2

Using the JFugue MusicString

This chapter will explain all you need to know to start creating music with JFugue. Specifically, you will learn about the features of JFugue's MusicString. This will enable 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 JFugue MusicString.

Introducing the MusicString

The magic behind JFugue – the reason that JFugue is so easy to use and allows a programmer to create music so quickly – is the MusicString, a specially formatted String object that contains musical instructions.

For example, to play a C note, one simply needs to program the following:

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

JFugue parses the MusicString and creates objects behind the scenes to represent each note, instrument, and so on. These objects are then used to generate the music, and unleash a torrent of melody from your speakers.

Groovy Note

Incidentally, JFugue is one of the few Java libraries that lets you do something interesting in one or two lines. This unique capability led to JFugue being used in an example in Manning Publications' book, "Groovy in Action", in a demonstration showing how easy it is to load and use a third-party library with the Groovy scripting language.

The JFugue MusicString 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 the MusicString as readable as possible, adherence to this particular style is not required for JFugue to properly parse the MusicString. Style is addressed in more detail after the elements of the MusicString are introduced.

Learning the Parts of the MusicString

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 MusicStrings each contain one token, and the last four MusicStrings each contain eight tokens.

Notes, Rests, and Chords

The specification of a note or rest begins with the note name or the rest character, which is one of the following: `C`, `D`, `E`, `F`, `G`, `A`, `B`, or `R` for a rest. After specifying the note itself, you may then append a sharp or flat, octave, duration, or chord, all of which are described below.

A note can also be represented numerically. This could be useful if you are creating algorithmic music, in which each note may be indicated by a calculated value instead of a letter. A numeric note is specified by providing the note's MIDI value in square brackets, such as `[60]`. The octave is already factored into the note value, so it is not necessary (nor possible) to specify an octave when providing a note value. Values over 127 are not permitted.

Octave	C	C#/D*b*	D	D#/E*b*	E	F	F#/G*b*	G	G#/A*b*	A	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				

Figure 2.1 Numeric note values

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

If you use Key Signatures (explained below), 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 hadn't indicated the B as a natural, JFugue would automatically change the note value based on the key signature (so, if your key signature was F-major, that B would be converted into a B-flat automatically). Key Signatures are explained in more detail below.

Octave

Default: 5 for notes, 3 for chords

You may optionally specify an octave for the note, which is represented by a number from 0 through 10; for example, `C6` plays a C note in the sixth octave. If no octave is specified, the default for a note will be the fifth octave, and the default for a chord will be the third octave.

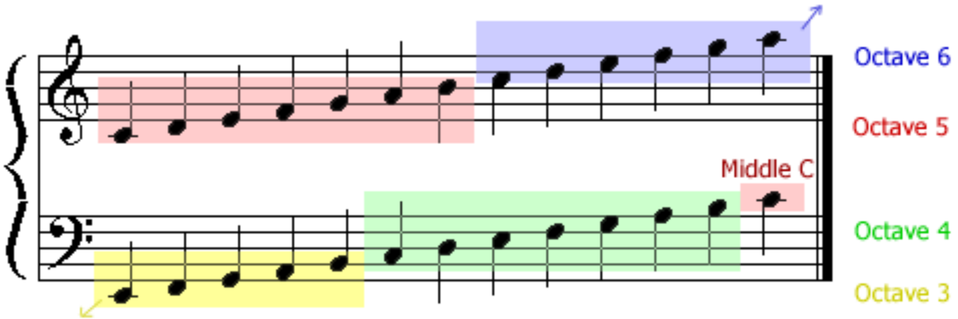


Figure 2.2 Octaves 0 through 10 span the various clefs; pictured above are Octaves 3 through 6.

Chords

If the given note is to be the root of a chord, the chord is specified next. JFugue supports a variety of chords, each of which is described in the table below.

Intervals in the table indicate the notes of the chord. For example, a major chord contains three notes, with intervals 0, 4, and 7. This means that the chord is comprised of the root (0), the root plus four half-steps (4), and the root plus seven half-steps (7). Therefore, a C-major chord is comprised of the notes C, E, and G.

Common Name	**JFugue Name**	**Intervals** (0 = root)
major	maj	0, 4, 7
minor	min	0, 3, 7
augmented	aug	0, 4, 8
diminished	dim	0, 3, 6
7th (dominant)	dom7	0, 4, 7, 10
major 7th	maj7	0, 4, 7, 11
minor 7th	min7	0, 3, 7, 10
suspended 4th	sus4	0, 5, 7
suspended 2nd	sus2	0, 2, 7
6th (major)	maj6	0, 4, 7, 9
minor 6th	min6	0, 3, 7, 9
9th (dominant)	dom9	0, 4, 7, 10, 14
major 9th	maj9	0, 4, 7, 11, 14
minor 9th	min9	0, 3, 7, 10, 14
diminished 7th	dim7	0, 3, 6, 9
add9	add9	0, 4, 7, 14
minor 11th	min11	0, 7, 10, 14, 15, 17
11th (dominant)	dom11	0, 7, 10, 14, 17
13th (dominant)	dom13	0, 7, 10, 14, 16, 21
minor 13th	min13	0, 7, 10, 14, 15, 21
major 13th	maj13	0, 7, 11, 14, 16, 21
7-5 (dominant)	dom7<5	0, 4, 6, 10
7+5 (dominant)	dom7>5	0, 4, 8, 10
major 7-5	maj7<5	0, 4, 6, 11
major 7+5	maj7>5	0, 4, 8, 11
minor major 7	minmaj7	0, 3, 7, 11
7-5-9 (dominant)	dom7<5<9	0, 4, 6, 10, 13
7-5+9 (dominant)	dom7<5>9	0, 4, 6, 10, 15
7+5-9 (dominant)	dom7>5<9	0, 4, 8, 10, 13
7+5+9 (dominant)	dom7>5>9	0, 4, 8, 10, 15

Figure 2.3 Chords supported by JFugue

To specify chords in a MusicString, provide the root's chord followed by the "JFugue Name" from the table above. For example, to play a C-major chord in the default octave, use the MusicString `Cmaj`. This is equivalent to saying `C+E+G`, but JFugue will automatically fill in the other notes based on the chord specified. Recall that the default octave for chords is the third octave, which is lower than the default fifth octave for individual notes.

