marker to provide the equivalent of comments in your Staccato string. Parser listeners will also be notified when a marker is parsed, so you can also use markers to encode special instructions that your parser listeners know how to respond to. A marker is the hash character followed by either a single word such as `#chorus`, or a phrase in parentheses, such as `#(this is the exciting part!)`.

If you are creating music in realtime using JFugue's `RealTimePlayer`, you can use markers to trigger effects that are outside the range of the musical events that parsers can listen to. For example, you may want to control a strobe light or animation of a character in addition to realtime music, in which case you can create and respond to markers like `#(strobe speed 100)` or `#(character1 start cabbage-patch)` and, presuming you've written a parser listener that knows how to parse these command, you can make your realtime application come to life. (Keep in mind that JFugue's standard Staccato-to-MIDI parsing is not done in realtime.)

## Lyrics

Lyrics can be added to a Staccato music string by using the apostrophe character followed by either a single word or a phrase in parentheses. Here are two examples: `'twinkle` and `'(twinkle twinkle little star)`.

While lyrics do not change the way that music is played, they are sent to parser listeners when they are found. This allows lyrics to be inserted into MIDI and other music formats. Keep in mind that as with markers, lyrics can be used in realtime (using the `RealTimePlayer` or `TemporalPLP`) if you would like to create an application that displays lyrics along with the music it plays, but the standard Staccato-to-MIDI parsing is not done in realtime.

# 2.7

# *Staccato Style and Music Transcription*

Congratulations, you have learned the major parts of Staccato! Don't worry, there's only one more section of this book that gets into even more details of what you can do with Staccato. But before that, let's take a break and share some thoughts on stylizing your Staccato strings so they are easy to read.

## Staccato Style Recommendations

The following recommendations of Staccato style are intended to help maximize the readability of Staccato strings as well as maintain consistency across users.

1. Staccato is not case-sensitive, and the number of whitespace characters between musical tokens (e.g., notes, instruments, voices) does not matter, as long as there is at least one space between tokens. Use upper/lowercase and spaces as much as necessary to improve the readability of your Staccato strings.

2. Use a capital letter for a character representing an instruction, such as `I`, `V`, and `L` (for Instrument, Voice, and Layer).

3. Use a capital letter for notes `C`, `D`, `E`, `F`, `G`, `A`, `B`, and the rest character, `R`. Use lowercase `b` for a flat and lowercase `n` for a natural.

4. Use lowercase letters when specifying chords: `maj`, `min`, `aug`, and so on. Along with the previous point, a C-major would be `Cmaj`.

5. Use a lowercase letter for note durations: `w`, `h`, `q`, `i`, `s`, `t`, `x`, `o`. However, if you are using durations after chords, it is more legible to use uppercase letters for note durations, such as `CmajQ`.

6. Make use of the fact that durations can be cumulatively added together to represent a total duration. You may find, for example, that in some cases is is more clear to say `Riii` than `Rq.` (dotted quarter duration).

7. Use mixed case (also known as camel case) to represent instrument names, percussion names, or tempo names: `I[Piano]`, `[Hand_Clap]q`, `T[Adagio]`.

8. Keep one space between each token, but if writing music for multiple voices, it's useful to put each voice on its own line, and use spaces to make the notes line up, as shown below.

9. Use the vertical bar character (also known as pipe), `|`, to indicate bar lines between measures.

10. Functions should use camel case for long names, or all capitals for abbreviated names (e.g., `:PolyphonicPressure` or `:PP`)

Below are examples of Staccto strings that employ some of these guidelines.

```
// First two measures of "Für Elise", by Ludwig van Beethoven
player.play("V0 E5s D#5s | E5s D#5s E5s B4s D5s C5s " +
            "V1 Ri       | Riii                    ");

// First a few simple chords
player.play("T[Vivace] I[Rock_Organ] Db4minH C5majW C4maj^^");
```

## Transcribing Sheet Music to Staccato

This section describes how to transcribe sheet music to Staccato. We'll use the first couple of measures of Antonio Vivaldi's "Spring" in this demonstration.

The following example uses the Pattern class, which you'll learn more about in future sections. For now, all you need to know is that a Pattern is an object that contains a Staccato string.

## *"Spring", from "The Four Seasons"*

*Antonio Vivaldi*

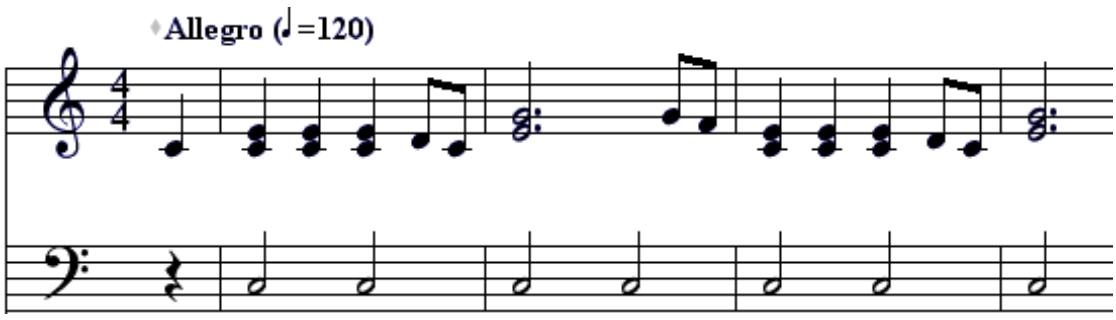



**Figure 12. Sheet music for a portion of "Spring"**

The first thing to notice is that there are two clefs, the treble clef and the bass clef, which means we will want to enter the music into two voices. We will put notes from the treble clef into Voice 0, and notes from the bass clef into Voice 1.

Before we start entering notes, let's start with the tempo ("Allegro") and time signature indicated in the sheet music.

```
Pattern pattern = new Pattern("T[Allegro] TIME:4/4");
```

Now we can start entering notes for the treble clef. We first see a C-note, quarter duration. Recall from Table 1 that this note is in the fourth octave. That means we have `C4q`.

Next is a bar line, indicating the end of the first measure. We'll add a pipe symbol, `|`, to our Staccato string.

Then we see a `C` and `E` note played in harmony. These are quarter notes again. The notation for these notes is `C4q+E4q`. And, we have three of them in a row.

Next are two eighth notes, `D` and `C`. Although they are barred together, the bar is purely stylistic, and does not change the way that eighth notes are played. We will need to add `D4i` and `C4i`.

Then there is another measure bar, so add another pipe symbol. Then there are two notes, `E` and `G`, played in harmony with a dotted half duration. We'll need to add `E4h.+G4h.` to the Staccato string. We can depend on the default octave here (the default is 4th octave for notes, 3rd octave for chords).

At this point, the transcribed music should look like this:

```
V0 C4q | C4q+E4q C4q+E4q C4q+E4q D4i C4i | Eh.+Gh.
```

Continuing on, there are the eighth G and F notes, so add `G5i F5i`.

The next eight notes are a duplicate of notes that we have already entered. We have a couple of options here. The most obvious option is that we can re-type the notes. Or, we could put the duplicated notes in a Pattern of their own, and use that Pattern whenever we see this set of eight notes. Or, we can also use methods on the Pattern class to repeat a subset of notes that have already been entered. Since Patterns aren't discussed in detail until the next chapter, let's leave the Pattern options aside and simply re-type (or copy-and-paste) the notes.

The transcribed music now looks like this:

```
V0 C4q | C4q+E4q C4q+E4q C4q+E4q D4i C4i | Eh.+Gh. G4i F4i |
C4q+E4q C4q+E4q C4q+E4q D4i C4i | Eh.+Gh.
```

Now we can work on the bass clef. First, there's a quarter rest. It is important to add this rest to the transcribed music so the clefs line up correctly. Add `Rq` to the bass clef. Add a bar line, too.

Next, we see a bunch of C notes, half duration. According to Table 1, these notes are in the 3rd octave. Add these notes to the bass clef, and the MusicString should look like this:

```
V1 Rq | C3h C3h | C3h C3h | C3h C3h | C3h
```

The program itself should look like this:

```
Pattern pattern = new Pattern("T[Allegro]");
pattern.add("V0 C4q | C4q+E4q C4q+E4q C4q+E4q D4i C4i | Eh.+Gh.
G4i F4i | C4q+E4q C4q+E4q C4q+E4q D4i C4i | Eh.+Gh.");
pattern.add("V1 Rq | C3h C3h | C3h C3h | C3h C3h | C3h ");
```

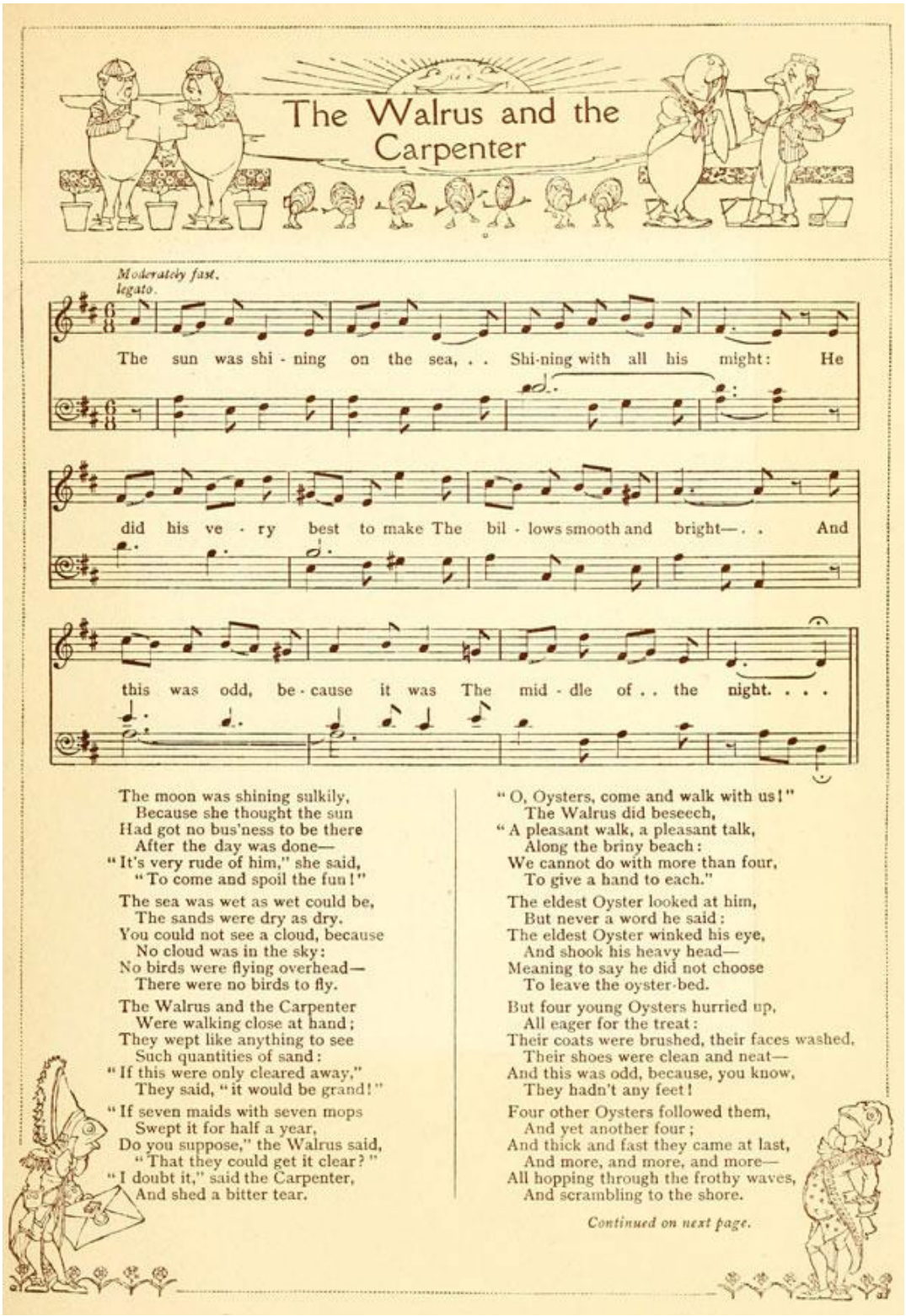
Since extra spaces are allowed in the Staccato string, you can space out the clefs so they line up more legibly:

```
Pattern pattern = new Pattern("T[Allegro]");
pattern.add("V0 C4q | C4q+E4q C4q+E4q C4q+E4q D4i C4i | Eh.+Gh.
G4i F4i | C4q+E4q C4q+E4q C4q+E4q D4i C4i | Eh.+Gh.");
pattern.add("V1 Rq  | C3h C3h                        | C3h C3h
        | C3h C3h                        | C3h ");


// Now, play the music!

Player player = new Player();
player.play(pattern);
```

Congratulations! You can now transcribe music into Staccato, the music representation system that has its origin in JFugue.

# Part 3

# Advanced Staccato



# The Walrus and the Carpenter

*Moderately fast, legato.*

The sun was shi - ning on the sea, . . Shi-ning with all his might: He
did his ve - ry best to make The bil - lows smooth and bright— . . And
this was odd, be - cause it was The mid - dle of . . the night. . . .

The moon was shining sulkily,
Because she thought the sun
Had got no bus'ness to be there
After the day was done—
"It's very rude of him," she said,
"To come and spoil the fun!"

The sea was wet as wet could be,
The sands were dry as dry.
You could not see a cloud, because
No cloud was in the sky:
No birds were flying overhead—
There were no birds to fly.

The Walrus and the Carpenter
Were walking close at hand;
They wept like anything to see
Such quantities of sand:
"If this were only cleared away,"
They said, "it would be grand!"

"If seven maids with seven mops
Swept it for half a year,
Do you suppose," the Walrus said,
"That they could get it clear?"
"I doubt it," said the Carpenter,
And shed a bitter tear.

"O, Oysters, come and walk with us!"
The Walrus did beseech,
"A pleasant walk, a pleasant talk,
Along the briny beach:
We cannot do with more than four,
To give a hand to each."

The eldest Oyster looked at him,
But never a word he said:
The eldest Oyster winked his eye,
And shook his heavy head—
Meaning to say he did not choose
To leave the oyster-bed.

But four young Oysters hurried up,
All eager for the treat:
Their coats were brushed, their faces washed,
Their shoes were clean and neat—
And this was odd, because, you know,
They hadn't any feet!

Four other Oysters followed them,
And yet another four;
And thick and fast they came at last,
And more, and more, and more—
All hopping through the frothy waves,
And scrambling to the shore.

*Continued on next page.*

"The Walrus and the Carpenter" from
"Songs from Alice in Wonderland and Through the Looking-Glass"

# 3.1

## *MIDI Effects*

In addition to the musical events discussed in the previous section, MIDI allows for a range of effects. For example, you can use the MIDI pitch wheel to make a note slide up and down, and you can use controller events to send special messages to MIDI controllers. You can specify these effects in Staccato using Functions.

If you're familiar with earlier versions of JFugue, the way MIDI effects are specified in this version differ because of the use of functions. In the past, various MIDI effects were specified by a variety of special characters – an ampersand (&) for pitch wheel, a plus (+) for channel pressure, and so on. These special characters had little resemblance to the effects that they represented, and were subsequentially difficult to use. The new Function capability provides an excellent mechanism for representing MIDI effects using meaningful names.

JFugue provides functions for the following MIDI effects:

- Pitch Wheel
- Channel Pressure
- Polyphonic Pressure
- Controller Events
- System Exclusive (SYSEX) Events

MIDI effects are discussed in the MIDI Specification. Please note that not all synthesizers (hardware *or* software) are capable of rendering all of these MIDI effects. If you don't hear these effects, check whether your synthesizer is capable of producing them.

## Pitch Wheel (aka Pitch Bend)

The pitch wheel can be used to create Theremin-like effects in your music, in which notes seem to slide within a range of ±8192 "cents", or 1/100ths of a Hertz, which musically corresponds to ±2 semitones. In addition to using the pitch wheel for sliding effects, JFugue uses the pitch wheel to make microtonal adjustments for notes, enabling some Eastern styles of music to be played easily (see also Section 3.2 for a discussion of microtones).

The function to adjust the pitch wheel for the following notes is `:PitchWheel`, `:PW`, `:PitchBend`, or `:PB`. The pitch wheel function can take two parameters, the least significant bit (LSB) and the most significant bit (MSB) of the pitch wheel change *or* a single parameter that is integer calculation equal to MSB*128+LSB. A value of `(0, 64)` (using LSB, MSB) or `8192` (using an integer) is the default value representing no change in the following notes' frequency.

Note that the pitch wheel applies to a channel (or what JFugue calls a voice). This means that you cannot play two microtonal notes in harmony in a single channel. However, pitch wheel changes can occur independently in separate voices, so microtones played in harmony can be created if each microtone is in a separate channel.

## Channel Pressure

Many MIDI devices are capable of applying pressure to all of the notes that are playing on a given channel. In JFugue, channel pressure is indicated by using the `:ChannelPressure` or `:CP` function, followed by a value from 0 to 127 (for example, `:CP(110)`). Channel pressure is applied to the channel (or voice) that is currently being played.

## Polyphonic Pressure

Polyphonic pressure, also known as key pressure, is pressure applied to an individual note. This is a more advanced feature than channel pressure, and not all MIDI devices support it. In JFugue, polyphonic pressure is indicated with the `:PolyPressure`, `:PolyphonicPressure`, or `:PP` function followed by the key value (i.e., the note value), specified as a value from 0 to 127, followed by a comma, and finally by the pressure value, from 0 to 127.

For example, the following function call applies a pressure of 75 to Middle-C (note 60): `:PP(60, 75)`. Note that this command does not accept note values, so using `C5` in this case would not work. (Ambitious software developers are invited to think about how JFugue's parsers could be adjusted to make that work.)

The difference between channel pressure and polyphonic pressure is that channel pressure applies equally to all of the notes played within a given channel, whereas polyphonic pressure is applied individually to each note within a channel.

## Controller Events

The MIDI specification defines a number of controller events that are used to specify a wide variety of settings that control the sound of the music. These include foot pedals, left-to-right balance, portamento (notes sliding into each other), tremulo, and so on. If you are interested in understanding more about controller events, you will definitely want to do research outside of JFugue, perhaps starting with the MIDI Specification.

The Controller Event function, `:ControllerEvent` or `:CE`, tells JFugue to set a value on the given controller. The function takes two values in parentheses: the controller name or number, and a value. For example, `:ControllerEvent(Chorus_Level,64)` will set the `Chorus_Level` event to 64. `:CE(93,64)` would do the same thing, since the `Chorus_Level` resolves to controller 93. A full list of the controllers is shown in Table 7.

If you're familiar with MIDI Controllers, you may know that there are 14 controllers that have both "coarse" and "fine" settings. These controllers essentially have 16 bits of data, instead of the typical 8 bits (one byte) for most of the others. There are two ways that you can specify coarse and fine settings.

The first way is quite uninspired – you can set a value on each controller. But if you have a 16 bit value in mind, you will have to break it up into a high and low byte value. For example, if you want to set the Foot Pedal to an overall value like 1345, you would say `:CE(Foot_Pedal_Coarse,10) :CE(Foot_Pedal_Fine,65)`.

Surely, JFugue can be smarter than this! Indeed it is: For any of those 14 controller events that have coarse and fine components, you can specify both values at the same time using `:CE(Foot_Pedal,1345)`. Isn't that nicer? Behind the scenes, JFugue will make the calculations necessary to figure out how much to give to the coarse and fine settings. Suppose you would like to set the volume to 10200, out of a possible range of 0 through 16383. Just use `:CE(Volume,10200)`. There is no need

for you to figure out the high byte and low byte of 10200; JFugue will split the values into high and low bytes for you. See Table 8 for the set of combined controller values.

Some controller events have on/off settings instead of a range of values. For these controllers, a value of 127 means "on" and a value of "0" means off. JFugue has defined two constants, `ON` and `OFF`, that you can use instead of the numbers, as in the expression `:CE(Local_Keyboard,ON)`. JFugue also defines the word `DEFAULT`, which has a value of 64.

| # | MidiDictionary Value |
|---|---|
| 0 | BANK_SELECT_COARSE |
| 1 | MOD_WHEEL_COARSE |
| 2 | BREATH_COARSE |
| 4 | FOOT_PEDAL_COARSE |
| 5 | PORTAMENTO_TIME_COARSE |
| 6 | DATA_ENTRY_COARSE |
| 7 | VOLUME_COARSE |
| 8 | BALANCE_COARSE |
| 10 | PAN_POSITION_COARSE |
| 11 | EXPRESSION_COARSE |
| 12 | EFFECT_CONTROL_1_COARSE |
| 13 | EFFECT_CONTROL_2_COARSE |
| 16 | SLIDER_1 |
| 17 | SLIDER_2 |
| 18 | SLIDER_3 |
| 19 | SLIDER_4 |
| 32 | BANK_SELECT_FINE |
| 33 | MOD_WHEEL_FINE |
| 34 | BREATH_FINE |
| 36 | FOOT_PEDAL_FINE |
| 37 | PORTAMENTO_TIME_FINE |
| 38 | DATA_ENTRY_FINE |
| 39 | VOLUME_FINE |
| 40 | BALANCE_FINE |
| 42 | PAN_POSITION_FINE |
| 43 | EXPRESSION_FINE |
| 44 | EFFECT_CONTROL_1_FINE |
| 45 | EFFECT_CONTROL_2_FINE |
| 64 | HOLD_PEDAL |
| 65 | PORTAMENTO |
| 66 | SUSTENUTO_PEDAL |
| 67 | SOFT_PEDAL |
| 68 | LEGATO_PEDAL |
| 69 | HOLD_2_PEDAL |
| 70 | SOUND_VARIATION |
| 71 | SOUND_TIMBRE |
| 72 | SOUND_RELEASE_TIME |
| 73 | SOUND_ATTACK_TIME |
| 74 | SOUND_BRIGHTNESS |
| 75 | SOUND_CONTROL_6 |
| 76 | SOUND_CONTROL_7 |
| 77 | SOUND_CONTROL_8 |
| 78 | SOUND_CONTROL_9 |
| 79 | SOUND_CONTROL_10 |
| 80 | GENERAL_PURPOSE_BUTTON_1 |
| 81 | GENERAL_PURPOSE_BUTTON_2 |
| 82 | GENERAL_PURPOSE_BUTTON_3 |
| 83 | GENERAL_PURPOSE_BUTTON_4 |
| 91 | EFFECTS_LEVEL |
| 92 | TREMULO_LEVEL |
| 93 | CHORUS_LEVEL |
| 94 | CELESTE_LEVEL |
| 95 | PHASER_LEVEL |
| 96 | DATA_BUTTON_INCREMENT |
| 97 | DATA_BUTTON_DECREMENT |
| 98 | NON_REGISTERED_COARSE |
| 99 | NON_REGISTERED_FINE |
| 100 | REGISTERED_COARSE |
| 101 | REGISTERED_FINE |
| 120 | ALL_SOUND_OFF |
| 121 | ALL_CONTROLLERS_OFF |
| 122 | LOCAL_KEYBOARD |
| 123 | ALL_NOTES_OFF |
| 124 | OMNI_MODE_OFF |
| 125 | OMNI_MODE_ON |
| 126 | MONO_OPERATION |
| 127 | POLY_OPERATION |

**Table 7. Controller values**

| Combined Value | MidiDictionary Value |
|---|---|

| | |
|---|---|
| 16383 | BANK_SELECT |
| 161 | MOD_WHEEL |
| 290 | BREATH |
| 548 | FOOT_PEDAL |
| 677 | PORTAMENTO_TIME |
| 806 | DATA_ENTRY |
| 935 | VOLUME |
| 1074 | BALANCE |
| 1322 | PAN_POSITION |
| 1451 | EXPRESSION |
| 1580 | EFFECT_CONTROL_1 |
| 1709 | EFFECT_CONTROL_2 |
| 12770 | NON_REGISTERED |
| 13028 | REGISTERED |

**Table 8. Combined controller constants. Integers can be assigned to these, and JFugue will figure out the high and low bytes.**

## System Exclusive Events

The MIDI specification provides system exclusive events for manufacturers and devices to use to their own unique settings. Again, for more information, please consult the MIDI specification. JFugue lets you specify System Exclusive events using the `:SysEx` or `:SE` function, which takes a list of comma-separated bytes as a parameter. There is not a specific good example here, but an expression similar to `:SysEx(34,56,24,32)` would be a valid use of the function. Several of the initial bytes may specify the manufacturer ID. There are also "Universal Exclusive Messsages" that are not specific to a particular manufacturer. More information should be available in both the MIDI specification and device-specific documentation.

# 3.2

# *Microtones*

Microtonal music is music in which the tuning is not based on twelve semitones (i.e., the frequency of each note is the 12th root of 2 greater than the previous note). It is popular in Indian classical music, Turkish music, and Indonesian gamelan music. Fortunately, Staccato provides a simple way to play microtones.

## Specifying Microtones in Staccato

To specify a microtone in Staccato, use the letter m followed by the frequency, in Hertz, of the sound you are interested in. For example:

```
Player player = new Player();
player.play("m512.3q");
```

This will play 512.3 Hertz at a duration of a quarter note.

To create the actual music that will be played through the MIDI system, Staccato converts the microtone to a series of MIDI Pitch Wheel and Note events. Recall that the MIDI Pitch Wheel is used to change the pitch of a note by hundredths of a half-step, or cents. The actual MIDI played by the line above is:

```
":PitchWheel(5192) 72/0.25 :PitchWheel(8192)"
```

An algorithm in the `MicrotonePreprocessor` class computes the pitch wheel and note that correspond to the desired frequency.

## Microtones and Musical Representation

It is likely that you may want to use microtones but not think of music as frequencies. For example, if you are playing Indian Carnatic music, it would be more natural for you to think of notes like S and R3 rather than microtones like 261.6256 and 290.6951. In the next section, you will learn about Replacement Maps, which will allow you to create a `Map<String, String>` that maps, for example, `S` to `m261.6256` and `R3` to `m290.6951`. Then, your `play()` statement would look like this:

```
player.play("<S>q <R3>q");
```

The `ReplacementMapPreprocessor` will convert this to microtones:

```
"m261.6256q m290.6951q"
```

And the `MicrotonePreprocessor` will convert this to MIDI commands:

```
":PitchWheel(8192) 60q :PitchWheel(8192) :PitchWheel(6750) 62q
:PitchWheel(8192)"
```

This line will then be sent to the `StaccatoParser` to be parsed and played by the `MidiParserListener`, which will receive the Pitch Wheel and Note events and make the desired music.

Keep in mind that not all MIDI Synthesizers implement the Pitch Wheel functionality. If you are not hearing the music that you expect, check to ensure that the synthizer you are using actually implements pitch bend or the pitch wheel. Additionally, recall that pitch wheel can be specified per channel (or voice). This means that you cannot play two microtones in harmony in a single channel since they depend on a different pitch wheel setting. However, you can play two microtones in harmony if they are played in separate channels.

# 3.3

## *Replacement Maps*

JFugue strives to let you express music the way you need to express music, while providing a means for converting those expressions to musical instructions that JFugue will understand. A good example of this is Indian Carnatic music, which uses different note names, like `S`, `R1`, and `M1`. The sounds that those notes should make are specific instances of microtones. Using Replacement Maps, you can tell JFugue to replace a note name like `S` with a microtone like `m261.62`.

### Creating a Replacement Map

A Replacement Map is really a simple `Map<String, String>`. The key for a map entry is the notation that you want to replace, and the value is what to replace it with. For example, here is a CarnaticReplacementMap, using the rarely-seen but nicely-succinct double-brace initialization feature in Java:

```
public class CarnaticReplacementMap extends HashMap<String,
String> {{
      put("S",  "m261.6256");
      put("R1", "m275.6220");
      put("R2", "m279.0673");
      put("R3", "m290.6951");
```

```
        put("R4", "m294.3288");
        put("G1", "m310.0747");
        put("G2", "m313.9507");
        put("G3", "m327.0319");
        put("G4", "m331.1198");
        put("M1", "m348.8341");
        put("M2", "m353.1945");
        put("M3", "m367.9109");
        put("M4", "m372.5098");
        put("P",  "m392.4383");
        put("D1", "m413.4330");
        put("D2", "m418.6009");
        put("D3", "m436.0426");
        put("D4", "m441.4931");
        put("N1", "m465.1121");
        put("N2", "m470.9260");
        put("N3", "m490.5479");
        put("N4", "m496.6798");
}}
```

Simply submit this to the `ReplacementMapPreprocessor`:

```
ReplacementMapPreprocessor.getInstance().setReplacementMap(
  new CarnaticReplacementMap());
```

Now, you can use any of the keys, placed between angle brackets, into your Staccato string, like the following:

```
Player player = new Player();
player.play("<S> <R1> <R2> <R3> <R4>");
```

The ReplacementMapPreprocessor will find all strings within angle brackets and replace them with the values from the map. (If the key is not found in the map, the key is placed back into the string). The output in this example is:

```
m261.6256 m275.6220 m279.0673 m290.6951 m294.3288
```

The `MicrotonePreprocessor` will then convert these microtones to Pitch Wheel and Note events so the music can be played with a MIDI synthesizer.