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.

Chord Inversions

A chord inversion indicates another way to play the notes of a chord by changing which note in the chord serves as the root note. This is sometimes called the *voicing* of a chord.

A first inversion means the chord's regular root note should be moved up an octave, making the second note in the chord become the new bass note. A second inversion means the chord's root note and second note should be played an octave higher, making the chord's third note become the new bass note. Chords with more than three members can have third inversions, chords with more than four members can have four inversions, and so on. See Figure 2.4 for examples of chord inversions.

Chord inversions may also be described by explicitly indicating the note that is to become the new bass note. You may see this in sheet music when you're asked to play a C/E chord. This indicates that a C-Major chord should be played with the E note as the bass note.

There are two ways to specify chord inversions in JFugue. The first is consistent with indicating the first, second, third, etc. inversion of the chord. State your chord as indicated in the section above (for example, Cmaj for a C-Major), then use a caret character, ^, for each inversion. As shown in Figure 2.4, a first inversion becomes Cmaj^, and a second inversion becomes Cmaj^^. Additional inversions are possible with chords that have more member notes.

The second way is consistent with indicating the new bass note for the chord. Again, state the chord as indicated in the section above (Cmaj for a C-Major), then use the caret character, ^, followed by the new bass note. For example, the C-Major inversion with E as the new bass note would be Cmaj^E; the C-Major inversion with G as the new bass note would be Cmaj^G.

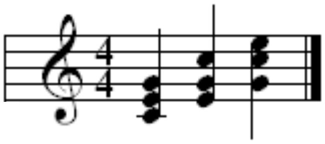


Figure 2.4 Chord inversions of C-Major: no inversion - Cmaj; first inversion - Cmaj^ or Cmaj^E; second inversion - Cmaj^^ or Cmaj^G

Duration

Default: Quarter duration ("q")

Duration indicates how long a note should be played. It is placed after the octave (or after the chord, if a chord is specified), or immediately after the note itself if the octave is omitted or if the note is specified as a value. Duration is indicated by one of the letters in the table below. If duration is not specified, the default duration of a quarter-note will be used.

Duration	Character
whole	w
half	h
quarter	q
e**i**ghth	i
sixteenth	s
thirty-second	t
si**x**ty-fourth	x
one-twenty-eighth	o

Figure 2.5 Durations that can be specified for a note

For example, a C6 note, half duration would be `C6h`, and a D-flat major chord, whole duration would be `DbmajW`.

Dotted duration may be specified by using the period character after the duration. For example, a dotted half note would be specified using `h` followed by a period (`h.`). A dotted duration is equal to the original duration plus half of the original duration. So, a dotted half note is equal to the duration of a half note plus a quarter note.

Durations may be appended to each other to create notes of longer durations. This is similar to a tie in musical parlance. For example, to play a `D6` note for three measures, the MusicString `D6www` would be used. (You could alternatively use ties and measure symbols to indicate ties in the MusicString, as described below.)

The duration may also be specified numerically. In this case, provide a decimal value equal to the part of a whole note. To indicate a numeric duration, use the slash character, followed by a decimal value. For example, to play an `A4` note for a quarter duration, provide the MusicString `A4/0.25`. A value of 1.0 represents whole duration. Decimal values greater than 1.0 indicate a note than spans multiple measures. For example, the `D6www` MusicString given above is equivalent to `D6/3.0`. Numeric durations may be useful for algorithmic music generators. They are also created when MusicStrings are generated when JFugue parses MIDI files, as explained more fully in Chapter 5.

Here are some examples of durations:

```
player.play("Aw");       // A5 whole note
player.play("E7h");      // E7 half note
player.play("[60]wq");   // Middle-C (C5) whole+quarter note
player.play("G8i.");     // G8 dotted-eighth note
player.play("Bb6/0.5"); // B-flat, 6th octave, half note

// C-major chord, second inversion, 7th octave, quarter note
player.play("C7maj^^q");
```

Triplets and Other Tuplets

Tuplets are groups of notes in which the duration of the notes is adjusted such that the duration of the group of notes is consistent with the duration of the next larger note duration. Figure 2.6 makes this a bit more clear.

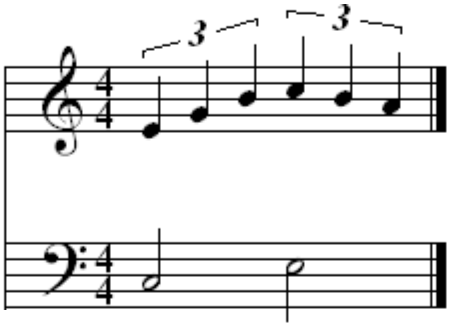


Figure 2.6 Two triplets (also known as 3-tuplets) of quarter notes. Notice how each triplet has the same duration of the half note in the bass staff.

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. For a triplet made up of quarter notes, as shown in Figure 2.6, this means the group of notes will be played in the duration of a half note, so each note in the triplet will be played at two-thirds (2/3) of its regular duration.

Consider tuplets of more notes – for example, a quintuplet, which consists of five notes. Five quarter notes may be played in the same duration as a whole note if they're part of a 5:4 tuplet, in which each note is played at 4/5 of its regular quarter duration.

To specify a tuplet in JFugue, use the asterisk, `*`, after the duration of a note that is part of a tuplet. For triplets, that's all you need to do. For other tuplets, the asterisk must be followed by the ratio that describes the tuplet, such as `5:4` in the example above. Each note in the tuplet must have the tuplet notation, and the ratio must be the same for each note in the tuplet (if it's not, nothing catastrophic will happen, but your music won't sound right).

Here are some examples:

```
player.play("Eq* Fq* Gq*");           // These two lines create
player.play("Eq*3:2 Fq*3:2 Gq*3:2"); // equivalent music
```

Each of these lines will play three quarter notes as a triplet. The group of three quarter notes will have the duration of a two quarter notes (identical to one half note).

```
player.play("Ci*5:4 Ei*5:4 Gi*5:4 Ei*5:4 Gi:5*4");
```

These five eighth notes (a quintuplet) will be played in the duration of a four eighth notes (identical one 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 2.7). 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 2.8).