## Eliminating the Need for Angle Brackets

You just read that the ReplacementMapPreprocessor looks for strings within angle brackets, and it knows that the string inside those angle brackets are the keys for looking into the replacement map to find replacement values. The angle brackets are there on purpose to prevent you from inadvertently replacing things you did not mean to replace.

But let's take off the training wheels. Wouldn't it be nice to eliminate those angle brackets and just say:

```
Player player = new Player();
player.play("S R1 R2 R3 R4");
```

Guess what? You can! (Which I suppose you expected when you read the title of this section.) All it takes is a call to `setRequireAngleBrackets(false)` in `ReplacementMapPreprocessor`.

## Replacing Musical Tones with Solfege

One way of introducing music to new learners is to use solfege (do, re, mi) instead of note letters (C, D, E). JFugue provides a `SolfegeReplacementMap` that lets you use solfege in the place of note letters. This is how you would write music in solfege using JFugue:

```
ReplacementMapPreprocessor rmp =
     ReplacementMapPreprocessor.getInstance();
rmp.setReplacementMap(new
     SolfegeReplacementMap()).setRequireAngleBrackets(false);
Pattern pattern = new Pattern("do re mi fa so la ti do");
System.out.println(rmp.preprocess(pattern.toString().
     toUpperCase(), null));
```

When you run this code (also provided in the source code as `SolfegeReplacementMapDemo`), you will see that the solfege string is converted to `C D E F G A B C`.

This example specifies that angle brackets do not need to be provided. That's fine for a simple note example, but if you want to start adding things like duration to notes, you will need a way to separate the solfege from the durations. Otherwise, the parser won't know what to do with something like "`doq`" (quarter-duration Do) or "`mimini`" (Eighth-duration Mi Minor chord!). This is when you would want to use angle brackets. The brackets protect the string that will be converted to something else. And, the angle brackets are removed when the replacement is successful. "`<do>q`" will resolve to "`Cq`", not "`<C>q`" (unless "`do`" could not be found in the replacement map).

Continuing with the previous example:

```
rmp.setRequireAngleBrackets(true);
player.play(new Pattern("<Do>q <Re>q <Mi>h | <Mi>q <Fa>q <So>h |
<So>q <Fa>q <Mi>h | <Mi>q <Re>q <Do>h"));
```

## Creating Your Own Replacement Map

Remember, a replacement map is any `Map<String, String>`, where the key is what you place in your Staccato string and the value is what replaces the string as part of the replacement. There are no limits to what you can place in the keys, especially when you separate the keys from other musical elements with angle brackets. And really, there are no limits to what the values can be, with the only constraint being that if you want to play the string with `Player.play()`, it will need to be valid Staccato. In that case, the values can be any valid Staccato, not necessarily limited to a single token. You could, for example, create a replacement map that replaces strings with entire Patterns, or one that replaces some keys with valid Staccato but other keys with values that would be keys the second time you run a replacement. Which brings us to the next interesting idea about iterative replacement.

## Replace, Replace Again, Replace Again… Iterative Replacement

There is a little secret in the `ReplacementMapProcessor` that you can use do experiment with some fun musical effects: Replacements can be iteratively applied. The value of a replacement can contain keys that will be replaced the next time around.

For example, let's say your replacement map contains an entry like "C → C D". You then create a string consisting of "C". As we've used the `ReplacementMap` so far, you would run the code and get "C D" as your string. But you can send this back to the `ReplacementMap` again. This time around, you'll get "C C D."

You can loop for a number of times that you specify through the `setIterations()` method. By default, the preprocessor uses only one interation. Later in this book, you will find a cool example that shows how to use multiple iterations to create fractal music using a Lindenmayer system!

# 3.4

# *Functions*

JFugue provides a new capability that lets you express functions that can automate musical tasks that might otherwise be time-consuming or arduous, let you experiment with novel musical effects, or send musical events on your behalf. You have already been introduced to functions for specifying MIDI effects like `:Sysex` or `:ChannelPressure`. This section provides more details on the use of functions, including instructions on creating your own functions.

## Preprocessor Functions vs. Subparser Functions

There are two types of functions: *preprocessor functions*, which are converted to Staccato strings before any parts of the Staccato string are parsed, and *subparser functions*, which are parsed as the rest of the Staccato string is parsed. Both functions look the same when written in Staccato. However, there are differences in how the functions are programmed and how the functions act when they are called.

Use a preprocessor function when you want your function to use the parameters passed to it to generate replacement Staccato elements. Use a subparser function when you want your function to fire events to ParserListeners. See the sections below for more information.

## Preprocessor and Subparser Functions in JFugue

The StaccatoParser is responsible for creating the list of preprocessor and subparser functions that will be used to convert an incoming Staccato string to music. An extensible approach might use Java reflection to find all of the classes that are preprocessors or subparsers and pull those in automatically, but as the code is written currently, the list of processors is hard-coded. However, the `FunctionManager` class is designed for extensibility. You can get an instance of each of this object using its `getInstance()` method and add new preprocessor or subprocessor function. The order in which processors are added is also significant: processors listed first get to run before the others.

The list of preprocessors that the StaccatoParser uses, in order, is:

- `ReplacementMapPreprocessor` - see Section 3.3.
- `InstructionPreprocessor` - see Section 3.5.
- `UppercasePreprocessor` - converts *all* characters in the string to uppercase.
- `CollectedNotesPreprocessor` – expands sets of notes that are grouped together with parentheses, for example converting `(C E G)q` to `Cq Eq Gq`.
- `ParenSpacePreprocessor` – for any remaining items in parentheses, removes the parentheses and replaces spaces that occur within those parentheses with underscore characters
- `FunctionPreprocessor` – you're about to learn about this!
- `MicrotonePreprocessor` – see Section 3.4.

After running all of the preprocessors, the `StaccatoParser` then attempts to parse the string using the following subparsers, in order:

- `NoteSubparser` – parses notes
- `BarLineSubparser` – parses the bar line, `|`.
- `IVLSubparser` – parses instrument, voice, and layer commands
- `SignatureSubparser` – parses key signatures and time signatures
- `TempoSubparser` – parses the tempo command
- `BeatTimeSubparser` – parses the beat time, bookmark, and bookmark request commands
- `LyricMarkerSubparser` – parses lyrics and markers

- `FunctionSubparser` – parses subparser functions – read more ahead!

## Creating a Preprocessor Function

A preprocessor function will be converted to a Staccato string before the any of the Staccato string is sent to the StaccatoParser. An example of a preprocessor function might be one that plays an arpeggiated chord. Given a chord, each note will be played in harmony for a fraction of the total duration of the chord. Here is what this will look like in Staccato:

```
Player player = new Player();
player.play(":Arpeggiated(Cmajq)");
```

And this is the actual music that we would like to have played:

```
C3/0.08333333333333 E3/0.08333333333333 G3/0.08333333333333
```

Each element of the chord is played in melody for one-third of the overall duration.

Let's create an arpeggiated chord function that does this. You will find a class called ArpeggiatedChordFunction in the JFugue API (it's in org.staccato.functions), but here you will learn how it was made.

First, the class needs to implement the PreprocessorFunction interface:

```
public class ArpeggiatedChordFunction implements
PreprocessorFunction {
```

Next, each function (whether it is a preprocessor function or a subparser function) is intended to be a singleton that maintains no state; so, it should have a static `getInstance()` method and a private constructor:

```
    private static ArpeggiatedChordFunction instance;

    private ArpeggiatedChordFunction() { }

    public static ArpeggiatedChordFunction getInstance() {
        if (instance == null) {
            instance = new ArpeggiatedChordFunction();
        }
        return instance;
    }
```

Then, a `PreprocessorFunction` needs an `apply()` method. For parameters, it takes two things: the string contained within the parentheses of the function call, and the `StaccatoParserContext`, which contains information about the context of the parser. This includes things like dictionary definitions. The `PreprocessorFunction's apply()`

method returns a String that contains the new Staccato string that will be used to replace this function call. This is different from a SubparserFunction, in which the `apply()` method is a void.

For an arpeggiated chord function, we'd like to get the chord from the parameter, use the Chord API to get the notes, compute new durations for each note, and create a Staccato string that would play those notes.

```
	@Override
	public String apply(String parameters,
StaccatoParserContext context) {
		Chord chord = new Chord(parameters);
		Note[] notes = chord.getNotes();
		double duration = chord.getRoot().getDuration();
		double durationPerNote = duration / notes.length;

		StringBuilder buddy = new StringBuilder();
		for (Note note : notes) {
		    buddy.append(Note.getToneString(note.getValue()));
		    buddy.append("/");
		    buddy.append(durationPerNote);
		    buddy.append(" ");
		}

		return buddy.toString().trim();
	}
```

Finally, each function needs a name. In fact, functions can respond to several names. This is intended to allow long-form names (e.g., "ControllerEvent") for readability and clarity, and shorter abbreviations (e.g., "CE") for brevity. These should be written in all capital letters, since the UppercasePreprocessor will come through and turn all characters in the Staccato string to uppercase, even though Staccato itself is not case-sensitive.

```
	@Override
	public String[] getNames() {
		return new String[] { "ARPEGGIATED", "AR" };
	}
```

And you're done! Close up the class with an ending brace, and you've got an `ApreggiatedChordFunction`!

To use this function, you need to register an instance of the `ArpeggiatedChordFunction` it with the `FunctionManager`:

```
FunctionManager.getInstance()
.addPreprocessorFunction(ArpeggiatedChordFunction.getInstance());
```

Now you can call that `play()` statement from before, and you'll hear an arpeggiated chord!

## Creating a Subparser Function

A subparser function is conceptually very similar to a preprocessor function, except it happens *as* the Staccato string is parsed instead of *before*. This means that for a subparser function to have an effect, it needs to fire musical events to `ParserListener`s. Keep in mind that these events would be fired as the music is *parsed*, rather than as the music is *played*. Played music has already been fully parsed. (But you can use the `TemporalPLP` to respond to parser events while – or before or after, given some offset in milliseconds – music is playing. See Section 5.3 to learn about `TemporalPLP`). A `ParserListener` will receive an event from `onFunctionParsed()` before the function is applied.

You have aready seen subparser functions if you have read the section on MIDI Effects. All of those effects, such as `:ControllerEvent` or `:PitchWheel`, are expressed as subparser functions.

Let's create a new subparser function that insert a special marker in the MIDI sequence to, say, control a light display as the music plays. We will assume that you have a separate light display (maybe an Arduino sketch?) that listens to the MIDI and waits for marker meta-messages (0x06). While JFugue already lets you specify a marker using the `#` character (as described in Section 2.6), we will use this as an example for creating a new subparser function.

Here is an example of what we want to express in Staccato:

```
Player player = new Player();
player.play(":Lights(RED)");
```

First, the subparser function needs to implement the SubparserFunction interface:

```
public class LightsFunction implements SubparserFunction {
```

Like the preprocessor function, the subparser function is also intended to be a singleton, so the following code should look familiar:

```
    private static LightsFunction instance;

    public static LightsFunction getInstance() {
        if (instance == null) {
            instance = new LightsFunction();
        }
        return instance;
    }
```

```
    private LightsFunction() { }

    @Override
    public String[] getNames() {
        return NAMES;
    }

    public static String[] NAMES = { "LIGHTS" };
```

The `apply()` function in a subparser function is different than in a preprocessor function. Here, the `apply()` function does not return a value. Instead, it is expected to fire an event through the parser.

```
    @Override
    public void apply(String parameters, StaccatoParserContext
context) {
        String[] params = parameters.split();
        if (params.length == 1) {
           context.getParser().fireMarkerParsed(params[0]);
        }
    }
```

That's all there is to creating your own subparser function. Like the preprocessor function, you need to add this through the FunctionManager:

```
FunctionManager.getInstance()
.addSubprocessorFunction(ArpeggiatedChordFunction.getInstance());
```

## Subparsers May Populate Context

Populating context is not a requirement for classes that implement `Subparser`, but if you create a new subparser, you may want to create `public static void populateContext(StaccatoParserContext context)` if you wish for your subparser to place its own dictionary definitions and other information within the context object. `IVLSubparser` does this to populate the word `PERCUSSION` and the names of instruments, `TempoSubparser` does this to place definitions of words like `ADAGIO`, and `NoteSubparser` does this to populate the names of percussion notes. If you want to call your own `populateContext()` method in a new subparser, you will have to edit the `StaccatoParser` constructor.

# 3.5

## *Instructions*

One of the important features of Staccato is that the music string is readable to humans as well as understandable to a computer program. As you saw in previous sections, especially the section on MIDI Controller Events, there are a lot of things that you can express in a Staccato string aside from basic musical notes. Unfortunately, these do not lend themselves to easy readability. If you see a Staccato string with the command `:CON(7,2)`, can you immediately identify the purpose of this musical event and the effect that the event is going to have on the music? My guess is that you would probably have to look up what MIDI Controller Event 7 is, and what a value of 2 does to the event. But you shouldn't have to do that. Enter Staccato Instructions. (Controller 7 is Coarse Volume, by the way).

### Understanding Instructions

Instructions in Staccato let you replace this:

```
player.play(":CON(65,127) :CON(5,70) C5h D5h E5h");
```

With this:

```
player.play("{turn portamento on} {set portamento time to 70} C5h
D5h E5h");
```

Personally, I'm a huge fan of that second line. Now when you read this music in the future, or when someone else sees your music, your intentions are quite clear, and it is easy to see what the Staccato string is trying to do.

Instructions work for any musical command, but MIDI controller events provide the best example (and probably the best use case) since there are so many of them and activating controller events by MIDI values alone is cryptic.

One downside of using instructions is that instructions are pieces of code, whereas Staccato strings are not code-specific objects. In an ideal world, you should be able to share Staccato strings with other people and they should be able to play that music with no code changes. Of course, since the interpretation of instructions is based on code that is not strongly connected to the Staccato string itself, music with instructions may not sound as the author intended if the instruction code is not also shared. The Staccato parser will ignore instructions that it does not know, so the music will at least be playable, but not with the adjustments that the instructions would have made.

In processing the Staccato string, Instructions are *preprocessed*, which means that the instruction is re-written as valid Staccato commands before the Staccato string is parsed. This means that `ParserListener`s will not know about Instructions, and there is no `onInstructionParsed()` method in `ParserListener`.

## Creating Instructions

There are several types of instructions that you can create depending on the effect you would like to produce.

### Simple Instructions

The first type of instruction is a simple substitution: When the parser sees A, replace it with B. In this case, A and B are the two parameters given to the `addInstruction()` method, like so:

```
InstructionPreprocessor ip =
InstructionPreprocessor.getInstance();
ip.addInstruction("when you see this", "replace it with this");
```

When you create a Staccato string that looks like this:

```
"{when you see this}"
```

You will get output like this:

```
"replace it with this"
```

Which is much more interesting if your key and value are musical:

```
ip.addInstruction("pump up the volume", ":CON(7,127)");
```

This capability is not significantly different from Replacement Maps, although the implementation is slightly different. Replacement Maps are purely mappings between a String key and a String value; Instructions are a mapping between a String key and an Instruction value, and in this case, the `InstructionPreprocessor` wraps a simple String into an anonymous Instruction that just returns the String. Instructions become more interesting when using actual Instructions, as you will see in the next several sections.

### Switch Instructions

There are some musical instructions that you just want to turn on and off. The portamento and vibrato effects are two examples of this. Usually, a value of 127 turns on a MIDI controller event, and a value of 0 turns it off, but this is not always necessarily the case.

Let's take a look at portamento for our example. You'd like to be able to say:

```
player.play("{portamento on} C5h D5h E5h");
```

You'll need to add an `Instruction` that keys on the phrase "portamento", then provide a snippet of valid Staccato code that includes a dollar sign to represent where a value should be placed, then two values: the off value and the on value for the switch. When a switch instruction is processed, the processor looks for the word "on" or "off", selects the appropriate value, and places that value where the dollar sign is in the Staccato string. The first value provided as a parameter to the Switch is the "off" value; the second value is the "on" value.

```
InstructionPreprocessor ip =
InstructionPreprocessor.getInstance();
ip.addInstruction("portamento",
    new Instruction.Switch(":CON(65,$)",0,127));
```

### Choice Instructions

A choice instruction lets you use one set of words as a key to specific set of choices, then the last value provided in the instruction is a number representing the choice of result you would like to use. This is useful when you have different pieces of music that all belong to the same category—say, a drum beat and its several variations, or different ways of

singing a chorus for a given song. You may think of think of these as different variations of a theme.

To create this type of instruction, provide an `Instruction.Choice` object that is constructed with each of the values you would like to choose among. You can also use the `getChoices()` method to get the `List<String>` of choices, to which you can add new choices directly. The number you pass into the instruction itself will be the index into the list of the thing to play. For example:

```
InstructionPreprocessor ip =
InstructionPreprocessor.getInstance();
ip.addInstruction("trio", new Instruction.Choice("c g e", "d f
a", "g b e"));
player.play("{trio 0}"); // Will play "c g e"
player.play("{trio 1}"); // Will play "d f a"
player.play("{trio 2}"); // Will play "g b e"
```

### Instructions in which the Last Word or Number is a Value

Sometimes, you may just want the final value of an instruction to be the parameter for a Staccato command, like a MIDI controller event, the duration of a note, the selection of an instrument, and so on. In this case, the last value passed to the instruction—that is, the text just before the final curly brace, and after the final space—is used to replace a dollar-sign character in the text you provide to the constructor of `Instruction.LastIsValue`.

Note that for any of these instructions, the key is triggered if the instruction starts with the specified key; words between the key and the final value are ignored. This can be useful for additional commentary or just goofing off, or especially for including words like “to” (or “a”/“an”) that make the instruction sound or read better.

```
// This will play ":CON(7,2) Cq I[Piano] Eq"
InstructionPreprocessor ip =
    InstructionPreprocessor.getInstance();
ip.addInstruction("volume", new
    Instruction.LastIsValue(":CON(7,$)"));
ip.addInstruction("change instrument", new
    Instruction.LastIsValue("I[$]"));
player.play("{volume should now be set to the glorious value of
2} Cq {change instrument to Piano} Eq");
```

### Last Value Instructions in which the Value is Split over Multiple Commands

There may be times when you want to take a single value provided to an Instruction, do some processing on that value, and use the processed values as parameters for multiple Staccato commands. The primary use case for this, which is demonstrated below, is specifying a value for

controller events that have low-byte and high-byte values, but I would argue this isn't the best example to use because the `:ControllerEvent` subprocessor function has a more elegant way to deal with low-byte/high-byte controller events. Nevertheless, I will proceed with the example, although I encourage you to think of more creative applications of this capability.

There are a number of MIDI capabilities that are controlled my multiple controller events. For example, "volume" is really a combination of Coarse Volume (Controller 7) and Fine Volume (Controller 39). But you may want to think about volume as a number between 0 and 16384 rather than a combination of 128 degrees of coarse volume and 128 degrees of fine volume. To do this, you can create a slightly more complicated instruction object called `Instruction.LastIsValueToSplit`, which takes a number of Staccato commands that the instruction will replace—each with a specific string that will be replaced by the value or by the result of a computation on that value—and an instance of `Instruction.Splitter`. The Splitter's `splitInstructionParameter()` method takes the value provided in the instruction and returns a `Map<String, String>` in which each replacement string is the key, and the value to replace is the value.

At this point, you probably either want to read that last paragraph again or just declare that you'll never need to do anything this complex. Let me try to break it down differently:

1. Start with a single capability that requires multiple commands. We'll use "full volume" as an example.

2. For "full volume", we'll accept values of 0 through 16384. For any given value, MIDI Controller 7 (Coarse Volume) will be set to `value/128`, and MIDI Controller 39 (Fine Volume) will be set to `value%128` (that's modulo).

3. We'll need in `Instruction.LastIsValueToSplit` to do this. The instruction will be keyed by the phrase, "set total volume". That phrase will be replaced with the following Staccato snippet: `":CON(7,$1) :CON(39,$2)"`

4. We need the `Instruction.Splitter` to replace the `$1` with `value/128`, and the `$2` with `value%128`. (We could use any string, but strings that start with dollar signs seem like something you would not have in the rest of the Staccato command.)

5. When `Instruction.Splitter` is asked to make its calculations, it needs to return a `Map<String, String>`, and these will be its two entries: (`$1` → `value/128`) and (`$2` → `value%128`). (And, since the

values need to be `String` objects, those integer results (actually, they should be bytes) need to be converted to `String`s.)

Hopefully, that makes this capability more clear! Let's see this in action.

```
// This will play ":CON(7,6) :CON(39,64) C D E"
InstructionPreprocessor ip =
InstructionPreprocessor.getInstance();
Instruction.Splitter splitter = new Instruction.Splitter() {
     public Map<String, String> splitInstructionParameter(String
value) {
           Map<String, String> retVal =
               new HashMap<String, String>();
           retVal.put("$1", ""+Integer.parseInt(value) / 128);
           retVal.put("$2", ""+Integer.parseInt(value) % 128);
           return retVal;
     }
};
ip.addInstruction("set total volume", new
    Instruction.LastIsValueToSplit(":CON(7,$1) :CON(39,$2)",
    splitter));
player.play("{set total volume to 1600} C D E");
```

## Using Patterns as Instructions

You may also use a `Pattern` (or anything that implements `PatternProducer`) as an instruction. This allows you to provide a string name that is mapped to something that generates a pattern. One example that demonstrates the utility of this is specifying the structure of a song, where each Instruction can be turned into a set of notes. For example, you can have this string:

```
"{intro} {verse} {chorus} {verse} {chorus} {breakdown} {verse}
{outro}"
```

Then, you could define a set of Patterns for each part of the song—and the content of those Patterns could be different each time you run your program. You would then create a couple of simple instructions as so:

```
InstructionPreprocessor ip =
InstructionPreprocessor.getInstance();
ip.addInstruction("intro", introPattern);
ip.addInstruction("verse", versePattern);
ip.addInstruction("chorus", chorusPattern);
ip.addInstruction("breakdown", breakdownPattern);
ip.addInstruction("outro", outroPattern);
```

Now when you parse the structure of your song, the InstructionPreprocessor will fill in all of the instructions with the patterns you specified! This seems easier and more intuitive than

creating a new pattern by adding together other patterns. And if you want to change the structure of your song, it's quite easy to do—just change that one line describing the structure.

## Creating Your Own Instructions

In addition to the variety of instructions that JFugue provides out of the box, it is easy to create your own instruction. Simply create a class that implements Instruction and overrides the `onInstructionReceived(String[] instructions)` method.

One use of this would be instructions for which you plan to do your own parsing. For example, suppose you aree writing a parser for Indian Carnatic music, and you would like to specify both the note and the octave for a particular sound. If you simply use a Replacement Map to replace Carnatic notes with a microtone frequency, you will only be able to replace one octave. Instead, you could create an Instruction that takes the note and the octave, like `{R1 3}` for R1 in the 3rd octave. You would then be able to create a microtone frequency specific to this combination.

As another example, suppose you are interested in creating music that incorporates segments of DNA converted to musical instructions. You could say something like this:

```
"{intro} {DNA gacagattacagcgatgatg} {chorus} {outro}"
```

Your DNA Instruction, which would key on the string "DNA" and which would receive the DNA sequence through `onInstructionReceived()`, could then do computations on the DNA sequence and put the resulting music back into the Staccato string.

## Dealing with Instructions that Start with the Same Words

Suppose you have two instructions:

1. `{the greatest instruction}`
2. `{the greatest instruction to ever}`

And in your `play()` statement, you have this:

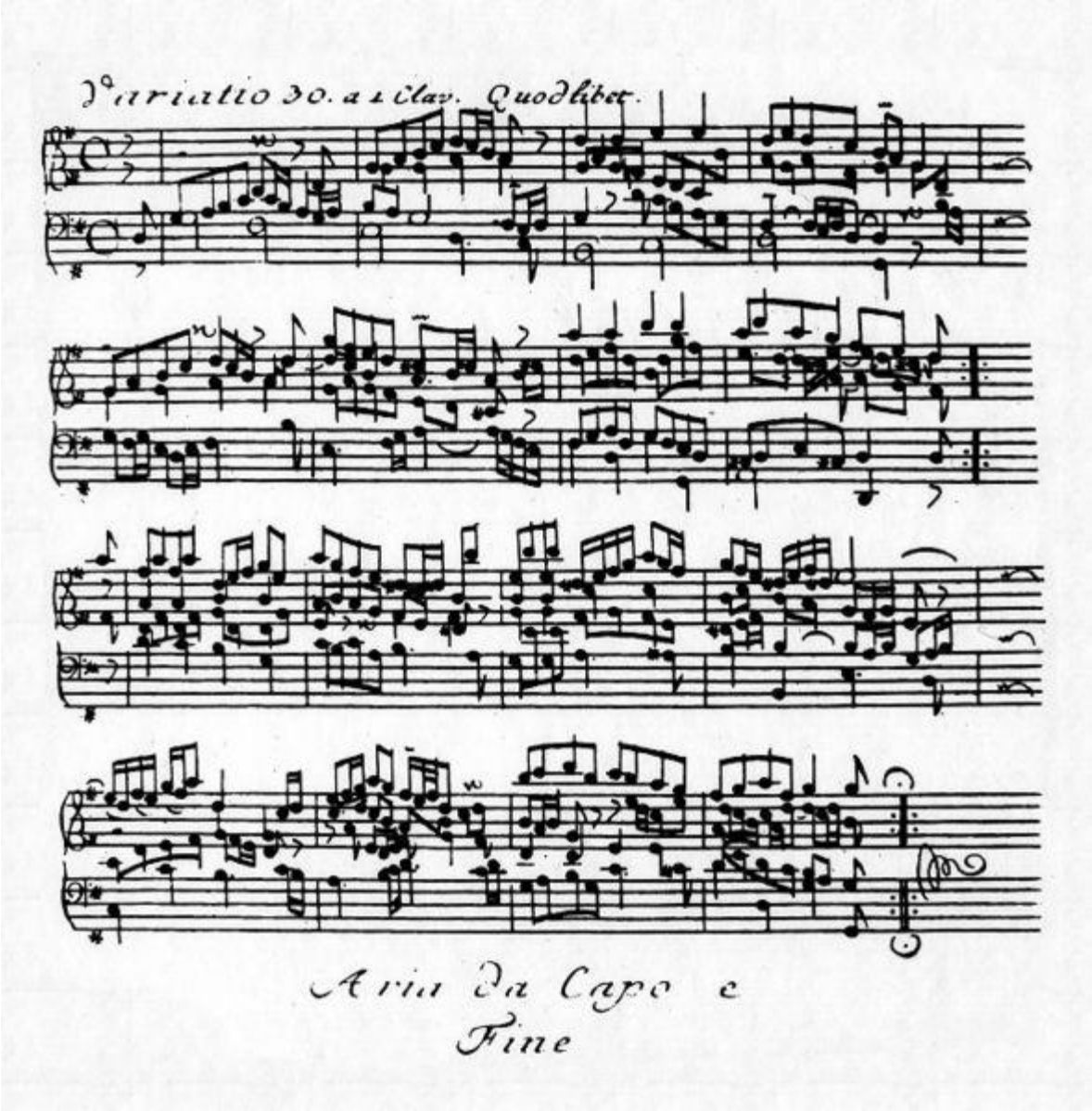
```
player.play("{the greatest instruction to ever exist}");
```

Which of the two instructions do you think will be triggered?

The Instruction preparser looks for instructions starting with the longest name first. In the example above, Instruction #2 would be found first, and it would be the instruction that is triggered here. Yes, Instruction #1 starts with the same phrase, but it is shorter in length and, since the longer instruction matches first, Instruction #1 will not be triggered.

# Part 4

# Music Theory with the JFugue API




Variation 30 of Johann Sebastian Bach's "Goldberg Variations" (BWV 988)

# 4.1

## *Introduction to Music Theory in JFugue*

In the previous sections, you have learned how to program music directly into JFugue using Staccato. JFugue also provides a set of classes for working with music theory. Even if you do not create music with JFugue, you may find the music theory classes useful for your application.

The music theory capabilities in JFugue are interconnected. A *key* can be specified as a set of *intervals*. A *scale* contains a set of *intervals*. A *key* can be made from a *scale*, and a *chord* can be made from a *key* or set of *intervals*. A *chord progression* specifies a set of *chords*. And, of course, *chords* are composed of *notes*. All of these levels of music theory work together seamlessly in JFugue.

In addition, all of these capabilities work in accordance with common music notation. If you have worked with intervals before, you know that, for example, a minor interval is represented like `"1 b3 5"`. If you have worked with chord progressions, you may be familiar with the "ii-V-I" turnaround. JFugue's `Intervals` and `ChordProgression` classes let you use that exact notation. For example:

```
Intervals intervals = new Intervals("1 b3 5");
```

```
ChordProgression progression = new ChordProgression("ii-V-I");
```

In fact, chords in JFugue are defined using Intervals. For example, the minor chord that you get when saying `Player.play("Cmin")` is defined in `Chord.java` as `Intervals("1 b3 5")`.

Similarly, a scale is defined using intervals, a key can be created from a scale and a root note, and a chord can be created from a key and vice versa—which is somewhat non-traditional but interesting—as well as from intervals.