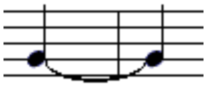


Figure 2.7 Tying two notes across a measure

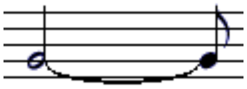


Figure 2.8 Tying two notes to achieve a combined duration

In JFugue, 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

[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 by JFugue.

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 2.9, which uses the Measure symbol (the vertical line or pipe character, `|`), which will be introduced soon.

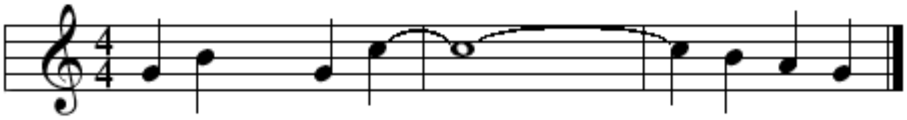


Figure 2.9 Examples of ties in a MusicString.
The MusicString for this sequence of notes is
"`G5q B5q G5q C6q- | C6-w- | C6-q B5q A5q G5q`"

Attack and Decay Velocities

Default: 64 for attack velocity, 64 for decay velocity

Notes may be played with a specified attack and decay velocity. These velocities indicate how long it takes for a note to "warm up" to its full volume, and "dissipate" from its peak volume. For example, a note with a long attack and a quick decay sounds like it build over a period of time, then turns off quickly. Notes with long attacks sound somewhat ethereal. Notes with a long decay sound like they continue to resonate after the note has been struck, like a bell or a guitar string.

Attack and decay for notes may be specified using the letters `a` and `d`, respectively. Each letter is followed by a value of 0 through 127; the default is 0. Low values indicate quicker attack or decay; high values indicate a long attack or decay. Either attack or decay may be used independently (but if they appear together, the attack must be specified first).

For example, the following are value notes with attack and decay velocities set:

```
player.play("C5qa0d127");      // Sharp attack, long decay
player.play("E3wwd0");         // Default attack, sharp decay
player.play("C7maja30");       // C7, E7, and G7 (components of
                               // C7maj) will all play with an
                               // of attack 30
```

Notes played in Melody and 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 2.10. So far, all of the MusicStrings examples have shown notes played in melody.

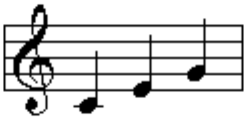


Figure 2.10 A melody; the MusicString is "`C5q E5q G5q`"

Notes may also be played in harmony – together with other notes. This can be indicated by combining the tokens with a plus symbol, `+`, instead of a space, as shown in Figure 2.11. Of course, notes in a chord are played in harmony automatically, but the `+` token lets you play *any* notes in harmony.

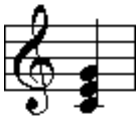


Figure 2.11 A harmony; the MusicString is "`C5q+E5q+G5q`"

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 look at Figure 2.12. In this example, the `C5` note is played continuously while the `E5` and `G5` notes are played in sequence.

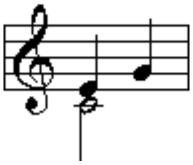


Figure 2.12 A harmony and a melody played together;
the MusicString is "`C5h+E5q_G5q`"

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

Measure

JFugue MusicStrings were created with the intention of making music creation easy; they were not developed to provide a fully complete syntax for representing sheet music. Indicating a bar line in a MusicString does not affect the musical output of the MusicString. Nevertheless, it is often useful to indicate the break between measures in a MusicString. To indicate a bar line, use the vertical line (or pipe) character, `|`, which must be separated from other tokens in the MusicString with spaces.

Key Signature

Default: C-major

A key signature may be indicated, which instructs JFugue to play the MusicString in a particular key or scale. To indicate a key signature, use the letter `K`, followed by the root of the key, then `maj` or `min` to indicate major or minor scale. For example, `KCbmaj` will set the key to C-flat major.

JFugue will automatically adjust the note values for the notes that are affected by the key signature. For example, if you set the key signature to F-major, then play a B note in your MusicString, JFugue will automatically convert that B to a B-flat. If you want the B to remain natural, you must indicate that by using the natural symbol, `n`, which is placed after the note. In this case, playing the B as a natural note would require the token `Bn`.

Instrument

Default: Piano

The music produced by JFugue uses MIDI to render audio that is played with instruments from the Java Sound soundbank. The MIDI specification describes 128 different instruments, and more may be supported with additional sound banks. Most MIDI devices use the same definitions for the first 128 instruments, although the quality of the sound varies by device and by soundbank. For example, MIDI instrument #0 often represents a piano, but the piano sound rendered by various MIDI devices may differ.

To select these instruments in JFugue's MusicString, use the instrument token, which is the `I` character followed by the instrument number from 0 to 127. For example, to specify a piano, you would enter the MusicString `I0`. Alternatively, JFugue defines *constants* that you can use to specify the instrument using the name of the instrument. This tends to be easier to read and remember. For example, the constant for a piano is PIANO, so the MusicString to specify a piano could also appear as `I[Piano]`. You can define your own constants as well; constants are described in more detail later in this chapter.

Figure 2.13 contains a list of instrument numbers and JFugue constants. You'll notice that some instruments contain more than one constant. In these cases, you can use either constant; they will both resolve to the same instrument number. Recall that the MusicString is not case-sensitive.