There are additional capabilities as well: chords can be inverted, intervals can be rotated, and chords, chord progressions, and keys can be given root notes. And importantly, all of these implement `PatternProducer`, the interface in JFugue that indicates that a class has a method called `getPattern()`, which means that all of these elements can be added to a JFugue Pattern, or even sent directly to `player.play()`.

Here is a complete one-line program that plays a "ii-V-I" chord progression in the key of E Minor:

```
new Player().play(new ChordProgression("ii V I").setKey("Emin"));
```

Probably the greatest thing about JFugue's music theory classes is that they let you write programs that use musical elements as top-level objects. In other words, you can write programs that talk directly about chords and chord progressions and keys and scales and intervals by referring to those things directly, instead of referring to something like arrays and strings and other data types that might encode musical elements but do not lend themselves to writing clear, concise musical programs.

**Author's note:** When I wrote JFugue 5—putting aside previous versions of JFugue and starting from an entirely clean slate—the first class I wrote was `Intervals`. These music theory classes are fundamental to the way you can work with and express music using the JFugue API. I'm excited to bring these capabilities to you in JFugue 5. Enjoy!

# 4.2

## *Notes*

The fundamental musical element is the note. At its most basic level, a note represents a tone, like Middle-C. One step away from the basics and we can have notes with duration; a further step and we can have notes with dynamics like on and off velocities. As a sequence, notes create a melody. In parallel, notes create harmony. This chapter will explain notes in more detail.

### Staccato vs. the JFugue API

In previous sections, you learned how to create notes in JFugue through Staccato. If you are writing music in JFugue, you will find it much more expedient to specify notes using Staccato strings instead of, say, creating `Note` objects for each note that you would like to add to a song. Behind the scenes, those Staccato strings are being turned into `Note` objects anyway.

But you might also want to use notes in your own programs. For example, you may be interested in getting all of the notes for a chord progression. Or you may be interested in writing programs, JFugue parsers, or parser listeners that deal with `Note` objects. If so, this section is for you!

JFugue takes advantage of the Staccato notation to let you define a Note object. You can create a Note using a Staccato string, like this:

```
Note note = new Note("C"); // Middle-C
```

(This works by parsing the string using the JFugue's `NoteSubparser`)

You can also specify a note using its MIDI value:

```
Note note = new Note(60); // Also Middle-C
```

If you read the section on notes in Staccato, then you have already seen the following table:

| Octave | C | C# | D | E*b* | E | F | F# | G | G# | A | B*b* | B |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| **1** | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| **2** | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 |
| **3** | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 |
| **4** | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 |
| **5** | **60** | 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 | 70 | 70 |
| **6** | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 | 80 | 81 | 82 | 83 |
| **7** | 84 | 85 | 86 | 87 | 88 | 89 | 90 | 91 | 92 | 93 | 94 | 95 |
| **8** | 96 | 97 | 98 | 99 | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 |
| **9** | 108 | 109 | 110 | 111 | 112 | 113 | 114 | 115 | 116 | 117 | 118 | 119 |
| **10** | 120 | 121 | 122 | 123 | 124 | 125 | 126 | 127 | | | | |

**Table 9. MIDI note values.**
**Middle-C is marked in green on the left side if the table.**

You can do all sorts of things with a note like this – you can play it with `player.play`, you can add it to a Pattern, you can use it as a root for a chord, and so on.

The numeric value for a note can be obtained using `getValue()` and, of course, there is a corresponding `setValue(byte value)` method. There is also a `changeValue(int delta)` method, which takes an amount, either positive or negative, by which to change the current value.

## Default Octave on Notes

In JFugue, notes can serve two purposes. They can be an actual note that you expect to play – e.g., `player.play("C")` – or they can be a logical note used in a manner consistent with music theory – e.g., `chord.setRoot("C")`. In the first case, the note needs to have a default octave; in the second case, the octave may not matter. The `Note` class keeps track of whether an octave has been explicitly set for note. You mostly only need to know this if you are ever diagnosing whether two notes are the same.

To demonstrate, notice that these two lines are false:

```
new Note("C").equals(new Note("C4")); // Returns false!
new Note("C").equals(new Note("C").setOctave(4)); // False!
```

But these two lines both return the same value:

```
new Note("C").getOctave();  // Returns 4
new Note("C4").getOctave(); // Also returns 4
```

So what is the difference? Here it is:

```
new Note("C").isOctaveExplicitlySet();  // Returns false
new Note("Cq").isOctaveExplicitlySet(); // Returns true
```

There is a similar discussion about default durations, so let's turn there for further explanation about why this is true.

If you are making a new note based off an existing note, and you want the two notes to share the same octave settings, use `useSameExplicitOctaveSettingAs(Note existingNote)`.

## Default Duration on Notes

If you play the note, you'll hear the note that you expect. But you may also notice that the note has an end (it does not play forever), and this is because the note has a default duration, set to a quarter duration. It is worth knowing how the default is implemented, because a note with a default duration is slightly different than a note with a duration that has been explicitly set, even if the set duration is 0.25.

To demonstrate, notice that these next two lines are false:

```
new Note("C").equals(new Note("Cq")); // Returns false!
new Note("C").equals(new Note("C").setDuration(0.25d)); // False!
```

But these two lines both return the same value:

```
new Note("C").getDuration(); // Returns 0.25
new Note("Cq").getDuration(); // Also returns 0.25
```

So what is the difference? Here it is:

```
new Note("C").isDurationExplicitlySet(); // Returns false
new Note("Cq").isDurationExplicitlySet(); // Returns true
```

And now, why is this important?

The importance is that it allows us to work with notes, and even play the notes, and take advantage of the note's default duration, while not polluting the other music theory elements with a duration if one has not been explicitly specified.

For example, in the next section you will learn about chord progressions, and you can provide a chord progression with a root note, then get the pattern represented by the chord progression. To get the pattern, JFugue generates a string representing each note, and if the `Note` class returned whatever duration was associated with the note, including a default duration, then this:

```
ChordProgression cp = new ChordProgression("iv v i");
Pattern pattern = cp.setKey("Cmaj").getPattern();
```

...would return `F4minq G4minq C4minq` despite that we did not use `q` or octave `4` in the key. Instead, we would like to get back `Fmin Gmin Cmin`, because then we can what we want with the elements in the pattern without dealing with an extraneous duration that had not been specified. But we still want to be able to *play* the chord progression, even if they key we provided did not contain a duration.

If you happen to be creating a note and you want the new note to reflect the same duration as an existing note, and whether the duration was explicitly set, please use the `useSameDurationAs(Note existingNote)` method.

## "Note On" and "Note Off" Velocities

Fortunately, there is less to say about velocities than there is about durations! The Note class has getters and setters for "note on" and "note off" velocity. When the note is being represented as a pattern, velocity is not added to the string if they are the same as the default values, which are defined as `MidiDefaults.MIDI_DEFAULT_ON_VELOCITY` and `MidiDefaults.MIDI_DEFAULT_OFF_VELOCITY`, unless you change those settings with `DefaultNoteSettingsManager` (which will be discussed a bit later in this section).

## Note States: isRest, isPercussionNote

A note has a lot in common with a rest: they both have duration, they can both be harmonic or melodic. For this reason, the `Note` class is also used to represent rests. If a note is a rest, `setRest(boolean)` and `isRest()` should be called as necessary. To easily create a Note that is a rest, use the static method `Note.createRest(double duration)` or the `Note.REST` constant, which provides a rest with default duration.

A note can also represent a percussion instrument when used in the 10th MIDI channel (JFugue's `V9`). While these are truly notes, the `Note` class keeps track of whether a note is a percussion note so it knows how to represent that note in a string:

```
new Note(56).getPattern(); // Returns "G#3"
new Note(56).setPercussionNote(true).getPattern(); // "[COWBELL]"
```

## Melodic and Harmonic Notes

Aside from belonging to a chord, a note can be part of a harmony. For example, in Staccato you can say `E+G` to play those two notes together. This is represented by the Note class with two separate instances of Note. In the first instance, the note is marked as a first note (so `isFirst()` returns true) and both notes are marked as harmonic (so `isHarmonic()` returns true). When a parser or parser listener that manages time comes across first notes, it knows to remember the time that the note sounds; and when it comes across subsequent harmonic notes, it knows to move the time pointer back to when the first note started, so these two (or more) notes can occur at the same time.

Within a harmony, it is possible to have some notes that are melodic—for example, the G and C in `Eh+Gq_Cq` are melodic. In this case, E is the first note, G is a harmonic note, and C is a melodic note.

## Start and End of a Tie

A note can indicate the start of a tie, in which case it is meant to keep playing until an end note of the same tone is encountered, or it can be the end of a tie, which ends a tie that started with a note of the same tone. The Note class provides getters and setters for indicating that a note is the start or end of a tie.

## Position in Octave

`getPositionInOctave()` essentially returns `tone_value % 12` and lets you know that no matter what note and octave has been specified, a C is in position 0, C# is 1, and so on through B, which is position 11.

If you would prefer to use a constant in your code instead of the number 12 to represent the number of notes in an octave, you can use `Note.OCTAVE`, which is a final public int set to 12. You can also use `Note.MIN_OCTAVE` and `Note.MAX_OCTAVE` (0 and 10, respectively) in your code.

## String Arrays for Notes

The Note class contains several public static String arrays that you may find useful in your programming. The `NOTE_NAMES_COMMON` array contains string representations of each of the twelve notes in an octave:

```
public final static String[] NOTE_NAMES_COMMON = new String[] {
"C", "C#", "D", "Eb", "E", "F", "F#", "G", "G#", "A", "Bb", "B"
};
```

There are two additional arrays, `NOTE_NAMES_SHARP` and `NOTE_NAMES_FLAT`, both of which use only sharps or only flats respectively in the accidental notes.

```
public final static String[] NOTE_NAMES_SHARP = new String[] {
"C", "C#", "D", "D#", "E", "F", "F#", "G", "G#", "A", "A#", "B"
};
public final static String[] NOTE_NAMES_FLAT = new String[] {
"C", "Db", "D", "Eb", "E", "F", "Gb", "G", "Ab", "A", "Bb", "B"
};
```

The `PERCUSSION_NAMES` array contains a list of strings representing percussion sounds. These are the same strings that the `NoteSubparser` uses to interpret percussion names used in Staccato music. For example, you'll find that `PERCUSSION_NAMES[2]` is `SIDE_STICK`. Just remember to add 35 to the array index to match the MIDI note value of that percussion sound – for example, `SIDE_STICK` corresponds to MIDI Note 37.

## Sorting Notes

Suppose you would like to sort an array of `Note` instances – maybe by their duration, or their octave, or their place within an octave, or any other feature of the notes. The `Note` class provides a static method, `Note.sortNotesBy()`, that takes an array of `Note` instances and an instance of `Note.SortingCallback`, which is an interface with one method, `getSortingValue()`, that returns the value of the property that you wish to sort on.

`Chord.getChordFromNotes(Note[] notes)` makes use of this feature to identify which note in an array of notes should be the bass note for the chord. Here is the code:

```
Note.sortNotesBy(notes, new Note.SortingCallback() {
    @Override
    public int getSortingValue(Note note) {
        return note.getValue();
    }
});
```

In this case, the notes are sorted on their numeric value, which equates to their MIDI number (where Middle C is 60). The `sortNotesBy()` method uses a standard BubbleSort algorithm to arrange the notes based on the value returned by `getSortingValue()`.

Here is the list of the getters in the Note class that you might consider using for your own sorting purposes:

```
getDuration()
getMillisecondDuration()
getOctave()
getOffVelocity()
getOnVelocity()
getPositionInOctave()
getValue()
```

## Getting Strings and Other Values About Notes

There are a number of ways to generate strings from a Note, and various parts of the JFugue codebase all depend on at least one of these. You should have access to these, too!

- `getOriginalString()` returns the actual string that was used to create the Note.
- `getPattern()`, which is an implementation of the PatternProvider interface, creates a Pattern which is composed of `toStringWithoutDuration()` concatenated with `getDecoratorString()`.
- `getPercussionPattern()` first checks if the Note's value is within the range of `MidiDefaults.MIN_PERCUSSION_NOTE` and `MidiDefaults.MAX_PERCUSSION_NOTE`. If the note value is outside of this range, this method returns `getPattern()`. Otherwise, this method returns a concatenation of the value of the static method `Note.getPercussionString(getValue())` and `getDecoratorString()` as a Pattern.
- `Note.getPercussionString(int value)` is a static method that returns the name of a percussion instrument (see code two pages ago) in square brackets. For example, passing a value of 50 would result in the String `[HI_TOM]`. If you have a friend named Tom, you can probably think of an especially cool use for this function.
- `getDecoratorString()` returns a concatenation of `Note.getDurationString()` if `isDurationExplicitlySet()` returns true, and `getVelocityString()`.
- `getVelocityString()` returns only the velocity portion of the Note. The String contains the on velocity only if the value is not set to

the default, and the off velocity only if the value is not set to the default. If both are set to the default, this will return an empty string.

- `getToneString()` returns a String that starts with the static `Note.getToneStringWithoutOctave(getValue())` and only adds an octave if `isOctaveExplicitlySet()` is true.

- `Note.getToneString(byte noteValue)` is a static method that returns a concatenation of `Note.getToneStringWithoutOctave(noteValue)` and `noteValue` divided by `Note.OCTAVE`.

- `Note.getToneStringWithoutOctave(byte noteValue)` is a static method that returns `NOTE_NAMES_COMMON[ noteValue % Note.OCTAVE ]`.

- `toString()` returns `getPattern().toString()`.

- `toStringWithoutDuration()` returns: `R` if the note is a rest; `Note.getPercussionString(getValue())` if `isPercussionNote()` is true, and otherwise, `getOriginalString()` if the original string is set or `Note.getToneString(getValue())` if it is not.

- `Note.getDispositionedToneStringWithoutOctave(int dispose, byte noteValue)` is a static method that uses the dispose value to determine whether to pull a tone string from `NOTE_NAMES_FLAT` or `NOTE_NAMES_SHARP`. If `dispose` is -1, the method pulls a name from `NOTE_NAMES_FLAT`, otherwise from `NOTE_NAMES_SHARP`. The index into the array is `noteValue % Note.OCTAVE`. No octave is returned.

- `Note.getDurationString(double decimalDuration)` is a static method that converts values like 0.5 to `h`, 0.25 to `q`, 0.09375 to `s.` (dotted sixteenth), 2.5 to `wwh`, and so on. Or, if it can't figure out the math (say it's given 0.234872), it will return a slash followed by the decimal value (e.g., `/0.234872`).

- `Note.getDurationStringForBeat(int beat)` is a static method that converts 2 to `h`, 4 to `q`, 8 to `i`, 16 to `s`, and any other value to `/` followed by 1.0 divided by `beat`.

- `toDebugString()` is discussed ahead.

Phew, I'm pretty sure that's all of them.

Since we're on the topic of interesting results you can get from static Note methods, there are two other interesting methods that you may be interested in:

- `Note.getFrequencyForNote(int noteValue)` and `Note.getFrequenctForNote(String note)` are static methods that return the frequency, in Hertz, for the given note. For example, A4 returns 440.0.

These frequency-related methods are used most frequently (ha!) with microtones. For example, `MicrotoneManualTest` uses `getFrequencyForNote()` several times.

## Modifying Default Values for Notes

All of that talk about the default duration being 0.25, or the default octave being 5 for the treble clef and 4 for the bass clef, or default on/off velocity being 64… well, you can change those things.

`DefaultNoteSettingsManager` is a static class (you use `getInstance()` to access it) with which you can override the default values for octave, duration, and on/off velocity for your notes.

Keep in mind that these settings happen outside the realm of Staccato, so if you change the settings and share a Staccato string with someone else, they will hear different music.

## Diagnosing Notes

As you can tell, notes get complicated pretty quickly. You can use the `toDebugString()` method to get a string that lists all of the properties of a note. Going through this string could help you identify why one note might not be the same as another note when you think they should be the same note.

# 4.3

# *Chords and Chord Progressions*

Chords are harmonies of notes, and chord progressions are melodies of chords. JFugue provides classes that let you work with both chords and chord progressions easily, and in a way that is consistent with what you may already know from music theory.

## Creating Chords

To create a chord, call the `Chord` constructor with a string representing the chord, which is a root note (e.g., `C`, `Bb`, `A4`, `60`) followed by a chord string (e.g., `MAJ`, `MIN`, with more discussed below)

```
Chord chord = new Chord("Cmaj");
```

If you do not specify an octave for the root note, JFugue automatically selects octave 4 (or more correctly, the value returned by `DefaultNoteSettingsManager.geInstance().getDefaultBassOctave()`), which is an octave lower than the default for treble clef notes. In other words, the code sample above produces a chord in which C4 is the root note, rather than C5 (Middle-C). If you specify an octave (e.g., `C5maj`) or use a note number (e.g., `60maj`), the note will be in the octave you expect.

You can also create a chord by providing the notes that comprise the chord. You can do this using the static `fromNotes` method, using one of the following method signatures:

```
public static Chord fromNotes(String noteString)
public static Chord fromNotes(String[] noteStrings)
public static Chord fromNotes(Note[] notes)
```

These are useful functions that do a bit of processing behind the scenes that allow you to enter notes in any order and get the expected result. For example, whether you provide `C E G` or `G E C`, you'll get a `Cmaj`. The functions also know how to translate the lowest note (not the first note) into a chord inversion, so `E4 G4 B4 C5` will return a `C5maj7^`.

The functions pay attention to octave and use octave in the returned chord if the provided notes are given an octave in the first place, and if the octaves in all of the notes can be used to create a clearly defined chord octave. For example, we just saw that `E4 G4 B4 C5` will return a `C5maj7^`. But `C`, `E`, `G`, and `B` anywhere on the keyboard logically represent a C-Major 7th chord. For example, `E3 F6 B2 C8` is also a `Cmaj7`, but we cannot say that it is a `Cmaj7` with a particular octave for the root, C (clearly, these notes do not represent `C8maj7`). Similarly, `E3 F6 E4 F8 B2 C7 C5` also comprise a `Cmaj7`. In these cases, the chord is returned without an octave.

## The ChordMap

JFugue defines chords using the `Intervals` class, and the `Chord` class contains a `chordMap` that defines a number of common types of chords. `chordMap` contains a bunch of entries like this:

```
chordMap.put("MAJ", new Intervals("1 3 5"));
```

The table below shows all of the chords that JFugue includes in its chord map. If there are chords that you want to use that you don't see defined here, it is easy to add them to the chord map! From anywhere in your application, simply say something like:

```
// Add a Power Chord to JFugue!
Chord.addChord("POW", "1 5");
```

There are a couple of guidelines for creating a chord name:

1. Don't start the chord name with a number, or it will be indistinguishable from the root's octave,
2. Don't use the slash, angle brackets, square brackets, parentheses, plus, or underscore in your chord name (instead, use percent, `%`),

3. Make sure to test your new chord name to make sure the parser can parse it correctly.

| Common Name | JFugue Name | Intervals |
|---|---|---|
| **Major Chords** | | |
| Major | MAJ | 1 3 5 |
| Major 6th | MAJ6 | 1 3 5 6 |
| Major 7th | MAJ7 | 1 3 5 7 |
| Major 9th | MAJ9 | 1 3 5 7 9 |
| Added 9th | ADD9 | 1 3 5 9 |
| 6/9 | MAJ6%9 | 1 3 5 6 9 |
| 7/6 | MAJ7%6 | 1 3 5 6 7 |
| Major 13th | MAJ13 | 1 3 5 7 9 13 |
| **Minor Chords** | | |
| Minor | MIN | 1 *b*3 5 |
| Minor 6th | MIN6 | 1 *b*3 5 6 |
| Minor 7th | MIN7 | 1 *b*3 5 *b*7 |
| Minor 9th | MIN9 | 1 *b*3 5 *b*7 9 |
| Minor 11th | MIN11 | 1 *b*3 5 *b*7 9 11 |
| 7/11 | MIN7%11 | 1 *b*3 5 *b*7 11 |
| Minor Added 9th | MINADD9 | 1 *b*3 5 9 |
| Minor 6/9 | MIN6%9 | 1 *b*3 5 6 |
| Minor Major 7th | MINMAJ7 | 1 *b*3 5 7 |
| Minor Major 9th | MINMAJ9 | 1 *b*3 5 7 9 |
| **Dominant Chords** | | |
| Dominant 7th | DOM7 | 1 3 5 *b*7 |
| Dominant 7/6 | DOM7%6 | 1 3 5 6 *b*7 |
| Dominant 7/11 | DOM7%11 | 1 3 5 *b*7 11 |
| Dom 7th Sus | DOM7SUS | 1 4 5 *b*7 |
| Dom 7/6 Sus | DOM7%6SUS | 1 4 5 6 *b*7 |
| Dominant 9th | DOM9 | 1 3 5 *b*7 9 |
| Dominant 11th | DOM11 | 1 3 5 *b*7 9 11 |
| Dominant 13th | DOM13 | 1 3 5 *b*7 9 13 |
| Dom 13th Sus | DOM13SUS | 1 3 5 *b*7 11 13 |
| Dom 7th 6/11 | DOM7%6%11 | 1 3 5 *b*7 9 11 13 |
| **Augmented Chords** | | |
| Augmented | AUG | 1 3 #5 |
| Augmented 7th | AUG7 | 1 3 #5 *b*7 |
| **Diminished Chords** | | |
| Diminished | DIM | 1 *b*3 *b*5 |
| Diminished 7th | DIM7 | 1 *b*3 *b*5 6 |
| **Suspended Chords** | | |
| Suspended 2nd | SUS2 | 1 2 5 |
| Suspended 4th | SUS4 | 1 4 5 |

**Table 10. Chords provided by JFugue**

You can also get a list of all of the available chords by using `Chord.getChordNames()`, you can find the intervals for a specific chord using `Chord.getIntervals(String chordName)`, and you can remove a chord using `Chord.remove(String chordName)`. These methods simply delegate to the `chordMap` object, which is a `Map<String, Intervals>`. There is also a method that returns the name of a chord if you pass the set of intervals (of course, this depends on `chordMap` having an entry in which the intervals exist). This is `Chord.getChordType(Intervals intervals)`.

## Creating a Chord with a Root and Intervals, or a Key

The Chord class has two other constructors. One takes a root note and a set of intervals; the other takes a key. And, since a key in JFugue can be specified using a root note and a scale (which is, at its most basic, a set of intervals), the behavior behind the scenes is identical for these two constructors.

To create a C-Major chord with the root and intervals constructor, you would say:

```
Chord chord = new Chord(new Note("C3"), new Intervals("1 3 5"));
```

(You could even pass the Intervals instance from the ChordMap, but that would only be a more verbose way of doing what you can already do with the basic constructor introduced earlier.)

Creating a chord with a key is an interesting idea, and I am not sure whether it is entirely useful in a musical context, but it is easy to do. The resulting chord will consist of all of the notes in the key. And, you can also define a key with a chord, in which case the key will consist of all of the notes in the chord.

## Methods on a Chord

Once you have a chord, there are a lot of things you can do with it! Of course, you can get the notes that comprise a chord:

```
Note[] notes = chord.getNotes();
```

And you can get the intervals:

```
Intervals intervals = chord.getIntervals();
```

You can change the bass note of a chord, or you can set the inversion numerically:

```
chord.setBassNote("E"); // Pass a string or a Note object
chord.setInversion(2);  // Now we have a second inversion
```

And, of course, you can get the bass note, root note, or inversion:

```
Note note = chord.getBassNote();
Note note = chord.getRootNote();
int inversion = chord.getInversion();
```

If you have a chord in which the root does not have an octave (or if you wish to change the octave of the root), you can use `chord.setOctave(int octave)`.

Finally, you can do some cool stuff that deserves a section of its own: doing an operation on all notes that comprise a chord. `ChordProgression` has something similar, so we'll cover this after we introduce chord progressions in the next section. If you're curious, skip ahead to the section titled, "Each Chord As, All Chords As, and Distribute."

## Getting Human-Readable Chord Names

As much as Staccato puts a premium on human-readable music expression, chords like `DOM7%6%11` are not as clear as saying "Dominant 7th 6/11" – but, because a chord like this would contain other characters with specific meaning in Staccato (in this case, spaces and the slash character), the chord names need to refrain from using those characters. However, the `Chord` class does have a static map between Staccato chord name and human-readable name, and you can get a chord's human-readable string by calling `getHumanReadableString()` on a `Chord` instance. This will return the human-readable name if one exists, otherwise it will return the original chord string. The returned value includes the root note. For example, if you create a chord using JFugue's `MAJ6%9` and then call `getHumanReadableString()` on it, you will see `6/9` returned, which you can print it for the benefit of your users. If you want to add your own human readable strings, use the static `Chord.putHumanReadableName(String chordName, String humanReadableChordName)` method.

## Creating Chord Progressions

To create a chord progression, call the `ChordProgression` constructor with a string representing the chord progression. For example, to specify a I-IV-V progression, use the following:

```
ChordProgression cp = new ChordProgression("I-IV-V");
```

You may use either dashes or spaces to separate the chords.

You can also create a `ChordProgression` by passing in existing chords using the two `ChordProgression.fromChords()` methods, one of which

takes a string that consists of chords, the other of which takes an array of `Chord` objects.

Once you have an instance of a `ChordProgression` object, you can do several things with it. You can play it by sending the chord progression to `Player.play()`. You can get a pattern from it using `getPattern()`. You can get the individual chords from the progression using `getChords()`.

For each of these things, you may first want to set the key in which the chord progression will be played. To do this, call the `setKey(key)` method:

```
cp.setKey("BbMAJ");
```

There are two `setKey` methods: one that takes a Staccato string, which is parsed behind-the-scenes to create a `Key` object, and the other is to pass a `Key` object directly... and as you'll learn in the next section, when you create a `Key` object you pass a Staccato string to the constructor anyway, so these are functionally equivalent.

Now you can get the chords in a chord progression. What chords do you get from a I-IV-V progression in B-flat major? Let's find out!

```
Chord[] chords = cp.getChords(); // Bb3MAJ, Eb4MAJ, F4MAJ
```

When you specify a chord progression, the case that you use is important. A "`ii`" chord is different from a "`II`" chord, for example.

You can also specify diminished chords by adding a "`o`" or "`d`" to the end of the chord (for example, "`viio`"), and you can add a "`7`" to the end of a chord to get a seventh.

## Each Chord As, All Chords As, and Distribute

Certain styles of music use chord progressions, but the song itself does not necessarily use those chords in the given order. For example, a blues song may have a chord progression of `I IV V`, but the way that is actually played for each measure of the song is `I I I I IV IV I I V IV I I`.

| | | | |
|---|---|---|---|
| **I** | **I** | **I** | **I** |
| **IV** | **IV** | **I** | **I** |
| **V** | **IV** | **I** | **I** |

**Figure 13. Blues progression**

If you have a `ChordProgression`, you can create a blues rhythm with one additional line of code. Let's introduce the `ChordProgression` method named `eachChordAs`.

This method takes a string that follows a special format. Within this string, you refer to each chord in the progression using an index. So, if you have a `I IV V` progression, in this string you would refer to `I` as `$0`, `IV` as `$1`, and `V` as `$2`. This allows you to change the underlying progression without changing the string passed to `eachChordAs`. You can also refer to each chord multiple times in the `eachChordAs` string. That blues progression from Figure 13 would be written like this:

```
ChordProgression cp = new ChordProgression("I IV V")
.eachChordAs("$0 $0 $0 $0 $1 $1 $0 $0 $2 $1 $0 $0");

Pattern pattern = cp.getPattern();
```

A blues song in two lines of code! How cool is that?!

There's a sibling method, `allChordAs()`. Just as `eachChordAs()` works on elements of a chord progression, `allChordAs()` works on elements of a chord – in other words, the actual notes. Let's say, for example, that you want to play each chord in the blues progression as an arpeggio: play the first note of the chord, then the second, then the third. In this case, the `$0`, `$1`, and `$2` indices refer to notes within a chord.

In both `eachChordAs()` and `allChordsAs()`, the `$x` strings are simply replaced with the element they point to. You can only use single-digit numbers here. Anything after the index will be appended to the element that replaces the index. That means, for `allChordsAs()`, we can say `$0i` which will mean, "Play the 0th note as an eighth note."

Here's what this would look like in code:

```
ChordProgression cp = new ChordProgression("I IV V")
.eachChordAs("$0 $0 $0 $0 $1 $1 $0 $0 $2 $1 $0 $0")
.allChordAs("$0i $1i $2i $3i $4i $3i $2i $1i");

Pattern pattern = cp.getPattern();
```

An arpeggiated blues song in two lines of code! More coolness!

Finally, there is one other method that works across all elements of a `ChordProgression` (but not a `Chord`): `distribute()`, which takes a string and applies that string to each Chord in the progression. You would use this when you want to keep your progression looking clean, but you need to add something to each element of the progression. A great example is that arpeggiated blues song. If we really wanted this to sound good, we'd play the notes up and down (something like C, E, G, A, A, G, E, C) but

what's missing from each chord is a 6th note in the 5th position. In other words, we would want a `Cmaj7%6` here. But how do we say that in the I IV V progression? `I7%6 IV7%6 V7%6` would be really messy. So instead, we can say:

```
ChordProgression cp = new ChordProgression("I IV V")
.distribute("7%6");
```

Much cleaner!