Piano	
0	PIANO *or* ACOUSTIC_GRAND
1	BRIGHT_ACOUSTIC
2	ELECTRIC_GRAND
3	HONKEY_TONK
4	ELECTRIC_PIANO *or* ELECTRIC_PIANO1
5	ELECTRIC_PIANO2
6	HARPISCHORD
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	ACCORIDAN
22	HARMONICA
23	TANGO_ACCORDIAN

Guitar	
24	GUITAR *or* NYLON_STRING_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

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	SYNTHBRASS_1
63	SYNTHBRASS_2

Figure 2.13 Instrument Values (continued on next page)

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	LEAD_SQUARE *or* SQUARE
81	LEAD_SAWTOOTH *or* SAWTOOTH
82	LEAD_CALLIOPE *or* CALLIOPE
83	LEAD_CHIFF *or* CHIFF
84	LEAD_CHARANG *or* CHARANG
85	LEAD_VOICE *or* VOICE
86	LEAD_FIFTHS *or* FIFTHS
87	LEAD_BASSLEAD *or* BASSLEAD

Synth Pad	
88	PAD_NEW_AGE *or* NEW_AGE
89	PAD_WARM *or* WARM
90	PAD_POLYSYNTH *or* POLYSYNTH
91	PAD_CHOIR *or* CHOIR
92	PAD_BOWED *or* BOWED
93	PAD_METALLIC *or* METALLIC
94	PAD_HALO *or* HALO
95	PAD_SWEEP *or* SWEEP

Synth Effects	
96	FX_RAIN *OR* RAIN
97	FX_SOUNDTRACK *or* SOUNDTRACK
98	FX_CRYSTAL *or* CRYSTAL
99	FX_ATMOSPHERE *or* ATMOSPHERE
100	FX_BRIGHTNESS *or* BRIGHTNESS
101	FX_GOBLINS *or* GOBLINS
102	FX_ECHOES *or* ECHOES
103	FX_SCI-FI *or* 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_TOM
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

Figure 2.13 Instrument values (continued)

Voice

Default: 0

Music is often broken down into multiple *voices*, also known as *channels* or *tracks*. Each voice contains a melody, often played with a specific instrument. For example, in a jazz song, you may have separate voices for the drums, the saxophone, the bass, and the piano. Or, in solo piano music, you can use one voice for treble clef, and one for the bass clef.

MIDI supports 16 simultaneous channels, which JFugue exposes through the Voice command. The Voice command is a `V`, followed by a number from 0 to 15.

MIDI editors often allow a song to be played with various channels turned on or off, so you can focus on one part of a song, or hear what a song would sound like without a certain voice.

MIDI Percussion Track

The tenth MIDI channel (i.e., `V9`) is special: it is the only channel that is capable of producing sounds for non-chromatic percussion instruments[2], typically drums. In the tenth channel, each note is assigned to a different percussion instrument. For example, if the tenth channel is given an A5 note (A note, 5th octave), it won't play an A5, but will instead play a bongo drum.

To make it easy to specify drum sounds for the tenth MIDI channel, JFugue provides a different way to specify notes in V9. Instead of entering `V9 A5q` and hoping for a bongo drum, you can use a *constant* to express the instrument more directly; in this case, you would enter `V9 [Hi_Bongo]q`. A list of constants representing percussion sounds is shown in Figure 2.14.

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.

[2] A percussion instrument is one that makes sound as a result of hitting or shaking things together. Examples include drums, tambourines, woodblocks, and cymbals. Chromatic percussion instruments are percussion instruments that can play notes, such as a steel drum. Non-chromatic percussion instruments can only make one sound, such as a snare drum, triangle, or cow bell.

Note Value	JFugue Constant
35	ACOUSTIC_BASE_DRUM
36	BASS_DRUM
37	SIDE_KICK
38	ACOUSTIC_SNARE
39	HAND_CLAP
40	ELECTRIC_SNARE
41	LOW_FLOOR_TOM
42	CLOSED_HI_HAT
43	HIGH_FLOOR_TOM
44	PEDAL_HI_TOM
45	LOW_TOM
46	OPEN_HI_HAT
47	LOW_MID_TOM
48	HI_MID_TOM
49	CRASH_CYMBAL_1
50	HIGH_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	LOW_BONGO
62	MUTE_HI_CONGA
63	OPEN_HI_CONGA
64	LOW_CONGO
65	HIGH_TIMBALE
66	LOW_TIMBALE
67	HIGH_AGOGO
68	LOW_AGOGO
69	CABASA
70	MARACAS
71	SHORT_WHISTLE
72	LONG_WHISTLE
73	SHORT_GUIRO
74	LONG_GUIRO
75	CLAVES
76	HI_WOOD_BLOCK
77	LOW_WOOD_BLOCK
78	MUTE_CUICA
79	OPEN_CUICA
80	MUTE_TRIANGLE
81	OPEN_TRIANGLE

Figure 2.14 Constants that represent percussion instruments, to be used in place of notes in V9.

Layer

A layer provides a way to specify separate melodies that are intended to be played in the same voice. Layers are specific to JFugue – they are not part of the MIDI specification. Layers were introduced to overcome a difficulty in programming music for the tenth MIDI channel – the one that plays percussion instruments. Specifically, if you had numerous melodies that each had their own rhythm, it would be difficult to combine these as "chords" in that voice. Using layers, you can easily combine a melody of, say, hand claps, snare drums, and cow bells.

In addition, layers can be used in other voices, too. They could be leveraged as a way to simulate getting more than 16 simultaneous melodies out of a MIDI system. They can also be used to send multiple events in the same track – for example, to change the pitch wheel while a note is playing, to produce a modulation of the playing note.

Like the voice token, a layer token is specified by using `L`, followed by a number from 0 to 15.

Tempo

Default: 120 beats per minute (roughly Allegro)

The tempo indicates how quickly a song should be played. It is often one of the first things set in a MusicString, since it applies to all musical events that follow the tempo command.

Tempo represents beats per minute (BPM). In older versions of JFugue, tempo represented "pulses per quarter" (PPQ), which indicates how many clock cycles to give a quarter note. PPQ is inversely proportional to BPM. Of course, PPQ is not intuitive, so JFugue now supports expressing tempo using BPM. Fortunately, the most common tempo setting, 120, happens to be an equivalent value for PPQ and BPM (120 PPQ = 120 BPM)[3].

The tempo token is a `T`, followed by an integer, or by one of the tempo constants in brackets, such as `T[Adagio]`. Figure 2.15 lists the tempo constants that you can use in your MusicStrings.

JFugue Constant	Beats Per Minute (BPM)
Grave	40
Largo	45
Larghetto	50
Lento	55
Adagio	60
Adagietto	65
Andante	70
Andantino	80
Moderato	95
Allegretto	110
Allegro *(default)*	**120**
Vivace	145
Presto	180
Pretissimo	220

Figure 2.15 Tempo constants that can be used with the `T[]` command

Pitch Wheel

The pitch wheel is used to change the pitch of a note by hundredths of a half-step, or cents. The pitch wheel can be used to change the frequency of an individual note 8192 cents in either the downward or upward direction.