In fact, here's a cool blues song, in technically one line of code that I've split on different lines to make it easier to read. (In addition to showing chords and progressions, this demonstrates the power of JFugue's fluent API, where setters return the object itself so the object can continue to be worked on):

```
new Player().play(new ChordProgression("I IV V")
     .distribute("7%6")
     .allChordsAs("$0 $0 $0 $0 $1 $1 $0 $0 $2 $1 $0 $0")
     .eachChordAs("$0i $1i $2i $3i $4i $3i $2i $1i")
     .getPattern()
     .setInstrument("Acoustic Bass")
     .setTempo(120));
```

And that's how we do chords and progressions and patterns and API calls in JFugue.

(By the way, if you're interested in that dollar sign replacement functionality, you'll find the implementation in `ReplacementFormatUtil`. The single static method, `replaceDollarsWithCandidates`, takes a string with the `$0` .. `$9` or `$_` entries, an array of `PatternProducer`s to use as replacements for the numeric indices, and a single `PatternProducer` for the underscore replacement. You can use it for your own creations if you'd like. The `Intervals` class also uses this capability. You'll learn about that next.)

# 4.4

# *Intervals, Scales, and Keys*

Scales, keys, and intervals are all closely related in both music theory and in JFugue. A scale indicates the pitches within an octave and can be specified using a set of intervals. A key consists of a scale and a root note.

## Intervals

Intervals are fundamental to music theory within JFugue – so much so that the `Intervals` class was the first class I wrote for the JFugue 5.0 reboot. This class lets you encode a sequence of arbitrary intervals that can be used for chords, scales, keys, or even melodies. The intervals are indicated using a string that consists of numeric whole number degrees, which may be modified using the flat (`b`) or sharp (`#`) characters. For example, a C-major chord, C-E-G, is indicated by a 1-3-5 interval, or `Intervals("1 3 5")`. A C-minor chord, C-Eb-G, is indicated by `"1 b3 5"`. Notice that when defining intervals, the flat or sharp character comes *before* the interval number; this is different than a Staccato string in which the flat or sharp comes after a note (`F#` or `Eb`, for example) but is consistent with interval notation traditionally used in music theory.

You can also create a set of intervals from a set of notes using the static `Intervals.createIntervalsFromNotes()` methods. There are three flavors that let you use a `Pattern`, a `String` of notes, or a `Note[]`. For example, `Intervals.createIntervalsFromNotes("C E G")` gives a 1-3-5 interval.

The `Intervals` class contains two maps which you may find useful in your programming. One is a map between whole number degrees to halfsteps; the other is a map between halfsteps to whole number degrees. When you create an interval like `"1 3 5"`, you are using whole number degrees to indicate the distance from the root to the other notes. Internally, those whole number degrees are converted to halfsteps (0, 4, and 7) to create the actual notes. If you would like to access these maps, which are defined as statics in the `Intervals` class, they are `Intervals.wholeNumberDegreeToHalfsteps` and `Intervals.halfstepsToWholeNumberDegree`. You can also get the halfsteps for the values in an Intervals class using `getHalfstepArray()`, which returns an `int[]` consisting of the halfsteps for each whole number degree in the interval. There is also a static convenience method, `Intervals.getHalfsteps(String wholeNumberDuration)`, which will convert a string like `"##5"` or `"b3"` to the corresponding number of halfsteps. This uses another static convenience method, `Intervals.calculateHalfstepDeltaFromFlatsAndSharps(String wholeNumberDuration)`, which returns an `int` counting the number of sharps or flats in the string, returning a negative number for flats or a positive number for sharps. `"##5"` will return +2, `"b3"` will return -1.

Like many classes in JFugue, `Intervals` is a `PatternProducer`, so you can call `getPattern()`, but first you must set the root of the interval using `setRoot(Note root)` or `setRoot(String rootString)`. `Intervals` is also unique in that it is a `NoteProducer`, which means you can get a list of notes from the class using `getNotes()` – again, you must set a root first (behind the scenes, `getNotes()` calls `getPattern()`).

You can find the size of a set of intervals – that is, the number of whole number degrees in the interval – using `getSize()`, and you can get the interval at a given index by using `getNthInterval(int n)`. For example, `getNthInterval(1)` on `"1 3 5"` will return 3 (the index is zero-based). Intervals can be rotated using `rotate(int n)`, where `n` is the number of front-to-back rotations to perform on the interval string. `"1 3 5 7 9 13"` (a Major 13th) rotated 3 times will result in an interval consisting of `"7 9 13 1 3 5"`.

Remember `ChordProgression`'s `eachChordAs()` and `allChordsAs()` methods? The `Intervals` class has something similar, which is simply called `as()`. Like the similar methods in `ChordProgression`, `as()` takes a string that contains dollar signs with numbers that indicate an index

into the interval, or `$_` which indicates all notes in the interval. The return value from `as()` is the `Intervals` instance itself because of the fluent API, but when you call `getPattern()` after calling `as()`, you will get a completed version of the string you sent in.

Here are two examples:

```
new Intervals("1 3 5").setRoot("C").as("$_i $0q $1h
$2w").getPattern();
// Returns "C5i E5i G5i C5q E5h G5w"

new Intervals("1 3 5").setRoot("C").as("$0q. $1q $2h");
// Returns "C5q. E5q G5h"
```

## Scale

A Scale in JFugue is a simple class that consists of an instance of Intervals, an optional name, and a value that indicates whether the scale is a major or minor scale (although this may be overly simplistic, given the range of possible scales).

Whether a scale is a major or a minor can get set using `setMajorOrMinorIndicator(byte indicator)` which takes a values of `Scale.MAJOR_INDICATOR` or `Scale.MINOR_INDICATOR`. There is a corresponding getter, `getMajorOrMinorIndicator()`, which returns the same values.

The Scale class also defines a few scales:

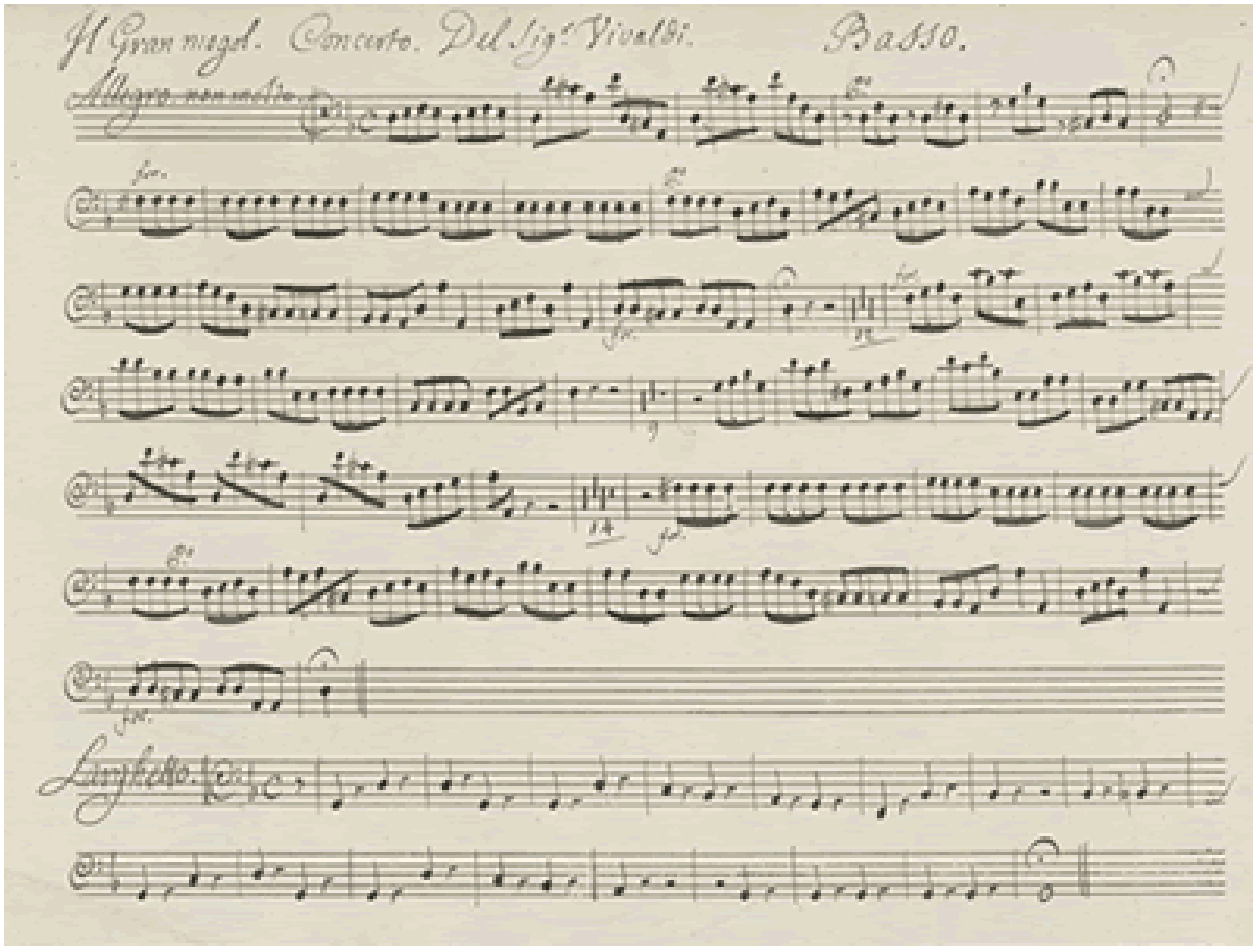
- The Major scale, `Scale.MAJOR`, consisting of `Intervals("1 2 3 4 5 6 7")`
- The Major scale, `Scale.MINOR`, consisting of `Intervals("1 2 b3 4 5 b6 b7")`
- The Circle of Fifths, `Scale.CIRCLE_OF_FIFTHS`, consisting of `Intervals("1 2 3b 4 5 6 7b")`

## Key

Like the Scale class, the Key class in JFugue is also simple. It lets you specify a Key using either a roote note and a `Scale` (which we just learned is basically a wrapper for `Intervals`), a `Chord` (which is interesting! But a chord is a root note and a set of `Intervals`), or a `String` key signature using Staccato notation – for example, `"F#MAJ"` or `"Kb"`, where the second method lets you just plop in the number of flats or sharps you see in sheet music and JFugue will arrive at the correct key. This is discussed in more detail in Section 2.6.

# Part 5

## Players and Parsers




The score of "Il Gran Mogol," a recently discovered flute concerto by Antonio Vivaldi.
Identified in the National Archives of Scotland by Andrew Wooley.
For more information, see http://www.nas.gov.uk/about/101007.asp

# 5.1

## *Players*

So far in this book, you have seen many examples of playing music using the `Player` class. In this chapter, you will learn more about how the Player class works, and the other types of players that you can use to play your music in different ways. JFugue provides two players: a managed player, which is the basic one that you have seen so far and that can also be used to stop, pause, and otherwise manage playback (great for playing back a known song), and a real-time player that can accept musical instructions added dynamically (great for user interfaces and interactive instruments).

### The Player

You've already learned the most-used method of the Player class: `player.play()`. This is clearly a central part of JFugue! The `play()` method can take a `String` or any `PatternProducer` (which is an interface that consists of a single method: `getPattern()`). You can also pass an array of `String`s or `PatternProducer`s (but not a combination of both) to the `play()` method.

`Player` provides the default capability that makes it so easy to create music in JFugue: it uses the `StaccatoParser` to turn strings into musical

events, and it sends these events to a `MidiParserListener`. The result is that you hear your text like `"C D E F G A B"` turn into music. Of course, JFugue lets you connext any Parser to any ParserListener, as you will learn in this section of the book. Player simply provides the most common combination of Parser and ParserListener that results in immediate and enjoyable results. That code essentially looks like this:

```
StaccatoParser staccatoParser = new StaccatoParser();
MidiParserListener midiParserListener = new MidiParserListener();
staccatoParser.addParserListener(midiParserListener);
```

When you call `play()` with a Staccato string, Player calls the `parse()` method on `staccatoParser`, then gets the resulting MIDI Sequence from `midiParserListener`:

```
    staccatoParser.parse(pattern);
    return midiParserListener.getSequence();
```

The sequence is passed to a MIDI Synthesizer and the music is played. In fact, it is useful to understand that Player first calls these methods to parse all of the Staccato and turn it into MIDI, and then it plays the MIDI sequence. Parsing Staccato into MIDI happens quickly; playing the sequence takes the actual duration of the song.

There are two other capabilities that Player provides. The first is that you can simply get a MIDI Sequence using `getSequence()`, which will take the same parameters as `play()`. Second, you can use the `delayPlay()` method to play the given music in the future, specified by a delay in milliseconds. This will create a new thread that first sleeps, then plays your music, again specified as either s `String`, a `PatternProducer`, an array of `String`s, or an array of `PatternProducer`s. This is useful when you are playing with the closest thing JFugue has to time travel. You'll read about the Temporal ParserListener-Player in this section.

Behind the scenes, `Player` delegates all of its playing activity to a class called `ManagedPlayer`. What's a `ManagedPlayer`, you ask? Good question.

## The Managed Player

`ManagedPlayer` actually handles the heavy lifting for `Player`, but it is something that you do not need to know about unless you want to provide the users of your code or applicaction with the ability to stop, pause, resume, and seek musical playback. `ManagedPlayer` provides all of those capabilities, and it maintains state so it knows whether the playback is started, finished, or paused. To get the `ManagedPlayer`, use `player.getManagedPlayer()`.

You can also create a `ManagedPlayerListener` to listen to your `ManagedPlayer`. This will let you know when the `ManagedPlayer` has been started, finished, paused, resumed, or when a specific tick value is accessed via seek.

`ManagedPlayer` lets `Player` have a clean API: the `Player` class has a couple of `play()` methods, and that's about it. It also consolidates starting/pausing/resuming/etc. in one place in case other players—either ones that are introduced in future versions of JFugue or ones that you create—can do whatever is necessary in the player itself but still hand off the managed capability to `ManagedPlayer`.

The state that the `ManagedPlayer` is in can be determined by callbacks to a `ManagedPlayerListener` (e.g., `onStarted()`, `onPaused()`) or by querying the ManagedPlayer using `isStarted()`, `isPaused()`, or `isFinished()`. A sequence is not in any of these three states before `start()` is called. When `start()` is called, `isStarted()` returns true. Whenever `pause()` is called, `isPaused()` returns true, and calling `resume()` leads to `isPaused()` returning false. When the sequence finishes, `isFinished()` will be true. `isStarted()` will only return false when the `ManagedPlayer` is first created and `start()` has not been called, or when `reset()` has been called.

The table below summarizes the methods in `ManagedPlayer` that let you manipulate the playback of a sequence:

| ManagedPlayer method | started | paused | finished | Fires this event in ManagedPlayerListener |
|---|---|---|---|---|
| `start(Sequence seq)` | True | False | False | `onStarted(Sequence seq)` |
| `pause()` | - | True | - | `onPaused()` |
| `resume()` | - | False | - | `onResumed()` |
| `seek(long tick)` | - | - | - | `onSeek(long tick)` |
| `finish()` | - | - | True | `onFinished()` |
| `reset()` | False | False | False | `onReset()` |

ManagedPlayer can also tell you the length of a sequence in ticks sing `getTickLength()` and the current tick position usig `getTickPosition()`. Both of these methods delegate to the Java Sound `Sequence` instance that is playing the music.

There is one final thing to mention about `ManagedPlayer`: It uses a sequener provided by `SequencerManager` and a synthesizer provided by `SynthesizerManager`. Read ahead for more on these classes.

## The Realtime Player

The player that you've learned about up to this point is great for when you already know what music you want to play. In other words, JFugue generates a MIDI Sequence from your Staccato string and sends it to Java Sound's Sequencer. But if you want to create an application in which your users will create music on the fly—maybe as part of an interactive instrument—or if you have live data or an algorithm that is building music in realtime, you need to play your music differently. In this case, you will not have a MIDI Sequence. Instead, you will want to send MIDI events directly to a Synthesizer. For this purpose, JFugue has a `RealtimePlayer`.

Through the `RealtimePlayer`, you can play `Patterns` just as you can with the regular `Player`, but you can play those patterns at a time of your choosing. You can also call specific methods that change the instrument or current track, and start and stop specific notes. Here is a description of the RealtimePlayer's methods.

- **`changeTrack`, `changeInstrument`, `changeChannelPressure`, `changePolyphonicPressure`, `changeController`, `setPitchBend`:** These all do exactly what you think they should do! In fact, they pretty much delegate to the MidiSynthesizer that is under the hood of the RealtimePlayer.

- **`startNote`, `stopNote`:** These are realtime ways of starting and stopping a note. They are similar to the `noteOn()` and `noteOff()` methods of MidiSynthesizer. Of course, the JFugue difference is that you can call either of these methods with a Note instead of a MIDI note value.

- **`play`**: The RealtimePlayer can play a Pattern or a Staccato string!

- **`startInterpolator`, `stopInterpolator`**: The RealtimePlayer lets you establish "interpolators," which can send commands to the RealtimePlayer while other events are being played. See the section on RealtimeInterpolator below.

- **`getCurrentTime`, `scheduleEvent`, `unscheduleEvent`**: The RealtimePlayer lets you schedule a musical event that will happen in the future. See the section on scheduled events below.

- **`close`**: You should call this method when you are done using the RealtimePlayer.

The following example provides a basic demonstration for some of the methods listed above.

```
import java.util.Scanner;

import javax.sound.midi.MidiUnavailableException;

import org.jfugue.pattern.Pattern;
import org.jfugue.realtime.RealtimePlayer;
import org.jfugue.theory.Note;

public class RealtimeExample {
    public static void main(String[] args) throws
MidiUnavailableException {
        RealtimePlayer player = new RealtimePlayer();
        Scanner scanner = new Scanner(System.in);
        boolean quit = false;
        while (quit == false) {
            System.out.print("Enter a '+C' to start a note, '-C'
to stop a note, 'i' for a random instrument, 'p' for a pattern,
or 'q' to quit: ");
            String entry = scanner.next();
            if (entry.startsWith("+")) {
                player.startNote(new Note(entry.substring(1)));
            }
            else if (entry.startsWith("-")) {
                player.stopNote(new Note(entry.substring(1)));
            }
            else if (entry.equalsIgnoreCase("i")) {
                player.changeInstrument((int)(Math.random() *
128));
            }
            else if (entry.equalsIgnoreCase("p")) {
                player.play(PATTERNS[(int)(Math.random() *
PATTERNS.length)]);
            }
            else if (entry.equalsIgnoreCase("q")) {
                quit = true;
            }
        }
        scanner.close();
        player.close();
    }

    private static Pattern[] PATTERNS = new Pattern[] {
            new Pattern("Cmajq Dmajq Emajq"),
            new Pattern("V0 Ei Gi Di Ci  V1 Gi Ci Fi Ei"),
            new Pattern("V0 Cmajq V1 Gmajq")
    };
}
```

### Scheduling Events

You can schedule events to execute at a future point in time. To find the current time, use the `getCurrentTime()` method. The `RealtimePlayer`

keeps time in milliseconds. To schedule an event, call `scheduleEvent()` and provide a desired time and a class that implements the `ScheduledEvent` interface. This interface has one method, `execute()`, that provides a reference to the `RealtimePlayer`, and also provides the time at which the event is being executed. You can use this to send events directly to the `RealtimePlayer`.

For example, if you want to schedule an event for 5 seconds from now that plays a specific pattern, you would use:

```
ScheduleEvent event = new ScheduledEvent() {
    @Override
    public void execute(RealtimePlayer player, long time) {
        player.play("C D E");
    }
};
RealtimePlayer player = new RealtimePlayer();
player.schedule(player.getCurrentTime()+5000, event);
```

You may also call the `unschedule()` method to cancel events that you had already scheduled.

### Realtime Interpolator

An interpolator is used to adjust values on a setting based on the elapsed time between a start and an end time. Interpolators are often used in animation. For example, given a starting point at time 0 and an ending point at time 100, an interpolator can determine the points where some figure should be drawn between times 0 and 100. Interpolators can use any function to determine that point; some interpolators are simply linear, whereas others might have "ease in" or "ease out" functions that make the animating figure appear to slow down at the end points.

In JFugue, you can use a `RealtimeInterpolator` with the `RealtimePlayer` to provide updates to any musical function over time, while you then send other musical notes to the player. The most common use cases might be adjusting volume or pitch bend while notes are playing, but there is no limit to the types of adjustments you can make using the interpolator.

The example below shows how to use an interpolator. First, create a class that implements the `RealtimeInterpolator` interface (or, like the example below, create an anonymous inner class). The interface has one method, `update`, that provides a reference to the `RealtimePlayer`, the amount of time that has elapsed since the interpolator started, and the percent complete given as a double value from 0.0 to 1.0. Within the `update` method, you can make changes to the music being played through the `RealtimePlayer`. To start the interpolator, call the

`startInterpolator()` method, providing a reference to the interpolator and the duration, in milliseconds, for which the interpolator should be active. In the example below, the interpolator changes the pitch bend on the synthesizer while a note is being played.

```
RealtimePlayer player = new RealtimePlayer();
RealtimeInterpolator ri = new RealtimeInterpolator() {
    int pitchBend;
    @Override
    public void update(RealtimePlayer realtimePlayer, long
elapsedTime, double percentComplete) {
        realtimePlayer.setPitchBend(pitchBend++);
    }
};
player.startInterpolator(ri, 3000);
Note note = new Note("C");
player.changeInstrument("Flute");
player.startNote(note);
try {
    Thread.sleep(3000);
} catch (InterruptedException e) { /* Handle exception */ }
player.stopNote(note);
player.close();
```

When you are done using an instance of `RealtimePlayer`, you should call the `close()` to wrap things up.

## Starting a Player with a Specific Sequencer or Synthesizer

`Sequencer` and `Synthesizer` are two classes provided by the Java Sound package. The sequencer sends a MIDI Sequence, which is generated when you use the `Player` class to play Staccato music, to a synthesizer; the synthesize creates the audio output. When you use the various Players, JFugue uses the default `Sequencer` and `Synthesizer` instances provided by the Java Sound `MidiSystem` class.

There may be times when you want a Player to use a sequencer or synthesizer that you have already set up. For example, if you are loading new soundbanks into a synthesizer, you will want to make sure that the Player uses that specific instance of the synthesizer.

JFugue provides two classes, `SequencerManager` and `Synthesizer Manager`, that let you provide your instances of with a sequencer or synthesizer. Both work similarly in that they are static (you call the static method `getInstance()` to access the instance), both have a method that allows you to get the default value (`getDefaultSequencer()` or `getDefaultSynthesizer()`), and both have getters and setters that let you set your own values (`getSequencer()/getSynthesizer()` and `setSequencer(Sequencer seq)/setSynthesizer(Synthesizer synth)`). If

you do not set a specific sequencer or synthesizer, the getters return the default values.

`SequencerManager` has several additional methods. You can use `connectSequencerToSynthesizer()` to connect the sequencer to the synthesizer returned by `SynthesizerManager`. You can also listen for end of track messages using `add/removeEndOfTrackListener()`. The `EndOfTrackListener` interface in `org.jfugue.player` consists of a single callback, `onEndOfTrack()`, that is sent when the sequence being played sends MIDI Meta Event 47. `ManagedPlayer` uses this to know when to call `finish()`.

# 5.2

## *Parsers and ParserListeners*

One of the capabilities that makes JFugue uniquely extensible is an architecture that provides parsers for interpreting musical data, and listeners that can respond to the musical events that the parser encounters. This architecture lets JFugue parse Staccato music and transform it into MIDI. It also allows JFugue to read a MIDI file and transform it into sheet music that can be used in LilyPond. In fact, it allows for any music format to be transformed into any other music format, whether that format is something well-known (such as MIDI or MusicXML), an emerging standard (such as Staccato), or your own experimental creation (for example, how about music that can be interpreted from, or written into, biological DNA!).

### A Common Pattern for Parsers and ParserListeners

If you use parsers and parser listeners, you will soon become familiar with the following pattern, which also makes an appearance in several places in the JFugue API. The pattern consists of these five steps:

1. Create an instance of a `Parser`
2. Create an instance of a `ParserListener`
3. Add the `ParserListener` to the `Parser` using `addParserListener()`

4. Call a `parse()` method on the Parser, passing in whatever value the Parser knows how to parse
5. Get a result from the ParserListener, in whatever format (and using whatever methods) the ParserListener provides

For example, the constructor in the `Player` class has the beginning of this pattern:

```
staccatoParser = new StaccatoParser();
midiParserListener = new MidiParserListener();
staccatoParser.addParserListener(midiParserListener);
```

and the `play()` method eventually finishes it:

```
staccatoParser.parse(pattern);
return midiParserListener.getSequence();
```

Similarly, the `loadPatternFromMidi()` method in the `MidiFileManager` class has the following code that parses MIDI and turns it into Staccato:

```
MidiParser midiParser = new MidiParser();
StaccatoParserListener staccatoListener = new
   StaccatoParserListener();
midiParser.addParserListener(staccatoListener);
midiParser.parse(MidiSystem.getSequence(in));
return staccatoListener.getPattern();
```

You will see (or write!) a similar pattern if you decide to use classes like `MusicXmlParser`, `MusicXmlParserListener`, `LilyPondParserListener`, and so on.

## Using Parsers and ParserListeners

Parsers are interesting from an object-oriented perspective. Here's why: While there is a `Parser` class, it does not define a `parse()` method! Every `Parser` presumably has its own way of dealing with whatever data it needs to parse. Since JFugue does not know how each Parser will work, or what kind of data each Parser needs to parse, it does not try to enforce an arbitrary structure by defining a `parse()` method for subclasses of `Parser` to deal with.

However, the `Parser` class does come with a bunch of good stuff, particularly the maintenance of the list of `ParserListeners` and a collection of methods that fire events to the `ParserListeners`. There are a fair number of events, and it is these events that allow `Parsers` and `ParserListeners` to speak the same language and share musical information.

In contrast to `Parser`, `ParserListener` is an interface that specifies many methods that an implementing class must override. Each of these is called in response to some musical information that the Parser finds.

Table 11 lists the musical events.

| Parser method to fire event | ParserListener method to override | Parameter |
|---|---|---|
| `fireBeforeParsingStarts` | `beforeParsingStarts` | *No parameter* |
| `fireAfterParsingFinished` | `afterParsingFinished` | *No parameter* |
| `fireTrackChanged` | `onTrackChanged` | byte track |
| `fireLayerChanged` | `onLayerChanged` | byte layer |
| `fireInstrumentParsed` | `onInstrumentParsed` | byte instrument |
| `fireTempoChanged` | `onTempoChanged` | int tempoBPM |
| `fireKeySignatureParsed` | `onKeySignatureParsed` | byte key, byte scale |
| `fireTimeSignatureParsed` | `onTimeSignatureParsed` | byte numerator, byte denominator |
| `fireBarLineParsed` | `onBarLineParsed` | long id |
| `fireTrackBeatTimeBookmarked` | `onTrackBeatTimeBookmarked` | String timeBookmarkId |
| `fireTrackBeatTimeBookmark Requested` | `onTrackBeatTimeBookmark Requested` | String timeBookmarkId |
| `fireTrackBeatTimeRequested` | `onTrackBeatTimeRequested` | double time |
| `firePitchWheelParsed` | `onPitchWheelParsed` | byte lsb, byte msb |
| `fireChannelPressureParsed` | `onChannelPressureParsed` | byte pressure |
| `firePolyphonePressureParsed` | `onPolyphonicPressureParsed` | byte key, byte pressure |
| `fireSystemExclusiveParsed` | `onSystemExclusiveParsed` | byte… bytes |
| `fireControllerEventParsed` | `onControllerEventParsed` | byte controller, byte value |
| `fireLyricParsed` | `onLyricParsed` | String lyric |
| `fireMarkerParsed` | `onMarkerParsed` | String marker |
| `fireFunctionParsed` | `onFunctionParsed` | String id, Object message |
| `fireNoteParsed` | `onNoteParsed` | Note note |
| `fireChordParsed` | `onChordParsed` | Chord chord |

**Table 11. ParserListener events and the fire methods that call them**

Just like the `Parser` class does not define a `parse()` method, because the signature of that method would be different for every type of `Parser`, the `ParserListener` class does not define a `getResult()` method. However, your ParserListener should have a `getResult()` method, or something with a similar function but a different name, that returns whatever the specific `ParserListener` needs to return.

`ParserListener` clearly has a lot of methods that implementing classes need to override. Fortunately, JFugue provides `ParserListenerAdapter`, an abstract class that provides empty implementations for all of the `ParserListener` methods. This means that your class can `extend ParserListenerAdapter` instead of `implement ParserListener`, and then you can override only the methods you need.

## Creating Musical Tools with ParserListener

You can do more than convert music from one format to another using `Parser` and `ParserListener`. You can also create "musical tools" that let you do interesting things with your music. For example, you can to data analysis on the notes being parsed; you can change all instruments from one type to another; you can create a new track based on music you hear in other tracks; the list is bounded only by your creativity! And whatever tool you create will work with music in any format that JFugue has a parser for – Staccato, MIDI MusicXML, and so on – including parsers that haven't even been invented yet. `ParserListenerAdapater` helps make it easy to create a musical tool by abstracting all of those `ParserListener` methods (with 22 methods, it is definitely a large interface) so you can focus on the one or two methods that you care about for your tool.

For example, suppose you want to get a list of all of the musical intruments used in a piece of music. You can create an `InstrumentTool` like this:

```
import java.util.ArrayList;
import java.util.List;

import org.jfugue.midi.MidiDictionary;

public class InstrumentTool extends ParserListenerAdapter
{
	private List<String> instrumentNames;

	public InstrumentTool() {
		super();
		instrumentNames = new ArrayList<String>();
	}

	@Override
	public void onInstrumentParsed(byte instrument) {
		String instrumentName =
		MidiDictionary.INSTRUMENT_BYTE_TO_STRING.
		get(instrument);
		if (!instrumentNames.contains(instrumentName)) {
			instrumentNames.add(instrumentName);
		}

	}

	public List<String> getInstrumentNames() {
		return this.instrumentNames;
	}
}
```

Now you can use this `InstrumentTool` like any other `ParserListener`. Here's an example: let's use the `InstrumentTool` to find all of the instruments used in any MIDI file:

```
public static void main(String[] args) throws IOException,
InvalidMidiDataException {
      MidiParser midiParser = new MidiParser();
      InstrumentTool instrumentTool = new InstrumentTool();
      midiParser.addParserListener(instrumentTool);
      midiParser.parse(MidiSystem.getSequence(
          new File("filename")));
      List<String> instrumentNames =
          instrumentTool.getInstrumentNames();
      for (String name : instrumentNames) {
            System.out.println(name);
      }
}
```

For one of my MIDI files, I get the following output:

```
Steel_String_Guitar
Piano
Fretless_Bass
Accordian
Flute
Synth_Strings_1
Electric_Jazz_Guitar
```

Of course, I could use this same tool on music defined in Staccato or MusicXML and it will still work.

Just listing the instruments used in a musical piece is one of the most basic use cases of this capability. You could easily do much more interesting things:

- Determine a Parsons Code for a given musical piece. "Parsons Code for Melodic Contours" indicates whether a melody is going up or down (or remaining the same) during a song. For example, the Parsons Code for "Twinkle, Twinkle, Little Star" is as follows, with the asterisk representing the first note, and *r*, *u*, and *d* for *remain*, *up*, and *down*: `*rururddrdrdrd urdrdrdurdrdrd drururddrdrdrd`. With this bit of information, you might be able to tell what song a person is trying to find just by humming!

- Do a statistical analysis on the notes of a song. Suppose you load and analyze a bunch of songs by Bach, Vivaldi, Handel, and other classical composers. Now, take an unanalyzed song and see if your algorithm can determine who wrote it, based on the similarity of melody, harmony, and cadence. Or, try creating a new song based on your statistical analysis of Bach – determine the next notes in

your composition based on the most likely notes according to your analysis!

- Create a game with the information you get by mapping each word in the lyrics of a song with the notes that are used to play that word. For example, the word "angel" is played differently in J. Geils Bands' "Centerfold", Aerosmith's "Angel", and Sarah McLachlan's "Angel". Players have to match the notes to the right song. (Or, maybe you'll discover that subconsciously, everyone sings "angel" with the same Parsons Code!)

## Chaining Multiple ParserListeners

This is just a friendly reminder about programming with interfaces... Remember that you can add many `ParserListener`s as you want to a single Parser. So you could, for example, run the `InstrumentTool` and something like a `DurationCounter` at the same time by adding both `ParserListener`s to, say, a `MidiParser`.

But wait! What if your `ParserListener` wants to change the stuff that is being parsed? What if you want to *change* the instruments coming in, and you want downstream `ParserListener`s to know about that change? In this case, you would *not* add multiple `ParserListener`s to a `Parser`. Instead, you would set up a chain of `ParserListener`s. You would actually add `ParserListener`s to `ParserListener`s! At that point, you do not want the downstream `ParserListener` to listen to the original Parser, which would now contain outdated data. You instead want the downstream `ParserListener` to listen to the updated data... and anything else that happens to be carried over from the Parser.

This means that your `ParserListener` needs to do something different than implement `ParserListener`. Instead, it needs to extend `ChainingParserListenerAdapter`. That Chaining Adapter both `extends Parser` and `implements ParserListener`, so it acts as both. And whereas `ParserListenerAdapter` provides empty implementations of all of the `ParserListener` methods, `ChainingParserListenerAdapter` provides implementations for each `ParserListener` method that fires the event for the next downstream `ParserListener`. Here is an example that would change all piano instruments to guitars:

```
class InstrumentChangingParserListener extends
ChainingParserListenerAdapter {
      int counter = 0;
    @Override
    public void onInstrumentParsed(byte instrument) {
      if (instrument ==
MidiDictionary.INSTRUMENT_STRING_TO_BYTE.get("PIANO")) {
```

```
            instrument =
MidiDictionary.INSTRUMENT_STRING_TO_BYTE.get("GUITAR");
            counter++;
        }
        super.onInstrumentParsed(instrument);
    }
}
```

And here is how you would use this in an example that reads a MIDI file, converts all piano instruments to guitar, and generates Staccato output. That Staccato output won't have any piano instruments, only guitars. Notice how the `StaccatoParserListener` is added to the `InstrumentChangingParserListener` instead of the `MidiParser`. Also notice that the first `ParserListener` in the chain, the `InstrumentChangingParserListener`, *is* added as a listener to the Parser.

```
public static void main(String[] args) throws
InvalidMidiDataException, IOException {
    MidiParser parser = new MidiParser();
    InstrumentChangingParserListener instrumentChanger = new
InstrumentChangingParserListener();
    StaccatoParserListener staccatoListener = new
StaccatoParserListener();
    instrumentChanger.addParserListener(staccatoListener);
    parser.addParserListener(instrumentChanger);
    parser.parse(MidiSystem.getSequence(new File("your midi
file")));
    System.out.println("Changed "+instrumentChanger.counter+"
instances if Piano to Guitar! "+
staccatoListener.getPattern().toString());
}
```

Remember that you can chain as many `ChainingParserListenerAdapters` as you want to each other. Just be sure to hook up the first one as the listener to the `Parser`.

## Using the DiagnosticParserListener

If you create your own Parser, one way to test your `Parser` is to connect a `DiagnosticParserListener` to it and check its output. JFugue provides the `DiagnosticParserListener` as a developer tool to help see what events are being sent by a `Parser`. Just add the `DiagnosticParserListener` to your `Parser`, and for each musical event, you will either get a print out or a log of the results (you can change whether you get a print out or a log (or both!) in the `print()` method in `DiagnosticParserListener`).

## MusicXML, LilyPond, and Other Parsers and ParserListeners

One of the strengths of JFugue's Parser/ParserListener architecture is that it provides a common interface that many other music formats can understand. This allows for easy integration with other musical tools and systems. And, since any Parser can be connected to any ParserListener, it also means that any new Parser or ParserListener can instantly be used with any other Parser or ParserListener available to JFugue.

Ideally, this should result in a whole bunch of new Parser and ParserListener classes for music formats that the wider community is interested in! In fact, because this is such an easy integration point, and because there are many more formats out there than I have the time to learn, I expect that most of the connection to other formats will be contributed by the JFugue community. At the time of this writing, the range of Parser and ParserListener implementations consists of MIDI, MusicXML, and LilyPond, as shown in Table 12. The MIDI classes are a necessary part of the JFugue core, and those reside in the `org.jfugue.midi` package. The other classes are part of the `org.jfugue.integration` package. Other formats that have been considered include Open Sound Control (OSC) and ABC Notation. There are currently no efforts to write those integration points (hint, hint).

| Format | Parser Name | ParserListener Name | Contribution |
|---|---|---|---|
| MIDI | `MidiParser` | `MidiParserListener` | Part of the core of JFugue |
| MusicXML | `MusicXmlParser` | `MusicXmlParserListener` (currently under development) | Contributed by community |
| LilyPond | (none) | `LilyPondParserListener` | Contributed by community |

**Table 12. Parser and ParserListener implementions in JFugue distribution**

These classes all work the same as any other `Parser` or `ParserListener`:

1. The `Parser`'s `parse()` method will be unique for each parser, but all parsers will `extend Parser` and call the `fireXxx()` methods in `Parser` to fire musical events to listeners
2. The specific method call to get the results of the `ParserListener`'s activity will be unique for each `ParserListener`, but all `ParserListener`s will implement `onXxxParsed()` events
3. Any `ParserListener` can be added as a listener to any Parser

Specific instructions on using the `MusicXMLParser/Listener` or `LilyPondParserListener` are outside the scope of this book, but my expectation is that these will be adequately documented by their author, and I will gladly post those explanations on JFugue.org.

# 5.3

## *Temporal ParserListener-Parser*

If you like time travel, you'll love the Temporal ParserListener-Parser. It's a `Parser` *and* a `ParserListener` at the same time, and it lets you know about musical events in "music time" (the time it takes to play the music) before (or after) the events are actually played. As a `ParserListener`, the Temporal ParserListerner-Parser listens to musical events from any Parser and creates a list of those events indexed by the time at which the events should occur in music time. Recall from the section on Players that music is parsed in a separate loop than it is played. Parsing happens quickly, while playing happens in music time. For example, if you have a MusicXML version of Bach's "Inventio 13", it will take the `MusicXmlParser` a few milliseconds to parse the file, but when you play the song, the music will play for about one minute. But, of course, playing the music simply sends the music to a MIDI synthesizer; no JFugue events are created while music is played. But what if you want to know about the musical events while the music is playing? That's where the Parser portion of Temporal PLP comes into action. Once the ParserListener side of Temporal PLP has its list of musical events indexed by music time, the Parser side goes through those events in music time and fires the events to any other ParserListeners. This means that you can know about music that is being played and, at the same time, you

can get the events that happen at the same time. So, for example, you could display lyrics while music is playing. And if you start the Temporal PLP parser while delaying the Player, you can listen for music events and know about them before they happen. Now you can do really cool things, like create a music training program that shows notes that are about to occur, or create an animated drummer who pulls the drumstick back before striking the drum that is heard through the played music.

## The Temporal ParserListener-Parser

With both `Player` and `RealtimePlayer`, Staccato music is parsed into MIDI as quickly as possible, and the music is then played, taking as long as the music needs to finish playing. If you're interested in getting musical events from the Staccato parser in realtime, neither of these players can provide that. Playing the music is handled with the MIDI classes in `javax.sound.midi`, and there are no JFugue-provided musical events to listen to as the music plays (although if you had a MIDI receiver listening to the MIDI sequence, you would get those MIDI events).

`TemporalPLP` allows you to create a program that receives JFugue-based musical events while the music is being played by the MIDI sequencer. It does this by being both a `ParserListener` *and* a `Parser`. You'll learn more about the `ParserListener` interface in the next section, but for now it's sufficient to know that a `ParserListener` receives musical events when music is parsed by a `Parser` (any kind of parser – a Staccato parser, a MIDI parser, a MusicXML parser, and so on). The `ParserListener` portion of `TemporalPLP` parses the Staccato music provided to it, listens to the resulting events, and records them in a data structure (a map) indexed by track time. Then, the `Player` portion of `TemporalPLP` plays the events in the data structure based on the track time at which they are scheduled to occur.

## Using Temporal PLP

Here's how to use TemporalPLP. First, let's define some music that we want to play, and we'll put this into a variable since we will need to use it twice: once to parse the music the first time, and again to play it.

```
String music = "C D E F G A B";
```

The first step in using `TemporalPLP` is to create a parser that can interpret the music, and add an instance of `TemporalPLP` as a listener. This means that all of the musical events that result from parsing the incoming music will go straight to `TemporalPLP`, rather than being rendered as sound. Of course, you could use any parser here. In this

example, we'll use the `StaccatoParser`. After adding `TemporalPLP` as a listener, we can then parse the music.

```
StaccatoParser parser = new StaccatoParser();
TemporalPLP plp = new TemporalPLP();
parser.addParserListener(plp);
parser.parse(music);
```

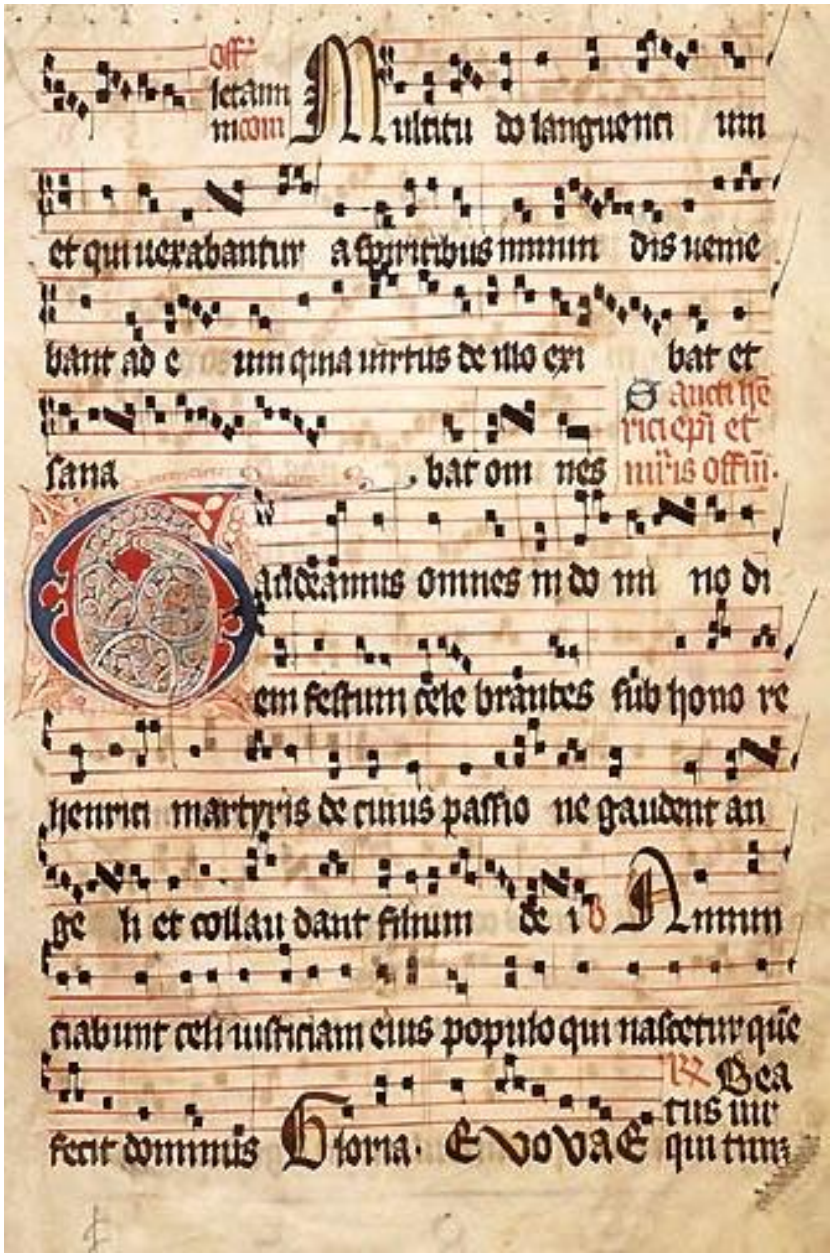
Step 1 is complete! For Step 2, we want to play back the events that `TemporalPLP` recorded in Step 1. We also want to do other things with the music while we are receiving those events – for example, we may want to play the music! In this Step 2 example, we will use the `DiagnosticParserListener`, a JFugue diagnostic tool that prints to `System.out` and also logs all of the messages it receives. We will add that as a parser listener to the `TemporalPLP`. Separately, we will also play the music using the `Player` with a delay of 1000ms (1 second). When we call `parse()` on the `TemporalPLP`, it will go through all of the events it recorded in Step 1 and release those events in realtime based on when they should occur.

```
DiagnosticParserListener dpl = new
     DiagnosticParserListener();
plp.addParserListener(dpl);
new Player().delayPlay(1000, music);
plp.parse();
```

You could do any number of things with the events that `TemporalPLP` is sends out. See what you can create!

# Part 6

# Patterns and Rhythms; MIDI Data and Devices



The *Gaudeamus omnes*, using square notation.
From the 14th-15th century *Graduale Aboense*.

# 6.1

## *Introduction to Patterns*

Patterns are a fundamental unit of music used throughout the JFugue library. At its most basic level, a Pattern wraps a Staccato string, and in fact a new Pattern can be defined by providing a Staccato string to the `Pattern` constructor:

```
// This pattern contains the first notes
// of "Twinkle, Twinkle, little star"
Pattern pattern1 = new Pattern("C5q C5q G5q G5q A5q A5q
Gh");
```

A Pattern is more than just a string. Semantically, a Pattern is expected to be a correctly specified string of Staccato music. Patterns also have known ways of being added together (ensuring that there is a space between two added patterns so the tokens in each pattern remain separate), being transformed into different patterns, repeating phrases of music within the pattern, and more. Finally, within the API, patterns are produced by any class that implements `PatternProducer` – and a lot of classes in JFugue implement `PatternProducer`. For example, `ChordProgression` is a `PatternProducer` and can generate Staccato music given a chord progression and a root note. Patterns are the fundamental unit of music used throughout the JFugue library.

Let's first start with showing how to use patterns to represent phrases of music using the code sample below.

```
// This pattern contains the first notes
// of "Twinkle, Twinkle, little star"
Pattern pattern1 = new Pattern("C5q C5q G5q G5q A5q A5q
G5h");

// This is "How I wonder what you are"
Pattern pattern2 = new Pattern("F5q F5q E5q E5q D5q D5q
C5h");

// This pattern can be used for both "Up above the world
// so high" and "Like a diamond in the sky"
Pattern pattern3 = new Pattern("G5q G5q F5q F5q E5q E5q
D5h");

// This is the full song, combining patterns
Pattern twinkleSong = new Pattern(pattern1, pattern2,
pattern3, pattern3, pattern1, pattern2);

// Now play it!
Player player = new Player();
Player.play(twinkleSong);
```

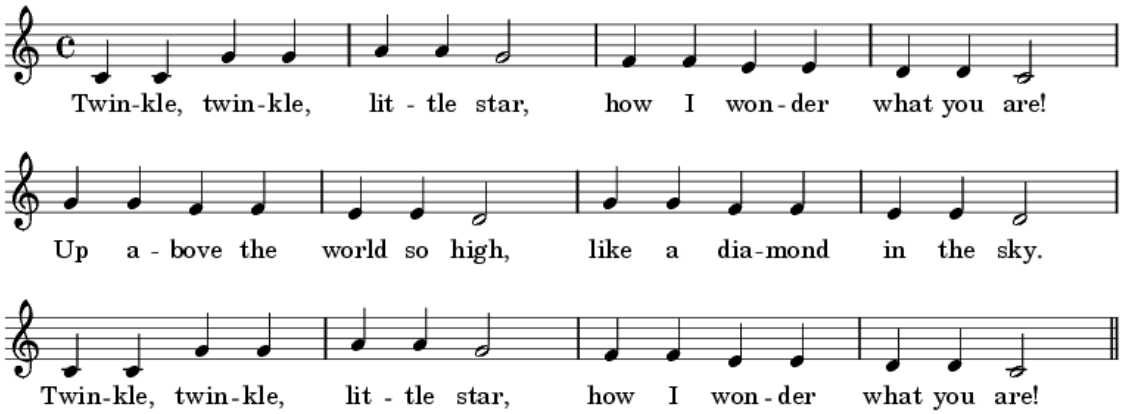



**Figure 14. Sheet music for "Twinkle Twinkle Little Star"**

In the example above, the full song consists of 42 notes, but thanks to patterns you don't have to enter repeated segments of music. A lot of music that we listen to has repeated patterns. For example, the song structure of popular music is often composed of the following individual pieces:

*Intro, Verse, Bridge, Chorus, Verse, Bridge, Chorus, Breakdown, Verse, Bridge, Chorus, Outro*

## Using Patterns to Construct Music

Patterns can also be used as a space for creating new music. You can create an empty pattern, and then add music to the Pattern as you determine what to build.

Behind the scenes, the `Pattern` class uses a `StringBuilder` object to construct the pattern when new musical segments are added. This is more efficient than concatenating `String`s, because when `String`s are concatenated together, new `String` objects are formed and memory need to be allocated.

You can create an empty pattern, and then add either patterns (as shown in the TwinkleSong pattern above) or Staccato strings to the pattern. For example:

```
Pattern pattern1 = new Pattern();
pattern1.add(pattern2);     // Add a Pattern to a Pattern
pattern1.add("C5q C5q");    // Add a String to a Pattern
```

Additionally, the `Pattern` class provides `add()` methods that let you add many things in one method call, and that allow you to indicate how many times to add a particular piece of music:

```
// Add pattern4, pattern5, and pattern6 to pattern1
pattern1.add(pattern4, pattern5, pattern6);

// Add a couple of Staccato strings to pattern1
pattern1.add("C5q", "G5q", "G5q", "Ab5q", "E4h");

// Add pattern3 to pattern1 four times
pattern1.add(pattern3, 4);

// Add G5q to pattern1 3 times
pattern1.add("G5q", 3);
```

Remember that you can add not only patterns, but PatternProducers – anything that creates a pattern. This means that you can also add musical objects, such as notes and chords, to a pattern quite easily:

```
// Add a Note object to pattern1
pattern1.add(new Note(60, 0.5));

// Add a chord progression to pattern 1
pattern1.add(new ChordProgression("I IV V").setRoot("C5"));
```

In addition to adding to a pattern, you can prepend to the beginning of a pattern using the `prepend()` method.

Pattern also provides methods that let you assign a tempo, voice, and instrument. The best use of these methods is when you have a pattern that you know to be a single phrase of music (as opposed to, say, a large pattern with multiple voices). Here is an example:

```
    Pattern p1 = new Pattern("Eq Ch. | Eq Ch. | Dq Eq Dq
Cq").setVoice(0).setInstrument("Piano");
    Pattern p2 = new Pattern("Rw     | Rw     | GmajQQQ
CmajQ").setVoice(1).setInstrument("Flute");
    Player player = new Player();
    player.play(p1, p2);
```

The `repeat()` method lets you repeat the contents of a pattern. The full content of the pattern is simply added to the end of itself as many times as you specify as a parameter. The `clear()` method removes all of the content of the pattern.

The `setTempo()`, `setVoice()`, and `setInstrument()` methods do not alter the patterns themselves, but when `toString()` is called, the tempo, voice, and instrument (in that order) will be prepended to the content of the pattern. These methods can take either numbers (e.g., `setTempo(120)`, `setVoice(5)`, or `setInstrument(110)`) or, for tempo and instrument, strings (e.g., `setTempo("Allegro")` or `setInstrument("Fiddle")`). The value for the strings are looked up in JFugue's `MidiDictionary`.

## Patterns are Composed of Tokens

Each space-separated section of a pattern is referred to as token. The pattern `"Eq Ch. | Eq Ch. | Dq Eq Dq Cq"` has ten tokens: eight note tokens and two barline tokens.

You can get all of the tokens from a `Pattern` using `getTokens()`. This returns a list of `Token` objects, and in the process of separating the Staccato string into tokens, each `Token` instance knows what type of token it is. For example, a token can report, through `getType()`, that it is a Note token – or more specifically, that it is a `TokenType.NOTE`. The Staccato subparsers are used to identify the type of tokens, and the `Subparser` interface has a method, `getTokenType()`, that implementors must implement.

One thing to be aware of is that the list of tokens you get from `getTokens()` are the set of tokens that are generated after each of the preprocessors has been executed on the Staccato string. It is necessary to first run the preprocessors on the string, before separating into tokens. One reason is that spaces are valid within lyrics, markers, and other expressions. For example, `'(this is great)` should be seen as one token, not three. Tokens should also be things that subparsers can

parse, and in many cases, preprocessed elements need to be turned into something else so they can be parsed into musical information. For example, the microtone `m542.6` needs to be turned into pitch bend and note events to be turned into musical events.

With this in mind, the tokens you get back that had spaces will have those spaces replaced with underscores. Calling `getTokens()` on `'(this is great)` will result in one token, `'(this_is_great)`.

One of the nice things about the `Token` class is that it implements `PatternProducer`! The pattern it returns is the token itself. This means that you can create a Pattern by using Tokens. This lets you do creative things like getting all of the tokens from a pattern and scrambling them to create a new pattern, or replacing all of a certain type of token with an alternate value (`if (token.getType() == TokenType.INSTRUMENT) then ...`).

## Adding to a Decorator to Each Note Token in a Pattern

The Note token in a Staccato string can get long, especially when you want to include dynamics (e.g., note on and note off velocity). The length of these tokens can negatively affect the readability of your music. If you find yourself in a situation where every note in a pattern needs to be given the same dynamics, you can use the `addToEachNoteToken()` method to separate this from the flow of the music itself. This method does not actually care whether you are adding dynamics or any other thing to each note in the pattern, although that is probably the most likely use case.

Here's how it works. When you call `addToEachNoteToken()` and pass a string to add, the method gets all of the tokens for the pattern, finds only the Note tokens, and adds your string to the end of each note token. It will then return a pattern (the same pattern object that you started with) with an updated set of contents.

You can also pass a space-separate string to this method. In that case, the method will split your string on spaces and it will cycle through each of your space-separated elements, adding one at a time to each note it finds. For example, a pattern of `"E C D A G"` with adding `"q i"` (demonstrating duration instead of dynamics) will result in a pattern of `"Eq Ci Dq Ai Gq"`. When the elements you provided run out, the list will be run from the beginning again. If your notes run out first – `"E C"` adding `"w h q i s"` – you will get the same number of notes you started with: `"Ew Ch"`.

## Loading and Saving Patterns

You can save a pattern using the `save(File file)` method, and you can load a pattern using the static `Pattern.load(File file)` method. In both cases, the file that is created or read is expected to simply contain one or more Staccato strings in plain text format. One way to use this is to change music that your program plays without needing to recompile your code. You can also use this to export the Staccato music from a pattern that is generated dynamically by a computational algorithm. Well, far be it from me to tell you when you might want to save and load a string of music – I'm sure you'll come up with your own needs, too! And while the ability to write a string to a file, or read a string from a file, is rather basic, it's nice to know that the `Pattern` class provides these methods to make your music programming life that much easier.

Since the pattern files are saved in a plain text format, you can edit them easily. You can also create new lines, and if a new line starts with a hash mark (`#`), it will be treated as a comment (and not as a Staccato "Marker" element). And, the `save(File file, String... comments)` method will insert your comments at the beginning of the file. When you load a file with comments, the comments are ignored.

While not enforced, a saved pattern should ideally use a `.staccato` extension.

Here is an example file that plays Für Elise by Beethoven. Save this text in a text file and name it `furelise.staccato`:

```
#
# "Fur Elise", Ludwig van Beethoven
# Transcribed into Staccato by David Koelle
# http://www.jfugue.org
#

T200

V0 E5s D#5s | E5s D#5s E5s B4s D5s C5s | A4i Rs C4s E4s A4s | B4i
Rs E4s G#4s B4s | C5i Rs E4s E5s D#5s | E5s D#5s E5s B4s D5s C5s
| A4i Rs C4s E4s A4s | B4i Rs E4s C5s B4s | A4q

V1 Ri       | Riii                      | A2s E2s A3s Rsi    | E2s
E3s G#3s Rsi    | A2s E2s A3s Rsi | Riii | A2s E2s A3s Rsi    |
E2s E3s G#3s Rsi | Riii
```

You can then load it and play it with this code:

```
import org.jfugue.pattern.Pattern;
import org.jfugue.player.Player;
```

```
public class LoadJFugueFile {
    public static void main(String[] args) throws IOException {
        Pattern pattern = Pattern.load(new
File("furelise.staccato"));
        Player player = new Player();
        player.play(pattern);
    }
}
```

One of the great things about specifying music this way is that you can change the music without recompiling the code! This is also a great way to share music with other people who use JFugue.

## Transforming and Measuring Musical Data Within Patterns

One of the exciting effects of having music represented within a pattern is that you can easily write code to transform patterns into different patterns – for example, by reversing the notes in a pattern, changing each note's duration, adding tracks from different songs together, and so on. You can also measure the data within a pattern – for example, you can find out which notes are used most frequently, or how often one note follows a sequence of two or more other notes (i.e., Markov chain), how long each track has notes playing versus resting, and so on.

In the past, JFugue provided a `PatternTransformer` class and a `PatternTool` class to assist with transforming and measuring patterns, respectively. However, those classes provided only a very light layer of "syntactic sugar" on the `ParserListener` class. Instead, JFugue 5.0 adds two method to the `Pattern` class: `transform()` and `measure()`. Both take a `ParserListener` as a parameter, and both work the same in that they create a `StaccatoParser`, add the listener provided through the method call, parse the pattern, and return the pattern. The pattern is not changed in this operation, so the value returned by the method is the same as the value when the method was called. This mechanism keeps the flow of the fluent API – just remember that even if you call `transform()`, the pattern remains unchanged. You will then need to call the specific methods on your `ParserListener` that you created to return the results of your listener. (There is no difference in how `transform()` and `measure()` work – in fact, they call the same common code within the `Pattern` class. The reason for the two method names is to make your intention clear when you write your code. It is easier to understand "transform" and "measure" separately than to try inventing a phrase that represents them both clearly.)