The pitch wheel can be used to create Theremin-like effects in your music. JFugue also uses the Pitch Wheel to make microtonal adjustments for notes, enabling some Eastern styles of music to be played easily.

[3] To convert BPM to PPQ, and back, divide 60,000,000 by the opposite (PPQ or BPM) value. In other words, BPM = 60,000,000/PPQ, and PPQ = 60,000,000/BPM.

The token to adjust the pitch of following notes is an ampersand, `&`, followed by an integer value from 0 through 16383. Values from 0 through 8191 make the pitch of the following notes lower; values from 8193 through 16383 make the pitch of the following notes higher. To reset the pitch wheel so it makes no changes to the notes, use `&8192`.

Channel Pressure

Many MIDI devices are capable of applying pressure to all of the notes that are playing on a given channel.

The MusicString token for channel pressure is a plus symbol, `+`, followed by a value from 0 to 127. It applies to the channel indicated by the most recent voice token used in the MusicString.

Don't confuse this token with the use of a plus symbol to connect notes within a harmony – in the channel pressure case, the token *begins* with a plus, so it is parsed differently.

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.

The MusicString token for Polyphonic Pressure is an asterisk symbol, `*`, 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 MusicString applies a pressure of 75 to Middle-C (note 60): `*60,75`. Note that this command does not accept note values, so using `C5` in this case would not 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. One way to remember the difference between the JFugue tokens for channel pressure versus polyphonic pressure is that plus character, `+`, representing channel pressure, represents a concept slightly simpler than the asterisk character, `*`, which represents polyphonic pressure.

Controller Events

The MIDI specification defines about 100 controller events, which 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 lots more. For a complete list, refer to a MIDI specification document.

The Controller command, X, tells JFugue to set the given controller:

```
Xcontroller_number=value
X37=18
X[Chorus_Level]=64
```

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:

```
X[Foot_Pedal_Coarse]=10
X[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:

```
X[Foot_Pedal]=1345
```

There you have it. Want to set the volume to 10200, out of a possible 16383? There's no need to figure out the high byte and low byte of 10200. Just use X[Volume]=10200. JFugue will split the values into high and low bytes for you.

Some controller events have two settings: ON and OFF. Normally, ON means 127 and OFF means 0. JFugue has defined two constants, ON and OFF, that you can use instead of the numbers: X[Local_Keyboard]=ON. JFugue has also defined DEFAULT, which is set to 64.

Controller	JFugue Constant
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 *or* HOLD
65	PORTAMENTO
66	SUSTENUTO_PEDAL *or* SUSTENUTO
67	SOFT_PEDAL *or* SOFT
68	LEGATO_PEDAL *or* LEGATO
69	HOLD_2_PEDAL *or* HOLD_2
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_BUTTON_1
81	GENERAL_BUTTON_2
82	GENERAL_BUTTON_3
83	GENERAL_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

Figure 2.16 Controller constants that can be used with the `X[]` command

Combined Contoller	JFugue Constant
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

Figure 2.17 Combined controller constants. Integers can be assigned to these, and JFugue will figure out the high and low bytes.

Constants

When you're programming music, your main task is to make beautiful sounds, not to be inundated with random and meaningless numbers. You should be able to set the VOLUME and use the FLUTE, without having to remember that VOLUME is controller number 935 (or, worse, that VOLUME is comprised of a coarse and fine value) and FLUTE is instrument number 73. To enable you to sit back, relax, and focus on concepts instead of numbers, JFugue has introduced *constants* that you can use in your MusicStrings, and that get resolved as the music is playing.

The command to set a constant is as follows:

```
$WORD=DEFINITION
```

Here's an example: `$ELEC_GRAND=2`. Of course, JFugue has already defined ELECTRIC_GRAND to be 2. But maybe you'd like to use a shorter name, or maybe you have a more memorable name for this instrument, like simply ELEC. You could then use your shorter name in the MusicString when you want to refer to this particular instrument.

JFugue defines a bunch of constants - around 375 - for things like instrument names, percussion instruments, tempo, and controller events. Creating these defintions is the job of the JFugueDefinitions class.

Constants are also useful in cases where you have settings that you may want to change some day. Suppose you want to play some music with your favorite instrument, the piano. You could definine FAV_INST to be 0, and then you can say `I[Fav_Inst]` whenever you want to use it. If your favorite instrument changes, all you have to change in your music string is the definition of FAV_INST; you do not have to change every place where you refer to your favorite instrument.

You can use a constant anyplace where a number would be expected, with the exception of the Octave value (but that's okay, because you can just specify the note itself, with the octave, as a single number), and if you're using a constant for a duration, you have to use decimal duration values (and precede the duration with a slash, `/`).

When using a constant in the MusicString, always place the word in square brackets.

Timing Information

When transcribing notes from sheet music, you will find that through a combination of rests and note durations, you can successfully create music that has the proper time delays between notes. However, when reading music from a MIDI file, notes are not guaranteed to follow each other in such a formal way. For this reason, JFugue uses the Time token to indicate the number of milliseconds into the sequence to play notes and other tokens. You will hardly ever need to use this when creating your own music, but you'll see it if you convert music from MIDI to a MusicString (which is discussed in more detail in Chapter 5).

The Time token is an ampersand, `@`, followed by a time in milliseconds. The time indicates when the following tokens should be played.

It is not necessary for the times to be sequential. The full JFugue MusicString is parsed before music is rendered, and timing information that represents any time will be played at the right time during playback.

MusicString Style

The following guidelines are recommended to help you create MusicStrings that are easy to read and easy to share. MusicStrings are not case-sensitive, so the use of upper- and lowercase characters can be used to maximize the MusicString's readability.

1. Use a capital letter for a character representing an instruction: `I`, `V`, `L`, `T`, `X`, and `K` (for Instrument, Voice, Layer, Tempo, Controller, and Key Signature, respectively)
2. Use a capital letter for notes. `C`, `D`, `E`, `F`, `G`, `A`, `B`, and the rest character, `R`.
3. Use lowercase characters when specifying chords: `maj`, `min`, `aug`, and so on.
4. Use a lowercase letter for note durations: `w`, `h`, `q`, `i`, `s`, `t`, `x`, `o`. However, if you are consistently using durations after chords, it may be more legible to use uppercase letters for note durations.
5. Use mixed case (also known as camel case) to represent instrument names, percussion names, tempo names, or controller names: `I[Piano]`, `[Hand_Clap]`, `T[Adagio]`,`X[Hold_Petal]`.
6. Use all capital letters when defining and referring to a constant: `$MY_WORD=10`.
7. 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.

8. Use the vertical bar character (also known as pipe), |, to indicate measures in a MusicString.

Below are a couple of sample MusicStrings that employ some of these guidelines.