Recall the `InstrumentTool` example from the section, Creating Musical Tools with ParserListener. You can use the same `InstrumentTool` class with the `measure()` method:

```
public static void main(String[] args) throws IOException,
InvalidMidiDataException {
      InstrumentTool instrumentTool = new InstrumentTool();
      Pattern pattern = new Pattern("V0 I[Piano] A B C V1
I[Flute] D E F");
      pattern.measure(instrumentTool);

      List<String> instrumentNames =
          instrumentTool.getInstrumentNames();
      for (String name : instrumentNames) {
            System.out.println(name);
      }
}
```

# 6.2

# *TrackTable: A Table of Patterns*

Patterns are a nice way to represent recurring musical phrases, but JFugue 5.0 introduces a new concept that takes song development one step further. `TrackTable` is a collection of patterns that spans both time (or song length) and tracks. It's like a spreadsheet, where each row is one track, each column is one segment of song, and each cell may contain a `PatternProducer`. The TrackTable API also provides some unique ways of populating the table.

## Creating and Populating a Track Table

When you create a TrackTable instance, you pass the size of the table, which is how many columns-worth of patterns the table should be able to hold. The constructor will automatically provide 16 rows, one for each track or voice. Remember that the 10th row is reserved for percussion.

You also need to pass a double value representing the duration of each segment, where 1.0d represents a whole duration. TrackTable will automatically populate each cell of the table with a default pattern containing the voice and a rest of the given duration, such as `V0 R/1.0`. This provides the timing necessary so when you start populating the table with patterns, the patterns will play at the expected time.

To illustrate this, calling `TrackTable(5, 1.0d)` will result in a table like the following:

| V0 | R/1.0 | R/1.0 | R/1.0 | R/1.0 | R/1.0 |
|---|---|---|---|---|---|
| V1 | R/1.0 | R/1.0 | R/1.0 | R/1.0 | R/1.0 |
| V2 | R/1.0 | R/1.0 | R/1.0 | R/1.0 | R/1.0 |
| V3 | R/1.0 | R/1.0 | R/1.0 | R/1.0 | R/1.0 |
| V4 | R/1.0 | R/1.0 | R/1.0 | R/1.0 | R/1.0 |
| V5 | R/1.0 | R/1.0 | R/1.0 | R/1.0 | R/1.0 |
| V6 | R/1.0 | R/1.0 | R/1.0 | R/1.0 | R/1.0 |
| V7 | R/1.0 | R/1.0 | R/1.0 | R/1.0 | R/1.0 |
| V8 | R/1.0 | R/1.0 | R/1.0 | R/1.0 | R/1.0 |
| V9 | R/1.0 | R/1.0 | R/1.0 | R/1.0 | R/1.0 |
| V10 | R/1.0 | R/1.0 | R/1.0 | R/1.0 | R/1.0 |
| V11 | R/1.0 | R/1.0 | R/1.0 | R/1.0 | R/1.0 |
| V12 | R/1.0 | R/1.0 | R/1.0 | R/1.0 | R/1.0 |
| V13 | R/1.0 | R/1.0 | R/1.0 | R/1.0 | R/1.0 |
| V14 | R/1.0 | R/1.0 | R/1.0 | R/1.0 | R/1.0 |
| V15 | R/1.0 | R/1.0 | R/1.0 | R/1.0 | R/1.0 |

Since `TrackTable` is a `PatternProducer`, you can play it with `Player.play()`. But if you played this example, you wouldn't hear much yet. Let's start putting patterns into the table!

`TrackTable` provides a variety of `put()` commands:

- `put(int track, int position, PatternProducer p)` does exactly what you think it will. The pattern you provide will be placed in the expected cell of the table.

- `put(int track, int start, int end, PatternProducer p)` will place the given pattern in each cell of the given track from start to end, including in the start and end cells. `put(0, 2, 4, p)` will place *p* three times, in columns 2, 3, and 4.

- `put(int track, int start, PatternProducer… ps)` will place each of the patterns given as a parameter one after another in subsequent columns from the start. `put(0, 5, p1, p2, p3)` will place *p1* in column 5, *p2* in column 6, and *p3* in column 7.

- `put(int track, String placementPattern, PatternProducer p)` requires a discussion of its own – see below.

- `putAtIntervals(int track, int nth, PatternProducer p)` will place the given pattern in every *nth* position starting with the first position. `putAtIntervals(0, 2, p)` will place *p* in position 1 and 3 (given a maximum table size of 5).

- `putAtIntervals(int track, int first, int nth, int end, PatternProducer p)` will place the given pattern in position *first*, then every *nth* position after that until *end*. `putAtIntervals(0, 1, 2, 4, p)` will put place *p* in positions 1 and 3.

- `put(Rhythm rhythm)` places a Rhythm in the TrackTable, and it's intelligent about how it does this: it puts each segment of the Rhythm into each respective column of the TrackTable. A nice, seamless interaction between two parts of the JFugue API!

Now for that special `put()` method with the placement pattern. This takes a specially formatted string in which each character of the string represents a column of the table. If the character is a period, the cell is untouched. If the character is any alphanumeric character, the pattern is placed in the cell. If the character is a dash, the content of the cell is cleared (you would use this if you intend to place a pattern that you know will be longer than the duration of a single cell).

Let's look at an example.

```
TrackTable t = new TrackTable(5, 1.0d);
t.put(0, "X-.X-", new Pattern("Cmajh Emajh Gmajh Emajh"));
t.put(1, ".X..X", new Pattern("Gq Cq Eq Gq"));
```

This would result in a table like the following (for only the first two rows):

| | | | | | |
|---|---|---|---|---|---|
| V0 | Cmajh Emajh Gmajh Emajh | | R/1.0 | Cmajh Emajh Gmajh Emajh | |
| V1 | R/1.0 | Gq Cq Eq Gq | R/1.0 | R/1.0 | Gq Cq Eq Gq |

Now you could pass that `TrackTable` to `Player.play()` and hear the music!

In addition to `put()` methods, `TrackTable` has a `get(int track, int position)`, `clear(int track, int position)`, and `reset(int track, int position)` method. The `clear()` method places an empty pattern in the position; the `reset()` method places a rest with the duration given in the constructor. Given JFugue's fluent API, the `put()`, `clear()`, and `reset()` methods all return the `TrackTable` instance itself, so you can chain these together (`t.put().put().put()`...). Finally, `getLength()` will return the number of columns in the table.

## Managing Track Settings

Each track in your table will probably have some settings that you do not need to repeat for each cell. For example (and maybe specifically), you will probably need to declare the instrument for each track only once. TrackTable has a `setTrackSettings(int track, PatternProducer p)` and a corresponding `getTrackSettings(int track)` that let you provide a pattern that will be prepended to the pattern produced by the table.

Let's build on our example from earlier:

```
TrackTable t = new TrackTable(5, 1.0d)
 .put(0, "X-.X-", new Pattern("Cmajh Emajh Gmajh Emajh"))
 .put(1, ".X..X", new Pattern("Gq Cq Eq Gq"))
 .setTrackSettings(0, "I[Flute]")
 .setTrackSettings(1, "I[Piano]");
```

This one chained line of code (not the use of the fluent API!) would result in a table like the following (for only the first two rows):

| | | | | | |
|---|---|---|---|---|---|
| I[Flute] | Cmajh Emajh Gmajh Emajh | | R/1.0 | Cmajh Emajh Gmajh Emajh | |
| I[Piano] | R/1.0 | Gq Cq Eq Gq | R/1.0 | R/1.0 | Gq Cq Eq Gq |

The settings are maintained in a list that is separate from the table, but they are added to the pattern when the table's pattern is returned from `getPattern()`.

# 6.3

## *Rhythms*

Just when you thought JFugue couldn't get any cooler, along come Rhythms. Imagine creating an awesome-sounding rock-n-roll beat in practically four lines of code (well, technically, you could do it in one line of code due to JFugue's fluent API, but let's not get carried away).

### Introduction to Rhythms

The Rhythm class provides a natural and intuitive way to specify rhythms and drum beats. To make this possible, the Rhythm class lets you specify a beat by hammering out a string. For example, you might imagine sitting at your computer and hammering out this little beat, where the X's are times that you tap your desk, and the periods are pauses:

```
X..XX...X..XXX..
```

Try drumming that with your hand on your desk right now. Really, go ahead and try it! As you are drumming this out on your desk (you are doing it, aren't you?), you might find that your other hand, or perhaps one of your feet, is anxious to join in the beat. In fact, your two hands

together, or one of your hands and one of your feet, may be drumming this beat:

**X..XX...X..XXX..**

**..O...O...O...O.**

(and if your other hand or foot didn't already start tapping this out, don't despair. You can do it now!)

If this makes sense to you, then you'll have no problem using the `Rhythm` class. It has a method that takes strings exactly like this, and converts them into a Pattern that you can pass to the Player.

The notation it uses is clearly different from the Staccato music that you saw earlier. This is intentional: it helps you focus on the beat rather than the content and length of elements in the Staccato string. The Rhythm class will convert the X's, O's, and periods to a Staccato string when the Rhythm's `getPattern()` method is called. It makes that conversion based on a dictionary called a "rhythm kit," which maps each X, O, period, or any other character to a Staccato replacement for that character.

The following example uses the Rhythm class to create an 8-beat rock rhythm:

```
Rhythm rhythm = new Rhythm();
rhythm.addLayer("O.OO...O.OO....O");
rhythm.addLayer("....o.......o...");
rhythm.addLayer("^.`.^.`.^.`.^.`.");
player.play(rhythm.getPattern().repeat(4));
```

Give that a try (or run `RhythmDemo.java` in the JFugue source code) and see what you think.

JFugue provides a default rhythm kit that contains the dictionary entries mapping each character in the rhythm above to a Staccato string representation. You will probably find that the default dictionary is sufficient, but if you ever want to create your own, just pass a `Map<Character, String>` to `setRhythmKit()`. Table 13 shows the dictionary entries in the default rhythm kit (which is defined at the end of `Rhythm.java`). A convention used in the default rhythm kit is that uppercase characters correspond to eighth notes, and lowercase characters correspond to the same drum but as a sixteenth note, preceded by a sixteenth rest. The assumption built into the default rhythm kit is that rhythms will each contain 16 eighth notes, and one segment of the rhythm is a 2/4 bar (total duration equal to two whole notes, or `ww`). You are free to design your own rhythm kits that use a different meter.

| Character | Staccato string |
|---|---|
| . | Ri |
| O (uppercase) | [BASS_DRUM]i |
| o (lowercase) | Rs [BASS_DRUM]s |
| S (uppercase) | [ACOUSTIC_SNARE]i |
| s (lowercase) | Rs [ACOUSTIC_SNARE]s |
| ^ (caret) | [PEDAL_HI_HAT]i |
| ` (back tick) | [PEDAL_HI_HAT]s Rs |
| * (asterisk) | [CRASH_CYMBAL_1]i |
| + (plus) | [CRASH_CYMBAL_1]s Rs |
| X (uppercase) | [HAND_CLAP]i |
| x (lowercase) | Rs [HAND_CLAP]s |

**Table 13. DefaultRhythmKit entries**

Keep in mind that all characters in a rhythm must have an entry in the dictionary. You cannot use regular Staccato elements when defining a Rhythm – you can only place Staccato in the rhythm kit dictionary. For example, you cannot add a bar line or another note within the Rhythm definition itself.

The Staccato output from a rhythm is played with the 10th MIDI channel (known in JFugue as Voice 9 or `V9`), which is the channel on which percussion instruments are played. In fact, elements like `[BASS_DRUM]i` are actually MIDI notes. In any other channel, these notes would be played chromatically, but in the 10th MIDI channel, notes correspond to percussion sounds. See Section 2.3 for more information on this special MIDI channel.

## Layers

As discussed in Section 2.3, there can be 16 layers within the 10th MIDI Channel. This means that there can be 16 notes played simultaneously (i.e., in harmony) within Voice 9. This is accomplished by the use of JFugue's layers.

In a Rhythm, each time you call `addLayer()`, you are adding an additional JFugue layer to Voice 9. Therefore, there is a limit (as indicated by the MIDI specification) of 16 layers that can be added to a Rhythm.

When the rhythm is converted to a pattern, you will see the `V9` indicator at the beginning of the resulting pattern, and you will see layer indicators, such as `L0`, for each of the layers.

## Getting Rhythms and Patterns

There are two types of computed results that you can get from a Rhythm. The first is the full-length Rhythm, with all layers and alternate layers (described below), expressed with the same characters as you initially described the rhythm. To obtain this, call the `getRhythm()` method.

The second is the `Pattern` that the `Rhythm` produces. As a class that implements `PatternProducer`, `Rhythm` provides a `getPattern()` method. This method returns the full rhythm with each character expanded to its Staccato string.

And, since `Rhythm` implements `PatternProducer`, you can send a `Rhythm` directly to `Player.play()` to hear your rhythm.

## Length and Segments

By default, the length of a rhythm is 1. This means that each rhythm string you provide as a layer is added one time when you call `getPattern()` or `getRhythm()`. If you provide a value to `setLength()`, you can set how many times you want each layer string to be repeated in the resulting pattern. For example, a rhythm with a layer of "`S..`" and a length of 3 would result in a final rhythm of "`S..S..S..`".

In the code, each instance of a rhythm pattern that is repeated to complete the length is a segment. In this example, there are three segments. They all happen to be "`S..`" in this case, but the use of alternate layers (described below) could cause different rhythms to be added during different segments as the full, length-long rhythm is computed.

## Combining Rhythms

There is a static method on Rhythm that lets you combine multiple Rhythms into a single Rhythm. This will cause all of the layers, rhythm kit dictionaries, and alternate layers from each of the rhythms in the parameter to be added together and returned in a new instance of Rhythm.

This is especially useful if you have portions of a Rhythm that you would like to merge together to form a more complete Rhythm. For example, you may have several commonly-used sequence of hi hats, and separately you might have several standard snare beats; and separately still you might have a favorite pattern of hand claps. You can select from your collection and combine any of these rhythms together to form a single rhythm.

## Alternate Layers

If you are crafting a rhythm for a song, you may realize that the rhythm is not identical during all portions of the song. During an intro or breakdown, there may be a different rhythm, at the end of each chorus, there may be a slightly different drum roll or accent, and so on. To support these cases, the Rhythm API provides for alternate layers that will be substituted for the base layer when the full length of the rhythm is computed.

There are four manners in which alternate layers may be defined:

1. **Recurring:** Every nth segment will be replaced with this rhythm.
2. **Ranged:** Segments from a given start to a given end will be replaced with this rhythm.
3. **One Time:** A specific segment will be replaced with this rhythm.
4. **By a unique function:** Given a segment number, you can write code to determine what rhythm to use to replace the base rhythm.

The four methods that you can use to define these alternate layers are as follows. Each of these takes the layer number as the first parameter, and additional parameters as noted:

1. **addRecurringAltLayer** takes a start and end segment number, and the "every nth" from the start. The start index counts as the first recurrence. For example:
2. **addRangedAltLayer** takes a start and end segment number.
3. **addOneTimeAltLayer** takes a single segment number.
4. **addAltLayerProvider** takes a `RhythmAltLayerProvider`, which is an interface with a single method, `provideAltLayer()`, that takes the current segment number. Code that you write within `provideAltLayer()` will use the segment number to determine what rhythm to return, or null if the base rhythm will not be replaced.

Here is an example of alternate layers in action. This example contains a one time layer, a ranged layer, and a recurring layer.

```
import org.jfugue.player.Player;
import org.jfugue.rhythm.Rhythm;

public class AdvancedRhythms2 {
    public static void main(String[] args) {
        Rhythm rhythm = new Rhythm().addLayer("O..oO...O..oOO..")
```

```
                .addLayer("..S...S...S...S.")
                .addLayer("````````````````")
                .addLayer("..............+")
                .addOneTimeAltLayer(3, 3, "...+...+...+...+")
                .addRangedAltLayer(2, 1, 2, "`...`.``...```.`")
                .addRecurringAltLayer(1, 1, 4, 2, ".sS..sS..sS..sS.")
                .setLength(8);
            new Player().play(rhythm.getPattern());
        }
}
```

## Z-Ordering of Layers

The base layer and each of the alternate layers has a z-order that determines which layers have precedence over which other layers if multiple alt layers are assigned to a given segment.

The base layer—the layer that consists of rhythms using the `addLayer()` method—always has a z-order of 0. It is the first layer that will be used for every segment of the full rhythm, unless determines otherwise.

Each of the methods to add an alt layer can optionally take a final parameter that is a z-order that you specify. If you do not specify a z-order, the alt layers have defaults. After the base layer, the following alt layers will be added in this order:

1. Recurring
2. Ranged
3. One-Time
4. Alt Provider

# 6.4

## *Working with MIDI Data*

In the next section, you will learn more about how JFugue works with MIDI data. One of the things you can do with JFugue is turn an existing MIDI file into Staccato music. This section covers some of the nuances of that translation.

### Expectations When Translating MIDI to Staccato

As you may know, MIDI is a set of musical instructions that tells a synthesizer how to play music. Each of the MIDI events is associated with a timestamp at which that event is meant to be used. In MIDI, when a note should be played, the NOTE_ON MIDI Event is sent with the note number, and when that same note is to be turned off, a NOTE_OFF MIDI Event is sent with the same note number. MIDI does not have concepts like rests or duration with respect to a time signature (e.g., there is no 'whole note' in MIDI).

Staccato, on the other hand, is not based on time. Instead, music in Staccato is built in sequential order and with durations that have musical meaning. Instead of asking for Note 60 at 5 seconds into a song, you'd have to first create a rest, and give it a duration for as many whole notes as 5 seconds translates to.

When MIDI is converted into Staccato, JFugue needs to compute the rests and durations for notes, since rests are not represented in MIDI (a rest simply means there are no musical events), and durations are expressed in realtime rather than musical time.

Often, the music you'll find in a MIDI file has been recorded from a person playing a MIDI device, rather than being entered by a computer system. You didn't hear it from me, but it turns out that people are not perfect, and whereas a person playing a musical instrument might attempt to play quarter or whole notes, or wait a sixteenth or a half rest between notes, they might be slightly off. JFugue attempts to translate time between notes into known durations, like whole, half, or quarter notes, but sometimes, JFugue is unable to do that translation with certainty, which will result in durations specified with decimals.

JFugue also makes use of the `@` time command when translating MIDI to Staccato.

That's an overview of two things you can expect to see when MIDI is translated to Staccato: Bizarre note and rest durations, and lots of `@` commands. Let's now see how we can turn MIDI into a Staccato pattern in the first place.

## Turning MIDI into a Pattern, and Vice Versa

By now, you can probably guess that turning MIDI data into a Staccato pattern is going to make use of something like a `MidiParser` and something like a `StaccatoParserListener`. Well, that's exactly what happens! There is indeed a `MidiParser`, and it adds a `StaccatoParserListener` as a listener. Rather than have you write this pattern yourself, `MidiFileManager` provides it for you – as well as the inverse: turning Staccato to MIDI using a `StaccatoParser` and a `MidiParserListener`.

The methods are static methods in `MidiFileManager` called `savePatternToMidi()` and `loadPatternFromMidi()`. Both take a `File` representing the MIDI file to either save or load (or they can take an `OutputStream`/`InputStream`). `savePatternToMidi()` also takes a `Pattern`, and `loadPatternFromMidi()` returns a `Pattern`. You'll need a try/catch block around each to catch an `IOException` and, in `loadPatternFromMidi()`'s case, an `InvalidMidiDataException`.

That's all there is to it! Now you can load up your favorite MIDI song as a `Pattern`, transform and measure it, pull it apart and throw it in a `TrackTable` – whatever you can imagine!

# 6.5

# *Using JFugue with MIDI Devices*

This chapter demonstrates how JFugue can be used to communicate to external MIDI devices, such as musical keyboards, mixers, and more.

## Why Communicate with External Devices?

The ability to easily interact with external devices increases the degree to which one can experiment with music. Wouldn't it be great if you could hear your JFugue song played on your keyboard? Wouldn't you have a lot of fun making music with your keyboard, while recording the Pattern that you generate, and then modify the pattern in interesting ways using the JFugue API?

The MIDI classes that come with the Java Development Kit (JDK) get you part of the way there. There is still a lot of additional work that needs to be done to make the combination of software and external devices seamless and easy to program in only a few lines of code. This is where JFugue steps in.

## Setting Up Communication with External Devices

If you'd like to experiment with the code in this chapter, you will need a MIDI device, preferably a keyboard, and a MIDI connector. You should be

able to connect your MIDI device to your computer using a MIDI-to-USB connector, and you should ensure that you have the correct drivers installed for your operating system. Note that you may need a driver from the manufacturer of your MIDI-to-USB cable, rather than a drive from the manufacturer of your MIDI device.

You may also be able to set up a MIDI loopback on your system, which would allow you to use JFugue to control MIDI-based music editing software, such as Propellerhead Reason.

JFugue has three classes for interacting with external devices, all of which can be found in the `org.jfugue.devices` package:

1. `MusicReceiver`, which sets up your external device as something that can receive music from your program;

2. `MusicTransmitterToParserListener`, which sets up your external device as a music transmitter, and when your device transmits music, it will be received by a ParserListener that you can listen to;

3. `MusicTransmitterToSequence`, which also sets up your external device as a music transmitter, and will record music sent from the device to a MIDI Sequence.

Each of these needs to be constructed with the device you want to connect to. The only way to know what devices are available is to list them and select the one that makes the most sense. JFugue provides a text-based tool, `MidiDevicePrompt`, to help you with this. In your sample program, simply call `MidiDevicePrompt`'s static `askForMidiDevice()` method, and you will see a list of available devices, printed to System.out, followed with a prompt asking for the device you would like to select. The `askForMidiDevice()` method returns an instance of a `MidiDevice` for the selected device (`MidiDevice` is a Java class from `javax.sound.midi`). You can use `MidiDevicePrompt` for your own experiments, but if you want to create a user-facing application, you will probably want to implement your own user interface for selecting the MIDI device. (Previous versions of JFugue tried to be "intelligent" about device selection by picking the device that had a name most similar to a MIDI input device name, but this was not sufficiently reliable and the old `IntelligentDeviceResolver` from JFugue 4 and earlier has been removed from the API.)

## Sending Music to a MIDI Device

If you have a MIDI file on your computer, it's pretty fun to send it to your keyboard and hear what it sounds like for your keyboard to play it. You can do this very easily using JFugue, although interestingly, there is a lot

that happens behind the scenes to make this possible. JFugue needs to sort all of the events in the sequence obtained from the MIDI file in chronological order, then send them to your device at the right time. Here is the only code you need to write:

```
try {
      MidiDevice device = MidiDevicePrompt.askForMidiDevice();
      MusicReceiver r = new MusicReceiver(device);
      File file = new File("yourfile.mid");
      Sequence sequence = MidiSystem.getSequence(file);
      r.sendSequence(sequence);
} catch (Exception e) {
      e.printStackTrace();
}
```

Let's break this down. First, you need to know which MIDI device to send the music to. Then, create a `MusicReceiver()` instance passing in that device. Open the MIDI file and send it to the device, and you're done!

If you really want to see the innards of what `r.sendSequence()` is doing, check out `MidiTools.sendSortedMessagesToReceiver()`.

## Listening to Music from a MIDI Device

After you have tested your connectivity by successfully sending music to your device, now let's listen to the music you play on your device! As mentioned previously, there are two classes that will direct the music to two different places: one to a MIDI Sequence, and one to any ParserListeners that you specify. Both of these listeners let you listen to music either:

1. Between when you call `startListening()` and when you call `stopListening()`, *or...*
2. For a set number of milliseconds provided in `listenForMillis(long milliseconds)`

Let's look at the Sequence one first. Here is an example showing how to listen for a Sequence using the `startListening()` and `stopListening()` methods:

```
try {
      MidiDevice device = MidiDevicePrompt.askForMidiDevice();
      MusicTransmitterToSequence transmitter =
          new MusicTransmitterToSequence(device);
      DemoPrint.step("Press [ENTER] when you're ready to start
playing...");
      Scanner scanner = new Scanner(System.in);
      scanner.nextLine();
      transmitter.startListening();
```

```
      DemoPrint.step("Press [ENTER] when you're ready to stop
playing...");
      scanner.nextLine();
      scanner.close();
      transmitter.stopListening();

      // Now you can call transmitter.getSequence()...
} catch (MidiUnavailableException e) { e.printStackTrace(); }
} catch (InvalidMidiDataException e) { e.printStackTrace(); }
```

You might imagine creating a user interface for listening to the device that uses buttons to start and stop the listening. In fact, this would be a great way to make a musical game – maybe one that teaches you how to play the keyboard, or one in which certain correctly-played chords power the lasers you use to eliminate the alien enemy.

Listening for a set duration of time instead of using the start/stop methods is pretty easy. Instead of that call to `transmitter.startListening()`, you can instead say:

```
      transmitter.listenForMillis(15000);
```

And, in this example, you will be able to get your music 15 seconds later. This method will actually wait for the specified time, so keep that in mind if you are working with threads.

Using `MusicTransmitterToParserListener` is roughly identical. The difference is that you first need to call `addParserListener()` to add a ParserListener (you can add as many as you like), and instead of calling `transmitter.getSequence()` at the end, you call methods directly on your ParserListeners. Remember that if you are experimenting with this capability, you could use the `DiagnosticParserListener` as a quick tool to see that music is being sent and your code is working as you expect.

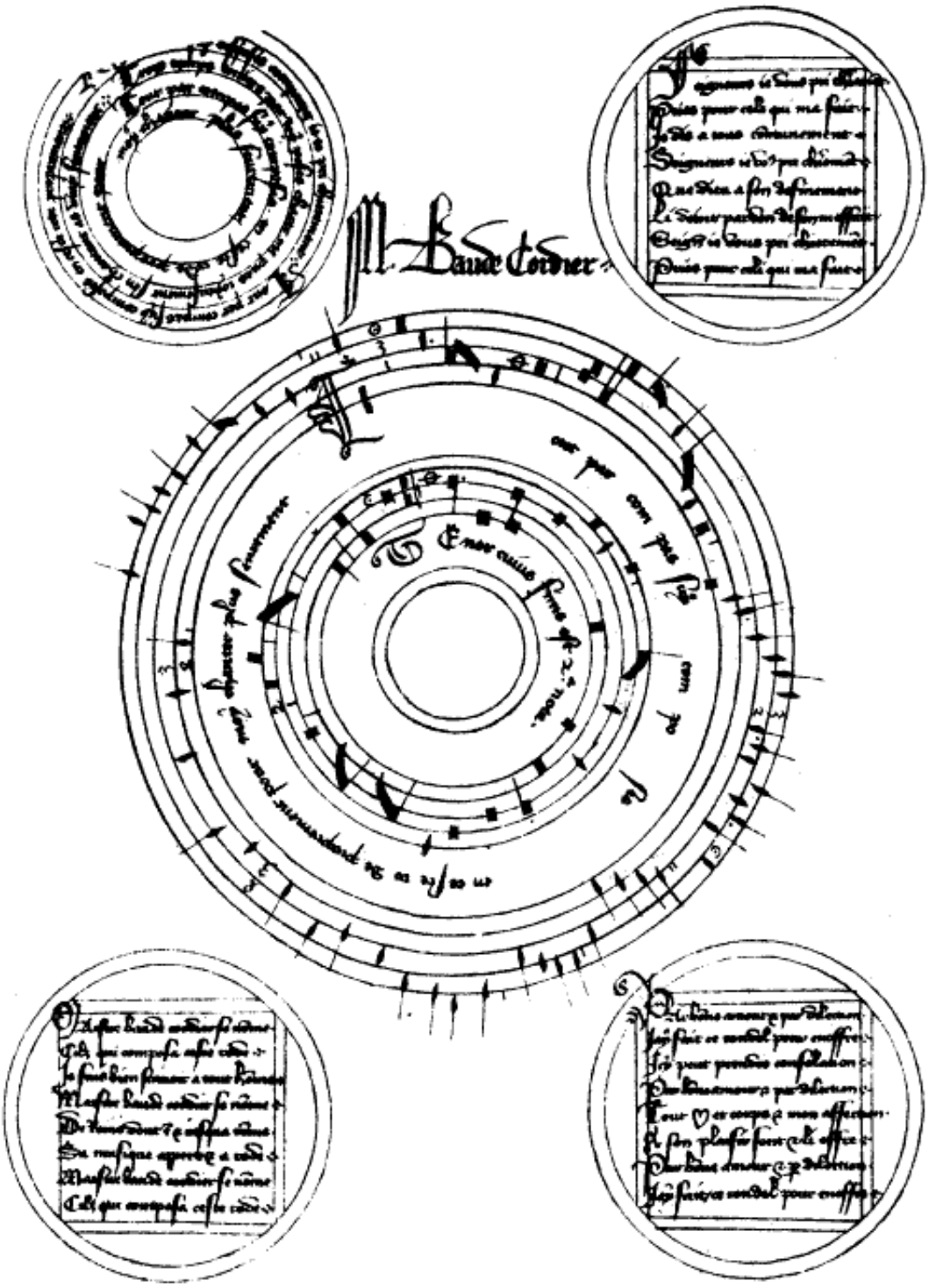
## Troubleshooting Your Connections

If you are having difficulty getting the examples to run, try plugging your MIDI-to-USB cable in a different USB port, and make sure that you are using a driver from the manufacturer of the MIDI-to-USB cable rather than the manufacturer of your MIDI device.