```
Player player = new Player();

// 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^^");
```

JFugue Elements: Using Objects instead of MusicStrings

So far in this chapter, you have learned how to create MusicStrings using JFugue's notation. You have not learned how to construct a song by creating many individual note objects and adding them together, mainly because creating music in such a way would be extremely tedious. It is far easier to craft a MusicString, and let JFugue create the objects behind the scenes.

However, there may be cases in which you *would* want to create music by instantiating individual objects. Perhaps you want to build a loop that actually generates Note objects. Or, maybe you want compile-time checking of values that you're passing as instruments, tempos, or percussive notes.

JFugue does provide the ability to create any musical element by instantiating a class, and adding the musical elements to a Pattern. You'll learn more about Patterns in the next chapter, but in the meantime, all you need to know is that a Pattern is a piece of music that can be played by the Player, and that you can add musical elements to it.

The following are the various calls that you might use to create an instance of a musical event. In some cases, you can use predefined constants to represent values (such as `Tempo.ADAGIO` and `Instrument.PIANO`).

```
// Create a new Voice instance
Voice voice = new Voice(byte voiceValue);

// Create a new Layer instance
Layer layer = new Layer(byte layerValue);

// Create a new Tempo instance (two examples)
Tempo tempo = new Tempo(int tempoInBPM);
Tempo tempo = new Tempo(Tempo.ADAGIO);
```

```
// Create a new Instrument instance (two examples)
Instrument instrument = new Instrument(byte instrumentValue);
Instrument instrument = new Instrument(Instrument.PIANO);

// Create a new Note instance (four examples)
Note note = new Note(byte value, long durationInMilliseconds);
Note note = new Note(byte value, double decimalDuration);
Note note = new Note(byte value, long durationInMilliseconds,
 byte attackVelocity, byte decayVelocity);
Note note = new Note(byte value, double decimalDuration,
 byte attackVelocity, byte decayVelocity);

// Create a new Pitch Bend instance
PitchBend pitchBend = new PitchBend(byte leastSignificantByte,
 byte mostSignificantByte);

// Create a new Channel Pressure instance
ChannelPressure channelPressure = new ChannelPressure(byte
 pressure);

// Create a new Polyphone Pressure instance
PolyphonicPressure polyPressure = new PolyphonicPressure(byte
 key, byte pressure);

// Create a new Measure instance
//   (which is purely decorative and results in no music)

Measure measure = new Measure();

// Create a new Key Siganture instance
//   keySig: a value from -7 to +7.  -7 means 7 flats, +7
//   means 7 sharps,  0 means no flats or sharps.
//   scale: 0 for major, 1 for minor
//   This follows the MIDI Specification on Key Signature.
KeySignature keySig = new KeySignature(byte keySig, byte
 scale);

// Create a new Controller instance
//   Note that for controllers that have a Coarse and Fine
//   setting, each of those settings has to be
//   instantiated individually.
Controller controller = new Controller(byte index, byte
 value);

// Create a new Time instance
//  milliseconds: The millisecond position of the next
//  musical event
Time time = new Time(long milliseconds);
```

Once you create these classes, you add them to a Pattern using the `addElement(JFugueElement element)` call:

```
Pattern pattern = new Pattern();
pattern.addElement(voice);
pattern.addElement(instrument);
pattern.addElement(note);

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

The music generated by this pattern would be equivalent to the music generated from a MusicString that contained the same musical events expressed in JFugue's notation:

```
Player player = new Player();
player.play("V0 I[Piano] C5q");
```

If you were so inclined, you could even create an array of note values and durations and use those to construct a large number of Note objects, then add the Note objects to a Pattern and play the Pattern. This is shown in the following example, "The First Measure of *Inventio 13*, The Hard Way":

```
// Define the value and decimal duration for each note
byte[] noteValues = new byte[]
  { 64, 69, 72, 71, 64, 71, 74, 72, 76, 68, 76 };
double[] durations = new double[]
  { 0.0625, 0.0625, 0.0625, 0.0625, 0.0625, 0.0625, 0.0625,
    0.125, 0.125, 0.125, 0.125 };

// Create a Pattern and set the tempo and instrument
Pattern pattern = new Pattern();
pattern.addElement(new Tempo(110));
pattern.addElement(new Instrument(Instrument.HARPISCHORD));

// Build up the pattern using the note values and durations
for (int i=0; i < noteValues.length; i++) {
    Note note = new Note(noteValues[i], durations[i]);
    pattern.addElement(note);
}

// Play the pattern
Player player = new Player();
player.play(pattern);
```

This method of creating music is not optimal in cases where you're transcribing known music. However, it might be useful when creating music algorithmically. Although truthfully, that's more of a rationalization on behalf of this author as opposed to an actual endorsement.

Getting Assistance with Notes

One might argue that notes are central to creating music. As such, notes are used in a variety of circumstances outside of the MusicString. For example, in Chapter 7 you'll learn about Interval notation, which allows you to specify music in terms of intervals (the differences between notes) instead of concrete notes themselves; you then pass a root note to the Interval notation class to create specific instances of notes based on the intervals provided.

As you know, notes can be specified in a MusicString using notation like `C5`. Or, notes can be specified using MIDI note values, like `[60]`. It is the MusicStringParser that is able to convert something like `C5` into a meaningful note value. Most methods aren't backed by a MusicStringParser, and those methods expect note values. The problem is that notation like `C5` becomes very comfortable after a while.