If you have your MIDI device hooked up to your computer and the examples above compile and run without errors, but you aren't getting the music you'd expect, try swapping the In/Out MIDI connectors on the MIDI device. The OUT connector on the MIDI-to-USB cable needs to be plugged into the IN port on the MIDI keyboard, and the IN connector needs to be plugged into the OUT port.

# Part 7

## Extras and Examples




"Tout par compas suy composes"
by Baude Cordier (ca. 1380 – ca. 1440)

# 7.1

# *A Quartet of Demonstrations*

This chapter presents four demonstrations that showcase the use of everything you have learned so far.

## J. S. Bach's Crab Canon with Tokens and Pattern Reversal

Johann Sebastian Bach was known for playfully experimenting with music through his various fugues, canons, and other compositions. One of his pieces, part of his "A Musical Offering" collection, is the Crab Canon. The Crab Canon consists of a sequence of notes played in harmony with the reverse of the same sequence of notes. In his book *Gödel, Escher, Bach*, Douglas Hofstadter refers to this piece in a remarkable dialog that is presented as a palindrome.

This example splits a Pattern into Tokens so it can create a reverse sequence of notes. Each note in the transformed set is also lowered by one octave. It is worth noting that reversing a pattern is not always trivial. For a pattern consisting simply of notes, like `"G5h A5q Bb5i"`, it is easy to see that the reverse would be `"Bb5i A5q G5h"`. Or, should the reverse be `"Bb5h A5q G5i"` – reversing the values of the notes but maintaining durations in the same place? Regardless, the problem gets even trickier when you add voice and instruments. The reverse of `"V0`

`I[Piano] A B C V1 I[Flute] D E F"` (let's call this Pattern 2) is definitely not `"F E D I[Flute] V1 C B A I[Piano] V0"` because the voice and instrument changes are clearly intended to modify the notes that follow them! So, one might think (as I started to) that perhaps in Staccato, there are tokens that get reversed, and tokens that maintain their position. It was easy enough for me to write a reversing function that might correctly reverse Pattern 2 as `"V0 I[Piano] C B A V1 I[Flute] D E F"`. Better! But then, what happens to `"V0 I[Piano] A B V1 I[Flute] C V0 I[Piano] D E"` – should this be `"V0 I[Piano] E D V1 I[Flute] C V0 I[Piano] B A"`? In the first case, `V1`'s `C` is played at the same time as `V0`'s `A`. In the second case, `V1`'s `C` is played with `V0`'s `E`. Is that correct? Well, yes and no. The voices are reversed properly, if that was the intention of the person doing the reversing, but the timing was not preserved. So instead, this could be reverse-justified as `…V1 I[Flute] Rq Rq Rq Rq C…`, but is that right? This discussion is why there is no ReversePatternTransformer in JFugue 5.0.

Back to the Crab Canon! All we want to do is simply reverse a bunch of things that we know will only be notes, and we want them to maintain their duration, so this is easy. Let's start by taking a look at the pattern we want to change. Here is Bach's upper melody for the Crab Canon.

```
// Upper melody of Bach's Crab Canon
Pattern meloldy1 = new Pattern("D5h E5h A5h Bb5h C#5h Rq A5q "+
                "A5q Ab5h G5q G5q F#5h F5q F5q E5q Eb5q D5q "+
                "C#5q A5q D5q G5q F5h E5h D5h F5h A5i G5i A5i "+
                "D6i A5i F5i E5i F5i G5i A5i B5i C#6i D6i F5i "+
                "G5i A5i Bb5i E5i F5i G5i A5i G5i F5i E5i F5i "+
                "G5i A5i Bb5i C6i Bb5i A5i G5i A5i Bb5i C6i D6i "+
                "Eb6i C6i Bb5i A5i B5i C#6i D6i E6i F6i D6i "+
                "C#6i B5i C#6i D6i E6i F6i G6i E6i A5i E6i D6i "+
                "E6i F6i G6i F6i E6i D6i C#6i D6q A5q F5q D5q");
```

Now we want the reverse of that. We'll call this the "simple reverse," and it goes like this… plus, we'll change the octave of the note while we're at it. Notice the use of `Note`'s `changeValue()` (which takes a delta from the current note value) and `Note.OCTAVE` to help make this easy, and the `prepend()` method on `Pattern` to add to the beginning of the pattern instead of the end:

```
// Create second melody using a simple reverse
Pattern melody2 = new Pattern();
for (Token token : melody1.getTokens()) {
    Note note = new Note(token.toString())
                .changeValue(-Note.OCTAVE);
    melody2.prepend(note);
}
```

Now all we have to do is set some voices and instruments on both patterns, and play them!

```
melody1.setVoice(0).setInstrument("Piano");
melody2.setVoice(1).setInstrument("Piano");
Player player = new Player();
player.play(melody1, melody2);
```

And there you have it, Bach's Crab Canon! This boils down to about nine lines of code. Honestly, I think Bach would have been a huge fan of JFugue if he were around to experience it, and he would have played with something like this.

## Lindenmayer System Music with a Replacement Map

The Hungarian theoretical botanist Aristid Lindenmayer originally developed Lindenmayer systems, or L-systems, to formally describe the development of types of algae; this was later used to represent plants and fractals. An L-system is a grammar rewrite system, which means it consists of a set of rules for replacing one character in a string with a sequence of other characters. For example, if I give you a string, like "A", and tell you to change every "A" to "B", and every "B" to "AB", you'll wind up with the following sequence of strings:

```
Initial string     A
Iteration 1        B
Iteration 2        AB
Iteration 3        BAB
Iteration 4        ABBAB
Iteration 5        BABABBAB
Iteration 6        ABBABBABABBAB
Iteration 7        BABABBABABBABBABABBAB
…and so on
```

If you then take the resulting string and convert the characters to some actionable result—for example, drawing a picture using A and B as commands for moving a pen on a drawing surface—you may get an interesting result. You can see some cool results on the Wikipedia page for L-system, and some especially beautiful results are presented in the book, "The Algorithmic Beauty of Plants" by Przemyslaw Prusinkiewicz and Aristid Lindenmayer. This book is now available for free on the site.

What does this have to do with music? L-systems exhibit self-similarity (which is why L-system fractals are interesting), and music itself is often self-similar. And instead of transforming characters, we can actually transform Staccato tokens! Of course, we also know that we can play Staccato tokens, so we have our actionable result. Here's an example; you can follow along with the code below by finding `LSystemMusic.java` in the `demo` section of the JFugue library.

First, let's create a set of transform rules. When the preprocessor sees the key, it will replace it with the value. Notice that the values contain things that will be recognized as keys during the next iteration.

```
// Specify the transformation rules for this Lindenmayer system
Map<String, String> rules = new HashMap<String, String>() {{
  put("Cmajw", "Cmajw Fmajw");
  put("Fmajw", "Rw Bbmajw");
  put("Bbmajw", "Rw Fmajw");
  put("C5q", "C5q G5q E6q C6q");
  put("E6q", "G6q D6q F6i C6i D6q");
  put("G6i+D6i", "Rq Rq G6i+D6i G6i+D6i Rq");
}};
```

Next, we will set up the `ReplacementMapPreprocessor`, including setting the number of iterations to three and eliminating the need forangle brackets around the keys—and we will also provide the replacement map.

```
// Set up the ReplacementMapPreprocessor to iterate 3 times
// and not require brackets around replacements
ReplacementMapPreprocessor rmp =
ReplacementMapPreprocessor.getInstance();
rmp.setReplacementMap(rules);
rmp.setIterations(3);
rmp.setRequireAngleBrackets(false);
```

Now we just have to create the "axiom" – the initial string that will be transformed. We will put this into a Pattern.

```
// Create a Pattern that contains the L-System axiom
Pattern axiom = new Pattern("T120 " + "V0 I[Flute] Rq C5q "
  + "V1 I[Tubular_Bells] Rq Rq Rq G6i+D6i "
  + "V2 I[Piano] Cmajw E6q "
  + "V3 I[Warm] E6q G6i+D6i "
  + "V4 I[Voice] C5q E6q");
```

And now to play the result!

```
Player player = new Player();
player.play(axiom);
```

The music you are listening to now is a piece called "Kebu", which I originally created in 2007 and which I describe in detail on my website in an article describing a small portion of my work in algorithmic music.

## Music Quiz with Chords, MIDI Device Input, and a Custom ParserListener

I'm sure you've seen games where you need to play the note running across a staff before the note reaches the end of the staff (or, perhaps, is eaten by a monster, or set on fire, or some variation). In this custom JFugue game, you'll need to find all of the notes in a chord, and you'll need to play them on your MIDI keyboard! Quick, what are the notes in a Bb diminished seventh chord?

For the sake of simplicity, we'll forego a graphical user interface in this game, and we won't use a timer. You can have as much time as you want to find the notes in that Bb diminished seventh chord – as long as you don't press a wrong note!

The game consists of three main parts. First, we'll connect to a MIDI device. Next, we'll create the logic that generates random chords. Finally, we'll create a ParserListener that knows when the notes in the chord have been played.

First, we need a class. Let's call it ChordGame, and we'll give it a `main()` method. We'll need to either catch or throw a `MidiUnavailableException` when we get to the device part, so let's just do that now.

```
public class ChordGame {
    public static void main(String[] args) throws
MidiUnavailableException {
```

Great! Now, let's connect to your attached MIDI keyboard. We'll need to use onf of the two MusicTransmitter classes – either `MusicTransmitterToParserListener` or `MusicTransmitterToSequence`. When the player plays keys on the keyboard, we'll want to know about it and react to it, so that indicates that we should use the ParserListener variant.

We can use the `MidiDevicePrompt` that is part of JFugue's developer tools. If you were creating a real application that you wanted to share with others, you would do something more professional here. But for the sake of demonstration, this is perfectly acceptable.

Finally, we'll need to attach a ParserListener. Let's call it a `ChordGameParserListener`, and we'll worry about the logic a little bit later.

```
MusicTransmitterToParserListeners transmitter = new
     MusicTransmitterToParserListeners(
     MidiDevicePrompt.askForMidiDevice());
ChordGameParserListener listener = new ChordGameParserListener();
```

```
transmitter.addParserListener(listener);
```

Now we're connected to the listener and we're ready to start the game! We'll create chords using a root note plus a chord name. Fortunately, the Note and Chord classes make it easy to get the full range of allowable notes and chords with one line of code each!

```
String[] rootNames = Note.NOTE_NAMES_COMMON;
String[] chordNames = Chord.getChordNames();
```

Now for a little game logic. We'll keep track of the number of correct guesses, and we'll set the total number of guesses in a variable. Let's also create an instance of Java's Random so we can pick a root and chord at random.

```
int correctGuesses = 0;
int totalGuesses = 10;
Random rnd = new Random();
```

Time for the game! It's easy enough to create a new chord, as shown below. Let's also use the human readable string for the chord so we're not presenting the player with a Staccato string.

```
for (int i=0; i < totalGuesses; i++) {
     Chord chord = new Chord(
                   rootNames[rnd.nextInt(rootNames.length)] +
                   chordNames[rnd.nextInt(chordNames.length)]);
     System.out.println("Play " + chord.toHumanReadableString());
```

Now that we've show the player what chord we want them to play, we need to wait for them to play it! We'll write the ChordGameParserListener in a bit. In the meantime, we know that we want the player to play the chord, and whether they get it correct or incorrect, we want to keep track of that. At the end of `totalGuesses`, we'll print the player's results.

```
ChordGameParserListener.GameState gameState =
listener.waitForChord(chord);
if (gameState == ChordGameParserListener.GameState.CORRECT) {
    System.out.println("CORRECT! \n\n");
    correctGuesses++;
} else {
    System.out.println("Incorrect. \n\n");
}
}  // closing for loop

System.out.println("You guessed " + correctGuesses +
    " chords correctly out of " + totalGuesses + " challenges!");
transmitter.close();

}  // closing main()
```

Now it's time for that ParserListener! We're only interested in `onNoteParsed()`, so since we only care about one method, we should extend `ParserListenerAdapter` and override `onNoteParsed()`. We also know that we'll need to implement `waitForChord()`. Let's do that first, since it helps us think about what state we'll need to maintain in the listener.

`waitForChord()` will take a chord. It will need to know what notes are part of that chord (easy with Chord's API!). It will need to set up a variable so it knows whether each note in the chord has been played. And, it will need to know about game state, and only return when the chord is either played completely, or the player has made a mistake.

In other words:

```
public GameState waitForChord(Chord chord) {
    this.expectedNotes = chord.getNotes();
    this.notesMatched = new boolean[this.expectedNotes.length];
    this.gameState = GameState.GUESSING;

    // Waituntil the state changes to either CORRECT or WRONG
    while (gameState == GameState.GUESSING) {
        try {Thread.sleep(50);} catch(InterruptedException e) { }
    }

    return gameState;
}
```

That makes it clear that we'll need three variables as state for our class – and we'll need to initialize these in the constructor:

```
class ChordGameParserListener extends ParserListenerAdapter {
    private Note[] expectedNotes;
    private boolean[] notesMatched;
    private GameState gameState;

    public ChordGameParserListener() {
        expectedNotes = new Note[] { };
        notesMatched = new boolean[] { };
        gameState = GameState.GUESSING;
    }
```

Now all we have to do is react to when a note is played. If the player plays a note that is in the chord – regardless of octave (so, we'll use Note's `getPositionInOctave()` so we can be octave-agnostic) – we'll keep track in `notesMatched` that the corresponding note has been played. If the player makes a mistake, we'll change the game state to indicate that.

And once `notesMatched` is full of `TRUE` values, we'll know we have a successful chord!

```
@Override
public void onNoteParsed(Note note) {
    // If the note played matches one of the notes expected, yay!
    boolean matchFound = false;
    for (int i=0; i < expectedNotes.length; i++) {
        if (expectedNotes[i].getPositionInOctave() ==
            note.getPositionInOctave()) {
            notesMatched[i] = true;
        }
    }

    // If the player played an unexpected note, that's wrong!
    if (!matchFound) {
        this.gameState = GameState.WRONG;
    }

    // If we haven't yet matched all chord notes, exit
    for (int i=0; i < notesMatched.length; i++) {
        if (!notesMatched[i]) return;
    }

    // If we've gotten this far, all of the notes have matched!
    this.gameState = GameState.CORRECT;
}

public enum GameState { GUESSING, CORRECT, WRONG };

} // Close ChordGameParserListener
```

There you go – your very own chord game!

## Virtual Instrument with Realtime Player and Notes from Intervals

The `RealtimePlayer` lets you play notes, patterns, and other musical events in realtime (which makes it a pretty well-named class). This is different from the typical `Player`, which first sends your entire Staccato string to a StaccatoParser to be parsed, and *then* played in not-realtime. One of the best uses of the `RealtimePlayer` is to create something that plays music in response to user interaction. Let's create something that a user can interact with!

The typical example here might be to draw several piano keys, or maybe some guitar strings, and let a person tickle the ivory or pluck the strings. Or, we could take advantage of the medium and create something colorful, different, and interesting to explore. Introducing... a pile of randomly colored shapes!

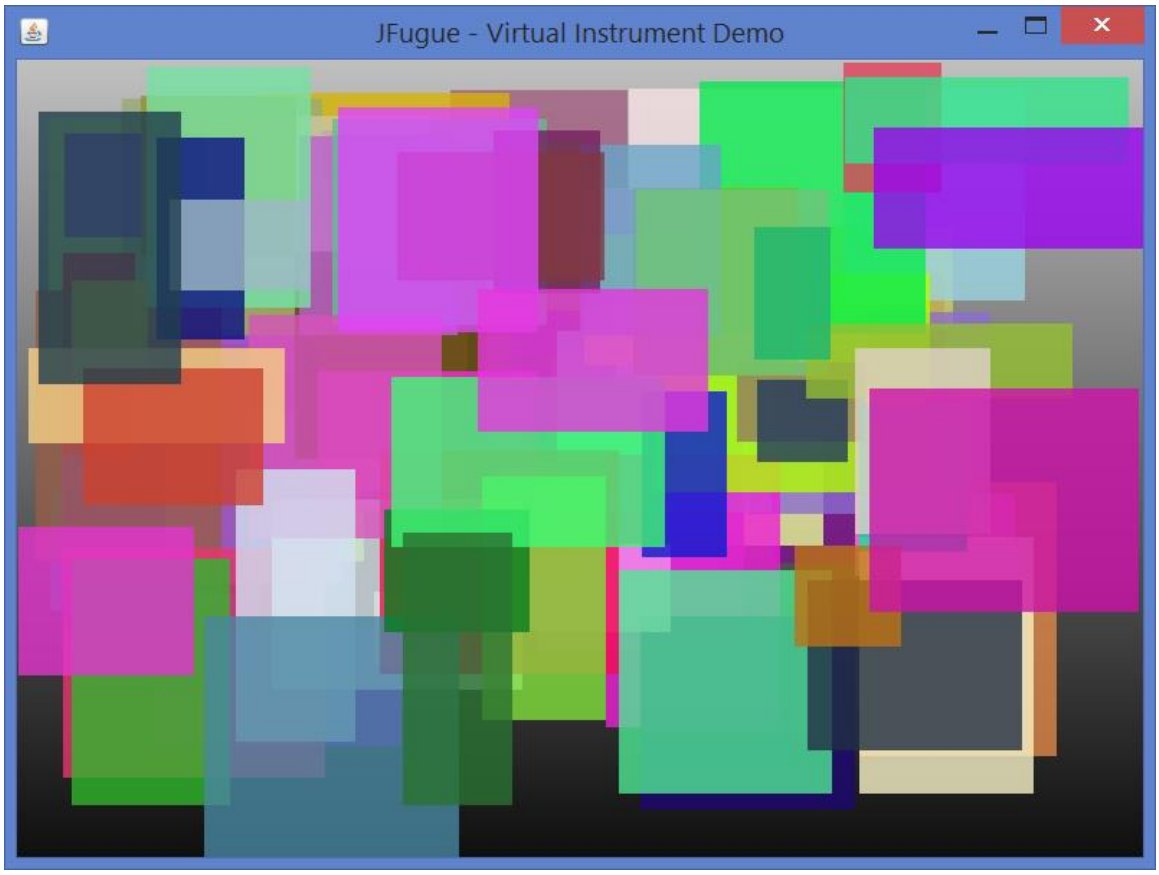



**Figure 15. The Virtual Instrument**

100 rectangles, to be exact. Each rectangle contains a note, and you can use the mouse to click on a rectangle and hear its note. If you click on a spot that has overlapping rectangles, you'll hear all of the notes for each of the overlapping rectangles. In other words, where the rectangles overlap, you'll hear a chord. Here are some of the first bits of code of this novel virtual instrument.

```
public class VirtualInstrument extends JPanel implements
MouseListener {
    private List<InstrumentZone> instrumentZones;
    private RealtimePlayer player;
    private Note[] notes;

    public VirtualInstrument() throws MidiUnavailableException {
        super();
        this.instrumentZones = new ArrayList<InstrumentZone>();
        this.addMouseListener(this);
        this.player = new RealtimePlayer();
```

```
        this.player.play("I[Crystal]");
        this.notes = new Chord(new Note("C"), new Intervals("1 2
3 5 6")).getNotes();
        createInstrumentZones();
    }
```

We're just getting started, but let's take a look at a few things here. First, as promised, we've using the `RealtimePlayer`. Next, we're going to use the `"I[Crystal]"` instrument, which is an interesting instrument that you may not have heard before. For this application, it's nice because the sound of the instrument fades away after a while, as opposed to something like a Flute that keeps playing forever. That's important because of how we'll let the user interact with this instrument: When they press the mouse button on the rectangles, they will start playing their notes, and when the user releases the mouse button over the same rectangle, they will stop playing their notes… but if the user moves to a different area, the notes won't get a chance to stop on their own. Choosing an instrument like the Crystal lets those notes fade away without being explicitly turned off.

We will also pull notes from a `"1 2 3 5 6"` interval. That's the Major Pentatonic Scale. If you like, you can instead try a Chinese Scale like `"1 3 #4 5 7"`, or any other scale you might find interesting. We'll use this scale to create a chord – which means the chord will contain each note in the scale – and we'll pull the notes from the chord.

Next, we'll create a bunch of "Instrument Zones" – in other words, rectangles. But not just rectangles! Rectangles that have color and play music. That includes creating the note that each of the zones will play. We'll build that note using one of the tones from the chord and a randomly selected octave. And, we'll override `VirtualInstrument`'s `paint()` method to delegate to the rectangles so they can paint themselves.

```
    private void createInstrumentZones() {
        Random rnd = new Random();
        for (int i=0; i < VirtualInstrument.NUM_ZONES; i++) {
            int x = rnd.nextInt(VirtualInstrument.WIDTH -
VirtualInstrument.MAX_RECT_WIDTH);
            int y = rnd.nextInt(VirtualInstrument.HEIGHT -
VirtualInstrument.MAX_RECT_HEIGHT);
            int w = VirtualInstrument.MIN_RECT_WIDTH +
rnd.nextInt(VirtualInstrument.MAX_RECT_WIDTH -
VirtualInstrument.MIN_RECT_WIDTH);
            int h = VirtualInstrument.MIN_RECT_HEIGHT +
rnd.nextInt(VirtualInstrument.MAX_RECT_HEIGHT -
VirtualInstrument.MIN_RECT_HEIGHT);
```

```
            Color color = new Color(rnd.nextFloat(),
rnd.nextFloat(), rnd.nextFloat(), 0.8f);
            Note note = new
Note(notes[rnd.nextInt(notes.length)].toString() + (2 +
(rnd.nextInt(4))));
            instrumentZones.add(new InstrumentZone(x, y, w, h,
color, player, note));
        }
    }

    @Override
    public void paint(Graphics g) {
        super.paint(g);

        Graphics2D g2 = (Graphics2D)g;
        g2.setPaint(new GradientPaint(new
Point2D.Double(VirtualInstrument.WIDTH / 2.0, 0.0),
Color.LIGHT_GRAY, new Point2D.Double(VirtualInstrument.WIDTH /
2.0, VirtualInstrument.HEIGHT), Color.BLACK));
        g2.fillRect(0, 0, VirtualInstrument.WIDTH,
VirtualInstrument.HEIGHT);
        for (InstrumentZone izone : instrumentZones) {
            izone.paint(g2);
        }
    }
```

We'll also delegate to the rectangles when the user presses or releases the mouse button. For the other `MouseListener` events that we don't need, we'll create empty implementations. (Actually, you could do something interesting, like have a rectangle light up when the mouse enters it, but I'll leave that as an exercise for the reader.)

```
    @Override
    public void mousePressed(MouseEvent event) {
        for (InstrumentZone izone : instrumentZones) {
            izone.mousePressed(event.getPoint());
        }
    }

    @Override
    public void mouseReleased(MouseEvent event) {
        for (InstrumentZone izone : instrumentZones) {
            izone.mouseReleased(event.getPoint());
        }
    }

    @Override public void mouseClicked(MouseEvent e) { }

    @Override public void mouseEntered(MouseEvent e) { }

    @Override public void mouseExited(MouseEvent e) { }
```

Now for a `main()` method, which will create a JFrame and display the instrument. And a couple of static integers – placing these values in one spot here lets you change the appearance of the virtual instrument in one easy-to-find place.

```
    public static void main(String[] args) throws
MidiUnavailableException {
        JFrame frame = new JFrame("JFugue - Virtual Instrument
Demo");
        frame.setSize(VirtualInstrument.WIDTH,
VirtualInstrument.HEIGHT);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(new VirtualInstrument(),
BorderLayout.CENTER);
        frame.setVisible(true);
    }

    private static int WIDTH = 800;
    private static int HEIGHT = 600;
    private static int MIN_RECT_WIDTH = 50;
    private static int MAX_RECT_WIDTH = 200;
    private static int MIN_RECT_HEIGHT = 50;
    private static int MAX_RECT_HEIGHT = 200;
}
```

Let's take a look at one of these Instrument Zones. It's quite simple: the constructor takes, among other things, an instance of the `RealtimePlayer` and the `Note` that will be played when the user presses the mouse button over this zone.

```
class InstrumentZone {
    public int x, y, width, height;
    public Paint paint;
    public RealtimePlayer realtimePlayer;
    public Note note;

    public InstrumentZone(int x, int y, int width, int height,
Paint paint, RealtimePlayer player, Note note) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
        this.paint = paint;
        this.realtimePlayer = player;
        this.note = note;
    }

    public void paint(Graphics2D g2) {
        g2.setPaint(this.paint);
        g2.fillRect(x, y, width, height);
```

```
    }

    public void mousePressed(Point2D point) {
        if (((point.getX() >= x) && (point.getX() <= x + width))
&& ((point.getY() >= y) && (point.getY() <= y + height))) {
            realtimePlayer.startNote(note);
        }
    }

    public void mouseReleased(Point2D point) {
        if (((point.getX() >= x) && (point.getX() <= x + width))
&& ((point.getY() >= y) && (point.getY() <= y + height))) {
            realtimePlayer.stopNote(note);
        }
    }
}
```

That's it! Run this instrument and have fun!

# 7.2

## *Tests, Demos, and Examples in the Source Code Distribution*

So far, all of the material in this book has covered the main JFugue code, which exists in the src/main/java. This is one of several high-level folders, and in this chapter, you will learn more about what to expect (and how to use) the source code in the other folders.

### src/main

The code in the `java` folder, and the `staccato.properties` file in the `resources` folder, generate the class files that are put into the JFugue jar file. All of the code that makes JFugue work is included in these folders.

### src/test

This folder is full of JUnit test cases. In general, there is a test case for each class in the main JFugue code. Each test case can be run individually, or the whole suite of tests can be run; in fact, successfully passing the tests is a necessary condition for JFugue's build script to generate a jar for use as a beta version or for distribution. There is one audible test in these test cases which is a sanity check to ensure that basic musical output is correct.

## src/manualtest

The `manualtest` folder contains programs that serve as test cases but they are not JUnit tests cases. They are runnable programs that are meant to test musical output from the JFugue classes they test. Determining whether these tests run successfully is a subjective measure – they need to be listened to. When these tests are run, they use `ManualTestPrint.expectedResult()` to output a description of the expected result of the test to the console. This is how you know what to listen for when running one of these tests. If you ever create a new manual test, please be sure to use `ManualTestPrint.expectedResult()` to describe the expected outcome of your test.

## src/demo

The demo folder contains demonstration use cases for most of the JFugue classes. If there is a particular class that you are interested in – say, the `Rhythm` or `MusicReceiver` class – you will find a corresponding `RhythmDemo` or `MusicReceiverDemo` class. The source code of the demonstrations is rather well documented, and some of the demos display output in the console to let you know what you are hearing as the demo runs.

## src/examples

The examples folder is where you will find the source code that appears in this book and on the JFugue.org website.

# 7.3

# *Building and Testing JFugue*

So far, all of the material in this book has covered the use of JFugue as a library. But what if you want to make your own changes to JFugue and build a new jar yourself?

## Building JFugue

JFugue has an `build.xml` file for use with Apache Ant. The build file contains the targets described in Table 14.

| Build Target | Description | Depends on |
|---|---|---|
| `init` | Creates a timestamp that is used by some of the other targets | *(none)* |
| `compile` | Runs javac on src/main, creating target/main | `init` |
| `betajar` | Creates a jar with the classes in target/main and gives the jar file a filename with the timestamp | `compile` |
| `compile-test` | Runs javac on src/test, creating target/test | `compile` |

| | | |
|---|---|---|
| `test` | Runs the tests | *(none)* |
| `clean` | Deletes the files in target/src and target/main | *(none)* |
| `build` | Runs the clean, compile, and test targets | `clean, compile, test` |
| `doc` | Generates the JavaDoc for the code in src/main | *(none)* |
| `zip-doc` | Zips the JavaDoc | `doc` |
| `zip-source` | Zips the sources | *(none)* |
| `betadist` | Creates a src zip, doc zip, and jar file, and gives each of those three files a filename with a timestamp | `build, doc` |
| `dist` | Creates a jar file and relies on zip-source and zip-doc to create their zips | `build, zip-source, zip-doc` |

**Table 14. Build targets in build.xml**

## Testing JFugue

JFugue comes with an excellent set of unit tests. If you make changes to the code, you should always check that all of the unit tests pass successfully.

Also, be sure to add new unit tests for any new code you create. And, feel free to contribute unit tests for those areas of JFugue that do not currently have much test coverage.

The `JFugueTestHelper` class comes with a method that makes it easy to check the results of passing a string to the StaccatoParser and checking the result. This is the `compare()` method, which takes three parameters: The string to parse, a string representing the expected event type (e.g., the event fired to a ParserListener), and an object representing the expected object. There are a bunch of `compare()` methods in `JFugueTestHelper` that all have different object parameters, and at the end of the class file you will also find a set of static final strings for the event names. The `compare()` method compares the *second-to-last* event that the ParserListener receives. The last event will always be a `afterParsingFinished()` event, so we can skip that one. But if you are testing a Staccato string that will send multiple events, realize that you can only check against the last non-finished event.

Here are some examples that use the `compare()` method within JUnit's `assertTrue()` method.

```
assertTrue(compare("C", NOTE_PARSED, new Note(60)));

assertTrue(compare("C0", NOTE_PARSED, new
Note(0).setOctaveExplicitlySet(true)));

assertTrue(compare("C-w-", NOTE_PARSED, new
Note(60).setDuration(1.0d).setStartOfTie(true).setEndOfTie(true))
);

assertTrue(compare("#(three word marker)", MARKER_PARSED, "three
word marker"));

assertTrue(compare(":PITCHBEND(16008)", PITCH_WHEEL_PARSED,
(byte)8, (byte)125));

assertTrue(compare("@200", TRACK_BEAT_TIME_REQUESTED, 200.0D));
```

# 7.4

## *Short JFugue Programs*

One of the exciting benefits of JFugue's easy-to-use API is that it provides a fun and accessible introduction to programming for young people who are learning how to write their first programs. In fact, JFugue has already been used in a few high schools around the world, and some of the classes have even contributed code back to the JFugue project. University students from around the world are also using JFugue in their projects.

### "Hello, World" in JFugue

Create music with only a few lines of code!

```
import org.jfugue.player.Player;

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

## Playing multiple voices, multiple instruments, rests, chords, and durations

This example uses the Staccato 'V' command for specifying voices, 'I' for specifying instruments (text within brackets is looked up in a dictionary and maps to MIDI instrument numbers), '|' (pipe) for indicating measures (optional), durations including 'q' for quarter duration, 'qqq' for three quarter notes (multiple durations can be listed together), and 'h' for half, 'w' for whole, and '.' for a dotted duration; 'R' for rest, and the chords G-Major and C-Major. Whitespace is not significant and can be used for visually pleasing or helpful spacing.

```
import org.jfugue.player.Player;

public class HelloWorld2 {
  public static void main(String[] args) {
    Player player = new Player();
    player.play("V0 I[Piano] Eq Ch. | Eq Ch. | Dq Eq Dq Cq   V1
I[Flute] Rw | Rw | GmajQQQ CmajQ");
  }
}
```

## Introduction to Patterns

Patterns are one of the fundamental units of music in JFugue. They can be manipulated in musically interesting ways.

```
import org.jfugue.pattern.Pattern;
import org.jfugue.player.Player;

public class IntroToPatterns {
  public static void main(String[] args) {
    Pattern p1 = new Pattern("V0 I[Piano] Eq Ch. | Eq Ch. | Dq Eq
Dq Cq");
    Pattern p2 = new Pattern("V1 I[Flute] Rw     | Rw     |
GmajQQQ  CmajQ");
    Player player = new Player();
    player.play(p1, p2);
  }
}
```

## Introduction to Patterns, Part 2

Voice and instruments for a pattern can also be set through the API. In JFugue, methods that would normally return 'void' instead return the object itself, which allows you do chain commands together, as seen in this example.

```
import org.jfugue.pattern.Pattern;
import org.jfugue.player.Player;
```

```
public class IntroToPatterns2 {
  public static void main(String[] args) {
    Pattern p1 = new Pattern("Eq Ch. | Eq Ch. | Dq Eq Dq Cq")
        .setVoice(0).setInstrument("Piano");
    Pattern p2 = new Pattern("Rw     | Rw     | GmajQQQ  CmajQ")
        .setVoice(1).setInstrument("Flute");
    Player player = new Player();
    player.play(p1, p2);
  }
}
```

## Introduction to Chord Progressions

It's easy to create a Chord Progression in JFugue. You can then play it, or you can see the notes that comprise the any of the chords in the progression.

```
import org.jfugue.player.Player;
import org.jfugue.theory.Chord;
import org.jfugue.theory.ChordProgression;
import org.jfugue.theory.Note;

public class IntroToChordProgressions1 {
  public static void main(String[] args) {
    ChordProgression cp = new ChordProgression("I IV V");
    Chord[] chords = cp.setKey("C").getChords();
    for (Chord chord : chords) {
      System.out.print("Chord "+chord+" has these notes: ");
      Note[] notes = chord.getNotes();
      for (Note note : notes) {
        System.out.print(note+" ");
      }
      System.out.println();
    }

    Player player = new Player();
    player.play(cp);
  }
}
```

## Advanced Chord Progressions

You can do some pretty cool things with chord progressions. The methods below use `$` to indicate an index into either the chord progression (in which case, the index points to the nth chord), or a specific chord (in which case the index points to the nth note of the chord). Underscore means "the whole thing". If you change the indexes, make sure you don't introduce an `ArrayOutOfBoundsException` (for

example, a major chord has only three notes, so trying to get the 4th index, `$3` (remember that this is zero-based), would cause an error).

```
import org.jfugue.player.Player;
import org.jfugue.theory.ChordProgression;

public class AdvancedChordProgressions {
  public static void main(String[] args) {
    ChordProgression cp = new ChordProgression("I IV V");

    Player player = new Player();
    player.play(cp.allChordsAs("$0 $0 $0 $0 $1 $1 $2
$0").eachChordAs("V0 $0s $1s $2s Rs V1 $_q"));
  }
}
```

## Twelve-Bar Blues in Two Lines of Code

Twelve-bar blues uses a I-IV-V chord progression. But really, it's the Major 7ths that you'd like to hear... and if you want to play each chord in arpeggio, you need a 6th in there as well. But creating a I7%6-IV7%6-V7%6 chord progression is messy. So, this code creates a I-IV-V progression, then distributes a 7%6 across each chord, then creates the twelve bars, and then each chord is played as an arpeggio with note dynamics (note on velocity - how hard you hit the note). Finally, the pattern is played with an `Acoustic_Bass` instrument at 110 BPM. With all of the method chaining, that is *kinda* done in one line of code.

```
import java.io.IOException;

import org.jfugue.pattern.Pattern;
import org.jfugue.player.Player;
import org.jfugue.theory.ChordProgression;

public class TwelveBarBlues {
    public static void main(String[] args) throws IOException {
        Pattern pattern = new ChordProgression("I IV V")
                .distribute("7%6")
                .allChordsAs("$0 $0 $0 $0 $1 $1 $0 $0 $2 $1 $0
$0")
                .eachChordAs("$0ia100 $1ia80 $2ia80 $3ia80
$4ia100 $3ia80 $2ia80 $1ia80")
                .getPattern()
                .setInstrument("Acoustic_Bass")
                .setTempo(100);
        new Player().play(pattern);
    }
}
```

## Introduction to Rhythms

One of my favorite parts of the JFugue API is the ability to create rhythms in a fun and easily understandable way. The letters are mapped to percussive instrument sounds, like "Acoustic Snare" and "Closed Hi Hat". JFugue comes with a default "rhythm set", which is a `Map<Character, String>` with entries like this: `put('O', "[BASS_DRUM]i")`.

```
import org.jfugue.pattern.Pattern;
import org.jfugue.player.Player;
import org.jfugue.rhythm.Rhythm;
import org.jfugue.theory.ChordProgression;

public class IntroductionToRhythms2 {
      public static void main(String[] args) {
            // Create a rhythm
            Rhythm rhythm = new Rhythm();
          rhythm.addLayer("O..oO...O..oOO..");
          rhythm.addLayer("..S...S...S...S.");
          rhythm.addLayer("`````````````````");
          rhythm.addLayer("................+");

            // Create a chord progression, give it a key (with
duration), and make sure it always plays in Voice 1
          Pattern chords = new ChordProgression("I IV V
I").setKey("AbMINww").getPattern().setVoice(1);

          // Combine the chords and the rhythm
            Pattern combinedPattern = new Pattern(chords,
rhythm.getPattern().repeat(4));

            // Play the *combined* pattern twice (that will be 8
total occurrences the rhythm)
            Player player = new Player();
          player.play(combinedPattern.repeat(2));
      }
}
```

## Advanced Rhythms

Through the Rhythm API, you can specify a variety of alternate layers that occur once or recur regularly. You can even create your own "RhythmAltLayerProvider" if you'd like to create a new behavior that does not already exist in the Rhythm API.

```
import org.jfugue.player.Player;
import org.jfugue.rhythm.Rhythm;
```

```
public class AdvancedRhythms {
    public static void main(String[] args) {
        Rhythm rhythm = new Rhythm()
          .addLayer("O..oO...O..oOO..")
          .addLayer("..S...S...S...S.")
          .addLayer("````````````````")
          .addLayer("...............+")
          .addOneTimeAltLayer(3, 3, "...+...+...+...+")
          .setLength(4);
        new Player().play(rhythm.getPattern().repeat(2));
    }
}
```

## All That, in One Line of Code?

Try this. The main line of code even fits within the 140-character limit of a tweet.

```
import org.jfugue.player.Player;
import org.jfugue.rhythm.Rhythm;
import org.jfugue.theory.ChordProgression;

public class TryThis {
    public static void main(String[] args) {
        new Player().play(new ChordProgression("I IV vi
V").eachChordAs("$_i $_i Ri $_i"), new
Rhythm().addLayer("..X...X...X...XO"));
    }
}
```

## See the Contents of a MIDI File in Human-Readable and Machine-Parseable Staccato Format

Want to see the music in your MIDI file? Of course, you could load it in a sheet music tool. Here's how you can load it with JFugue. You'll get a Pattern of your music, which you can then pick apart in interesting ways (for example, count how many "C" notes there are... that's coming up in a few examples)

```
import java.io.File;
import java.io.IOException;

import javax.sound.midi.InvalidMidiDataException;

import org.jfugue.midi.MidiFileManager;
import org.jfugue.pattern.Pattern;

public class SeeMidi {
    public static void main(String[] args) throws IOException,
InvalidMidiDataException {
```

```
        Pattern pattern = MidiFileManager.loadPatternFromMidi(new
File("filename.mid"));
        System.out.println(pattern);
    }
}
```

## Connecting Any Parser to Any ParserListener

You can use JFugue to convert between music formats. Most commonly, JFugue is used to turn Staccato music into MIDI sound. Alternatively, you can play with the MIDI, MusicXML, and LilyPond parsers and listeners. Or, you can easily create your own parser or listener, and it will instantly interoperate with the other existing formats. (And if you convert to Staccato, you can then play the Staccato music... and edit it!)

```
import java.io.File;
import java.io.IOException;

import javax.sound.midi.InvalidMidiDataException;
import javax.sound.midi.MidiSystem;

import org.jfugue.midi.MidiParser;
import org.jfugue.pattern.Pattern;
import org.jfugue.player.Player;
import org.staccato.StaccatoParserListener;

public class ParserDemo {
    public static void main(String[] args) throws
InvalidMidiDataException, IOException {
        MidiParser parser = new MidiParser();
        StaccatoParserListener listener = new
StaccatoParserListener();
        parser.addParserListener(listener);
        parser.parse(MidiSystem.getSequence(new
File("filename.mid")));
        Pattern staccatoPattern = listener.getPattern();
        System.out.println(staccatoPattern);

        Player player = new Player();
        player.play(staccatoPattern);
    }
}
```

## Create a Listener to Find Out About Music

You can create a ParserListener to listen for any musical event that any parser is parsing. Here, we'll create a simple tool that counts how many "C" notes (of any octave) are played in any song.

```
import java.io.File;
import java.io.IOException;
```

```
import javax.sound.midi.InvalidMidiDataException;
import javax.sound.midi.MidiSystem;

import org.jfugue.midi.MidiParser;
import org.jfugue.parser.ParserListenerAdapter;
import org.jfugue.theory.Note;

public class MyParserListenerDemo {

    public static void main(String[] args) throws
InvalidMidiDataException, IOException {
        MidiParser parser = new MidiParser();
        MyParserListener listener = new MyParserListener();
        parser.addParserListener(listener);
        parser.parse(MidiSystem.getSequence(new
File("filename.mid")));
        System.out.println("There are "+listener.counter+" 'C'
notes in this music.");
    }
}

class MyParserListener extends ParserListenerAdapter {
    public int counter;

    @Override
    public void onNoteParsed(Note note) {
        if (note.getPositionInOctave() == 0) {
            counter++;
        }
    }
}
```

## Play Music in Realtime

Create interactive musical programs using the `RealtimePlayer`.

```
import java.util.Random;
import java.util.Scanner;

import javax.sound.midi.MidiUnavailableException;

import org.jfugue.pattern.Pattern;
import org.jfugue.realtime.RealtimePlayer;
import org.jfugue.theory.Note;

public class RealtimeExample {
    public static void main(String[] args) throws
MidiUnavailableException {
        RealtimePlayer player = new RealtimePlayer();
        Scanner scanner = new Scanner(System.in);
        Random random = new Random();
```

```
        boolean quit = false;
        while (quit == false) {
            System.out.print("Enter a '+C' to start a note, '-C'
to stop a note, 'i' for a random instrument, 'p' for a pattern,
or 'q' to quit: ");
            String entry = scanner.next();
            if (entry.startsWith("+")) {
               player.startNote(new Note(entry.substring(1)));
            }
            else if (entry.startsWith("-")) {
                player.stopNote(new Note(entry.substring(1)));
            }
            else if (entry.equalsIgnoreCase("i")) {
                player.changeInstrument(random.nextInt(128));
            }
            else if (entry.equalsIgnoreCase("p")) {

player.play(PATTERNS[random.nextInt(PATTERNS.length)]);
            }
            else if (entry.equalsIgnoreCase("q")) {
                quit = true;
            }
        }
        scanner.close();
        player.close();
    }

    private static Pattern[] PATTERNS = new Pattern[] {
            new Pattern("Cmajq Dmajq Emajq"),
            new Pattern("V0 Ei Gi Di Ci  V1 Gi Ci Fi Ei"),
            new Pattern("V0 Cmajq V1 Gmajq")
    };
}
```

## Anticipate Musical Events Before They Occur

You might imagine creating new types of ParserListeners, like an `AnimationParserListener`, that depend on knowing about the musical events before they happen. For example, perhaps your animation is of a robot playing a drum or strumming a guitar. Before the note makes a sound, the animation needs to get its virtual hands in the right place, so you might want a notice 500ms earlier that a musical event is about to happen. To bend time with JFugue, use a combination of the `TemporalPLP` class and `Player.delayPlay()`. `delayPlay()` creates a new thread that first waits the specified amount of time before playing. If you do this, make sure to call `delayPlay()` before `plp.parse()`.

```
import org.jfugue.devtools.DiagnosticParserListener;
import org.jfugue.player.Player;
import org.jfugue.temporal.TemporalPLP;
```

```
import org.staccato.StaccatoParser;

public class TemporalExample {
    private static final String MUSIC = "C D E F G A B";  // Feel
free to put your own music here to experiment!
    private static final long TEMPORAL_DELAY = 500;       // Feel
free to put your own delay here to experiment!

    public static void main(String[] args) {
        // Part 1. Parse the original music
        StaccatoParser parser = new StaccatoParser();
        TemporalPLP plp = new TemporalPLP();
        parser.addParserListener(plp);
        parser.parse(MUSIC);

        // Part 2. Send the events from Part 1, and play the
original music with a delay
        DiagnosticParserListener dpl = new
DiagnosticParserListener();
        plp.addParserListener(dpl);
        new Player().delayPlay(TEMPORAL_DELAY, MUSIC);
        plp.parse();
    }
}
```

## Use "Replacement Maps" to Create Carnatic Music

JFugue's ReplacementMap capability lets you use your own symbols in a music string. JFugue comes with a CarnaticReplacementMap that maps Carnatic notes to microtone frequencies.

```
import org.jfugue.player.Player;
import org.staccato.maps.CarnaticReplacementMap;

public class CarnaticReplacementMapDemo {
     public static void main(String[] args) {

     ReplacementMapPreprocessor.getInstance().setReplacementMap(
new CarnaticReplacementMap());

          Player player = new Player();
          player.play("<S> <R1> <R2> <R3> <R4>");
     }
}
```

## Use "Replacement Maps" to Play Solfege

JFugue comes with a SolfegeReplacementMap, which means you can program music using "Do Re Me Fa So La Ti Do." The ReplacementMapParser converts those solfege tones to C D E F G A B. Using Replacement Maps, which are simply Map<String, String>, you

can create any kind of music in a pattern that will be converted to musical notes (or whatever you put in the values of your Map).

```
import org.jfugue.pattern.Pattern;
import org.jfugue.player.Player;
import org.staccato.ReplacementMapPreprocessor;

public class SolfegeReplacementMapDemo {
     public static void main(String[] args) {
           Player player = new Player();
           ReplacementMapPreprocessor rmp =
ReplacementMapPreprocessor.getInstance();
           rmp.setReplacementMap(new
SolfegeReplacementMap()).setRequireAngleBrackets(false);
           Pattern pattern = new Pattern("do re mi fa so la ti
do");

     System.out.println(rmp.preprocess(pattern.toString().toUppe
rCase(), null));

           rmp.setRequireAngleBrackets(true);
           player.play(new Pattern("<Do>q <Re>q <Mi>h | <Mi>q
<Fa>q <So>h | <So>q <Fa>q <Mi>h | <Mi>q <Re>q <Do>h"));

     }
}
```

## Use "Replacement Maps" to Generate Fractal Music

A Lindenmayer system is a string rewriting system that can be used to create fractal shapes. You can use JFugue's ReplacementMap capability to create music based on string rewrite rules! (File this one under, "I didn't *intentionally* create a fractal music tool, it just kinda happened. Oops.")

```
import java.util.HashMap;
import java.util.Map;

import org.jfugue.pattern.Pattern;
import org.jfugue.player.Player;
import org.staccato.ReplacementMapPreprocessor;

public class LSystemMusic {
     public static void main(String[] args) {

           // Specify the transformation rules for this
Lindenmayer system
           Map<String, String> rules = new HashMap<String,
String>() {{
              put("Cmajw", "Cmajw Fmajw");
              put("Fmajw", "Rw Bbmajw");
```

```
            put("Bbmajw", "Rw Fmajw");
            put("C5q", "C5q G5q E6q C6q");
            put("E6q", "G6q D6q F6i C6i D6q");
            put("G6i+D6i", "Rq Rq G6i+D6i G6i+D6i Rq");
            put("axiom", "axiom V0 I[Flute] Rq C5q V1
I[Tubular_Bells] Rq Rq Rq G6i+D6i V2 I[Piano] Cmajw E6q V3
I[Warm] E6q G6i+D6i V4 I[Voice] C5q E6q");
        }};

        // Set up the ReplacementMapPreprocessor to iterate 3
times
        // and not require brackets around replacements
        ReplacementMapPreprocessor rmp =
ReplacementMapPreprocessor.getInstance();
        rmp.setReplacementMap(rules);
        rmp.setIterations(3);
        rmp.setRequireAngleBrackets(false);

        // Create a Pattern that contains the L-System axiom
        Pattern axiom = new Pattern("T120 " + "V0 I[Flute] Rq
C5q "
                + "V1 I[Tubular_Bells] Rq Rq Rq G6i+D6i "
                + "V2 I[Piano] Cmajw E6q "
                + "V3 I[Warm] E6q G6i+D6i "
                + "V4 I[Voice] C5q E6q");

        Player player = new Player();
        System.out.println(rmp.preprocess(axiom.toString(),
null));
        player.play(rmp.preprocess(axiom.toString(), null));
    }
}
```

# Conclusion

Thank you again for using JFugue. I hope this book has clearly communicated all of the wonderful things that JFugue can do. While the benefits of JFugue are first evidenced by that simple `player.play("C")` call, there's really a lot more power to be discovered – and all of it is designed to be as intuitive and easy-to-use as possible. I firmly believe that if software isn't easy to use, it won't *be* used. If JFugue were not easy to use, I would have wasted my time developing the tool, and you wouldn't be able to create and explore music in new and creative ways. Fortunately, neither is true, and the world is a slightly happier place.

If you have enjoyed using JFugue, please share your joy with friends and acquaintances. Post to blogs, mention JFugue in your books and presentations, and so on. Help spread the word about this easy-to-use API that resurrects some of the joy of programming.

If you have found this book to be enlightening and beneficial, please communicate that, as well. The more encouragement that I get from users of my software, the more likely I will be to develop additional tools that are just as delightful and easy to use.

Have fun, and stay creative! -David Koelle

# Appendix

## *Setting Up JFugue*

JFugue makes programming music easy. This chapter explains how to set up and get started with JFugue.

### Downloading JFugue

The easiest way to start being productive with JFugue is to download the `jfugue.jar` file and include it as a library in your own programs. You can always obtain the latest jfugue.jar file from http://www.JFugue.org.

> **Personal Tip**
>
> When I download third-party libraries, I place them into a folder called "C:\Java Libraries", where I extract the library's compressed files, including source files and JavaDoc. When I need to use the jar file in a specific project, I make a copy of the jar file and paste it into my project's lib directory.

Once you have the `jfugue.jar` file, you are ready to include it in your program. Make sure that `jfugue.jar` appears in your classpath, or if you are using an integrated development environment (IDE), include `jfugue.jar` as a library.

Now you are ready to start writing programs with JFugue!

> **Java Version Compatibility**
>
> As of this writing, JFugue is compiled with Java Runtime Environment (JRE) version 1.8.0

## Running a Test Program

To be sure you are able to use JFugue after you download it, compile and run the following program, which should be saved as `HelloWorld.java`.

```
import org.jfugue.player.Player;

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

To compile and run this program from a command prompt, follow the following steps. If you are using an IDE, you may jump ahead to the next section.

Step 1. To *compile* this program, enter this command at the command prompt, replacing `%JFUGUE_DIR%` with the directory into which you have placed `jfugue.jar`:

```
javac -classpath %JFUGUE_DIR%\jfugue.jar HelloWorld.java
```

This will compile MyMusicApp.java and generate a .class file.

Step 2. To *run* the .class file, enter this line:

```
java -cp %JFUGUE_DIR%\jfugue.jar;. HelloWorld
```

Be sure to copy this line exactly. The semicolon and period indicate where Java will find the HelloWorld class – in the current (i.e., ".") directory.

> **Special Note for Mac Users**
>
> If you're using a Mac, replace the semicolon (;) with a colon (:).

## Using JFugue from an Integrated Development Environment

If you're using an Integrated Development Environment (IDE), like Eclipse or NetBeans, you'll need to include the JFugue jar file into your project. If you're using Eclipse (http://www.eclipse.org), go *Project* >

*Properties*, select *Java Build Path*, select the *Libraries* tab, and click the *"Add JARs..."* or *"Add External JARs..."* button. Find `jfugue.jar` and add it to your project.

> **Personal Tip**
>
> For each of my projects, I create a lib directory, where I place third-party jar files.

To run the test program from Eclipse, right-click on the test program's filename and select *Run As...> Java Application*.

## Using MIDI Soundbanks

JFugue relies on Java's MIDI capabilities to produce music. Java MIDI uses the Java Sound engine, which in turn uses a soundbank to generate sounds using the synthesizer. A soundbank is a collection of audio samples for each instrument that are played by the synthesizer. A variety of soundbanks provided by Sun Microsystems are available for free download; some of these may provide richer sounds than the default soundbank that is packaged with the Java Runtime Environment (JRE).

In addition, there are third-party MIDI soundbanks that have incredibly rich sound. Many of these are available for purchase only. Try doing an online search for "midi sound bank" to see some examples.

### Downloading Soundbanks

Soundbanks provided by Sun Microsystems can be downloaded from http://www.oracle.com/technetwork/java/soundbanks-135798.html. These same three soundbanks have been available for download for at least ten years. My recommendation: Don't bother with anything other than Deluxe.

**Minimal** (0.35 MB)

This soundbank is packaged by default with Java SDK Standard Edition versions 1.2.2 and higher. It is the smallest soundbank available, and its sound samples are of slightly less quality than those found in the midsize soundbank.

**Midsize** (1.09 MB)

This soundbank shipped with Java2 versions 1.2 and 1.2.1.

**Deluxe** (4.92 MB)

This soundbank contains higher-quality sound samples.

### Installing the Java Media Soundbanks

Installing a soundbank is as simple as moving the file you've downloaded to the correct directory.

First, download and unzip the soundbank you are interested in. You will see a file with a ".gm" extension.

On Windows computers, move this file to

`C:\Program Files\java\<jre version>\lib\audio`. If there is no `audio` directory, create it. In addition, if you are using a Java SDK that you've downloaded, also copy the soundbank file to `<jdk-install-dir>\jre\lib\audio`.

On Linux or Solaris machines, move the soundbank file to `<install-dir>/jre/lib/audio`. If the `audio` directory does not exist, create it.

Java will automatically use the highest-quality soundbank available, so if there is an existing soundbank file in the `audio` directory, you don't have to delete or rename it.

After you have moved your soundbank to the correct directory, be sure to exit any running Java programs. When you start them up again, they will use the new soundbanks.

# *Image Credits*

Figures that include musical notation were created by the author using NoteWorthy Composer from NoteWorthy Software, Inc.

The images of sheet music that decorate each of the major parts of the book are all in the public domain.

**Cover:** See description on Page 4.

**Before Table of Contents:** Score of "Belle, bonne, sage" by Baude Cordier. From *The Chantilly Manuscript*. The piece uses the red color to show notes that have a different duration than the corresponding notes in black.

- Image is under the Creative Commons Attribution-Share Alike 3.0 Unported license and was obtained from Wikimedia Commons: No modifications were made to this image. Original image originated between the years 1350 and 1400. https://commons.wikimedia.org/wiki/File:CordierColor.jpg
- For more information on *ars subtilior*, the type of musical style that this image demonstrates, see https://en.wikipedia.org/wiki/Ars_subtilior

**Part 1:** The *Ricecar a 6*, a six-part fugue from Johann Sebastian Bach's "The Musical Offering" (BWV 1079), written in Bach's hand.

- Image is in the public domain and obtained from Wikimedia Commons: https://commons.wikimedia.org/wiki/File:Ricercar_a_6_BWV_1079.jpg
- For more information, see https://en.wikipedia.org/wiki/The_Musical_Offering

**Part 2:** Front page of the autograph for J. S. Bach's "Sonata for single violin #1 in G minor" (BWV 1001)

- Image is in the public domain and obtained from Wikimedia Commons: https://commons.wikimedia.org/wiki/File:Bwv1001-adagio-handwriting.jpg
- For more information, see https://en.wikipedia.org/wiki/Sonatas_and_partitas_for_solo_violin_%28Bach%29

**Part 3:** "The Walrus and the Carpenter" from "Songs from Alice in Wonderland and Through the Looking-Glass"

- Image is in the public domain and obtained from Project Gutenberg: http://www.gutenberg.org/files/36308/36308-h/36308-h.htm
- According to https://www.gutenberg.org/wiki/Gutenberg:Permission_How-To, "Most permission request are not needed. The vast majority of Project Gutenberg eBooks are in the public domain in the US. This means that nobody can grant, or withhold, permission to do with this item as you please. "As you please" includes any commercial use, republishing in any format, making derivative works or performances, etc."
- The Project Gutenberg page for this book includes music notation written in MusicXML!

**Part 4:** Variation 30 of Johann Sebastian Bach's "Goldberg Variations" (BWV 988).

- Image is in the public domain and obtained from Wikimedia Commons: https://en.wikipedia.org/wiki/File:Quodlibet.jpg
- More information on Goldberg Variations: https://en.wikipedia.org/wiki/Goldberg_Variations

**Part 5:** The score of "Il Gran Mogol," a recently discovered flute concerto by Antonio Vivaldi. Identified in the National Archives of Scotland by Andrew Wooley.

- Image is in the public domain and obtained from the National Archives of Scotland. The copyright notice at http://www.nas.gov.uk/terms/copyright.asp states, "The material featured on this website (www.nas.gov.uk) is subject to Crown copyright protection unless otherwise indicated. The Crown copyright protected material may be reproduced free of charge in any format or medium provided it is reproduced accurately, not used in a misleading context and is acknowledged."

- For more information, see http://www.nas.gov.uk/about/101007.asp

**Part 6:** The *Gaudeamus omnes*, using square notation. From the 14th-15th century *Graduale Aboense*.

- Image is in the public domain and obtained from Wikimedia Commons: https://commons.wikimedia.org/wiki/File:Graduale_Aboense_2.jpg

- For more information, see https://en.wikipedia.org/wiki/Church_music

**Part 7:** "Tout par compas suy composes" by Baude Cordier (ca. 1380 – ca. 1440)

- Image is in the public domain and obtained from Wikimedia Commons: https://en.wikipedia.org/wiki/File:Cordier_circular_canon.gif

- You can learn more about Baude Cordier here: https://en.wikipedia.org/wiki/Baude_Cordier

- The musical notation is an example of "Eye music," more information for which may be found here: https://en.wikipedia.org/wiki/Eye_music