To make your musical life as simple as possible, the Note class has a number of static methods that can produce MIDI note values given MusicString-like notation for notes (would that be note-tation?). Additionally, you may find it necessary to convert note values to Strings, to find the duration letter for a specific decimal duration, and so on. The Note class contains the following static methods that you may use at any time:

```
    // Returns a MusicString represention of the given MIDI note value.
    // For example, given 60, this method returns C5.
    public static String getStringForNote(int noteValue)

    // Returns a MusicString represention of the given MIDI note value
    // and decimal duration.  For example, given 60 and 0.5,
    // this method returns C5h.
    public static String getStringForNote(int noteValue, double
decimalDuration)

    // Returns the frequency, in Hertz, for the given note value.
    // For example, the frequency for A5 (MIDI note 69) is 440.0
    public static double getFrequencyForNote(int noteValue)
```

JFugue also contains the following methods for converting decimal values to MusicStrings representing duration, and vice versa. The first of these methods, `getStringForDuration(double decimalDuration)`, returns a MusicString representation of the given a decimal duration. For example, given 0.5, this method returns `h`. This method only converts single duration values (not compound durations, like 3.0) representing whole, half, quarter, eighth, sixteenth, thirty-second, sixty-fourth, and one-hundred-twenty-eighth durations; and dotted durations associated with those durations (such as 0.75, representing `h.`). This method does not convert combined durations (for example, 0.625, epresenting `hi` for 0.625) or anything greater than a duration of 1.0 (for example, 4.0, representing wwww). For these values, the original decimal duration is returned in a string, prepended with a / to make the returned value a valid MusicString duration indicator.

The second method, getDecimalForDuration(String stringDuration), acts in a similar way: it takes a String representing a single duration character or a dotted duration character, and returns the decimal value corresponding to that duration. This method does not work on combined duration strings, like hi or wwww.

Finally, the Note class contains a public static String array, NOTES, which contains string representations of each of the twelve notes in an octave:

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

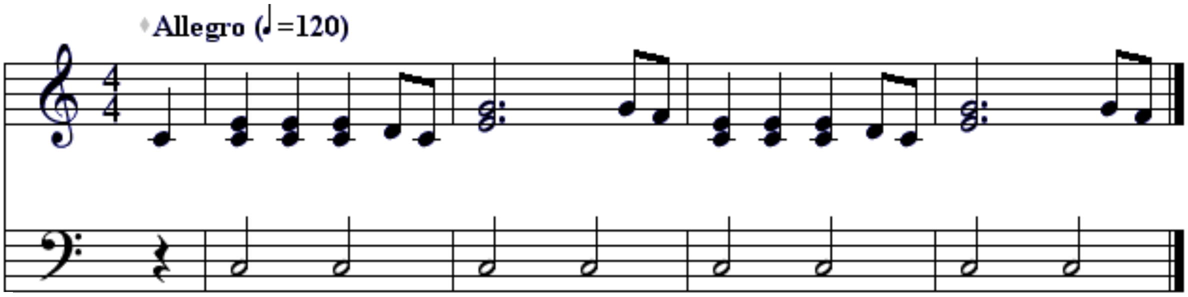
Transcribing Sheet Music to JFugue MusicString

This section describes how to transcribe sheet music to JFugue notation. 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 the next chapter. For now, all you need to know is that the Pattern class is an object that contains a MusicString.

"Spring", from "The Four Seasons"

Antonio Vivaldi



The first thing to notice is that there are two clefs, the treble clef and the bass clef, which means we'll want to enter the music into two voices – that is, two pieces of music that can be played in harmony. We'll put notes from the treble clef into Voice 0, and notes from the bass clef into Voice 1.

You'll also notice that there's a Tempo, so we can enter that as well. So far, we would have this MusicString:

```
    Pattern pattern = new Pattern("T[Allegro]");
    pattern.add("V0 notes-for-treble-clef");
    pattern.add("V1 notes-for-bass-clef");

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

There is a time signature represented in the sheet music, but JFugue does not currently contain a time signature token. This is because the time signature is important for knowing how many notes appear within a measure. However, JFugue is capable of playing notes with just the tempo and the note durations. In fact, even the bar lines are optional in JFugue's notation.

Let's start entering notes for the treble clef. We first see a C-note, quarter duration. Recall from Figure 2.2 that this note is in the fifth octave. That means we have `C5q`.

Next is a bar line, indicating the end of the first measure. We'll add a pipe symbol, `|`, to our MusicString; this will aid in legibility of the music.

Then we see a C and E note played in harmony. These are quarter notes again. The notation for these notes is `E5q+C5q`. 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 `D5i` and `C5i` to our MusicString.

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 `Eh.+Gh.` to the MusicString.

At this point, our MusicString should look like this:

```
V0 C5q | E5q+C5q E5q+C5q E5q+C5q D5i C5i | 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've already types. We have a couple of options here. The most obvious option is that we can re-type the notes. 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.

Therefore, our MusicString now looks like this:

```
V0 C5q | E5q+C5q E5q+C5q E5q+C5q D5i C5i | Eh.+Gh. G5i F5i |
E5q+C5q E5q+C5q E5q+C5q D5i C5i | Eh.+Gh. G5i F5i
```

Now we can work on the bass clef. First, there's a quarter rest. It's very important to add this rest to the MusicString, 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 Figure 2.2, these notes are in Octave 4. Add these notes to the bass clef, and the MusicString should look like this:

```
V1 Rq | C4h C4h | C4h C4h | C4h C4h | C4h C4h
```

Therefore, the program itself should look like this:

```
    Pattern pattern = new Pattern("T[Allegro]");
    pattern.add("V0 C5q | E5q+C5q E5q+C5q E5q+C5q D5i C5i |
Eh.+Gh. G5i F5i | E5q+C5q E5q+C5q E5q+C5q D5i C5i | Eh.+Gh. G5i
F5i");
    pattern.add("V1 Rq | C4h C4h | C4h C4h | C4h C4h | C4h C4h");
```

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

```
    // This looks better on a larger display!
    Pattern pattern = new Pattern("T[Allegro]");
    pattern.add("V0 C5q | E5q+C5q E5q+C5q E5q+C5q D5i C5i |
Eh.+Gh. G5i F5i | E5q+C5q E5q+C5q E5q+C5q D5i C5i | Eh.+Gh. G5i
F5i");
    pattern.add("V1 Rq  | C4h C4h                         | C4h
C4h             | C4h C4h                         | C4h C4h
   ");

    // Now, play the music!
    Player player = new Player();
    Player.play(pattern);
```

Congratulations! You can now transcribe music to JFugue.

(As you'll learn later, you can also save and load Patterns, so it's easy to share your transcriptions online, over webpages, through email, etc.)

3

Working with Patterns

This chapter introduces the second important concept in JFugue: the Pattern, a fragment of music that can be combined with other patterns, changed in interesting ways, and more – all of which facilitates experimentation with music in more ways than the MusicString can do on its own. The Pattern compliments the MusicString by providing an assortment of additional behaviors that can really make your music sing.

What is a Pattern?

To start with the basics: a Pattern is an object that contains a MusicString. For example, let's create a Pattern that we can play with throughout this chapter. The code sample below represents the first two measures of "Twinkle Twinkle Little Star".

```
Pattern pattern1 = new Pattern("C5q C5q G5q G5q A5q A5q Gh");
```

One thing we can do with this Pattern is pass it to a Player object, just like we would pass a MusicString. For example:

```
Player player = new Player();
Pattern pattern1 = new Pattern("C5q C5q G5q G5q A5q A5q Gh");
player.play(pattern1);
```

By itself, that isn't very interesting, but it's an important first step to working with patterns.

Using Patterns as Musical Building Blocks

You can think of the pattern as a musical building block. Patterns can be added together, thereby creating larger pieces of music. If you imagine a song that has a repeated section, you could create a pattern that represents that repeated section, and add that pattern to the song every time you need it. To continue with the "Twinkle Twinkle Little Star" example, we might create the following additional patterns:

```
// This is "Twinkle, twinkle, little star"
Pattern pattern1 = new Pattern("C5q C5q G5q G5q A5q A5q Gh");

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

// This is "Up above the world so high"
// and "Like a diamond in the sky"
Pattern pattern3 = new Pattern("G5q G5q F5q F5q E5q E5q D5h");
```

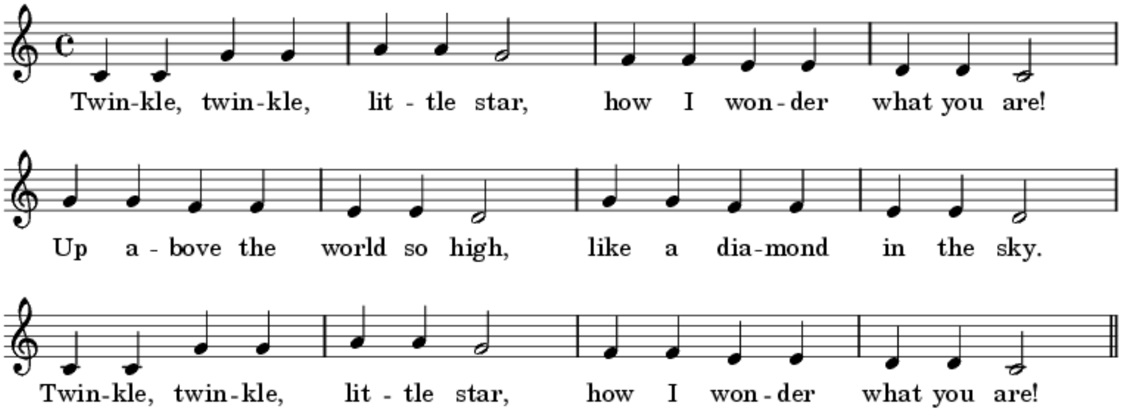

Twin-kle, twin-kle, lit - tle star, how I won-der what you are!
Up a - bove the world so high, like a dia-mond in the sky.
Twin-kle, twin-kle, lit - tle star, how I won-der what you are!

Figure 3.1 Sheet music for "Twinkle Twinkle Little Star"

Each of the three patterns is repeated twice in the song. If we create a pattern representing the whole song, we can add the repeated patterns to the song when we need them:

```
Pattern twinkleSong = new Pattern();
twinkleSong.add(pattern1);
twinkleSong.add(pattern2);
twinkleSong.add(pattern3);
twinkleSong.add(pattern3);
twinkleSong.add(pattern1);
twinkleSong.add(pattern2);
```

Now we can pass the `twinkle` Pattern to the Player, and it will play the entire song:

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

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 Strings, because when Strings are concatenated together, new String objects are formed.

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

```
Pattern pattern1 = new Pattern();
pattern1.add(pattern2);  // Adding a Pattern to a Pattern
pattern1.add("C5q C5q"); // Adding a MusicString 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 MusicStrings 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);
```

JFugue objects representing musical events may also be added to a pattern by using the `addElement()` command. This would be most commonly used if you're constructing a pattern from within an object that implements the ParserListener interface. That interface defines a number of methods which are called when music is parsed by a Parser, and when any of those methods is called, a JFugueElement-derived class, such as `Note` or `Tempo`, is provided to the listener method.

```
// Add a Middle-C, half duration to pattern1
pattern1.addElement(new Note(60, 0.5));
```

Patterns can also repeat themselves. For example, if you have a pattern like a drum beat and you want to make it play again and again for some set number of iterations, you can call `pattern1.repeat(x)`, which will modify the pattern in-place (i.e., it does not return a new Pattern object):

```
// Repeat pattern1 four times
pattern1.repeat(4);
```

Methods to repeat only a subset of a pattern also exist. For example, if you have a pattern that represents the MusicString `T[Allegro] I[Piano] C5q C5q G5q`, and you only want to repeat the C notes twice, you can call `repeat()` and indicate the start and end index of the tokens you want to repeat:

```
// Repeat: number of times, start index, end index
pattern1.repeat(2, 2, 3);
```

You may also give repeat a start index, and the end index will be assumed to be the end of the MusicString:

```
// Repeat: number of times, start index.  End index is length
// of music string.
pattern1.repeat(2, 2);
```

If at any point you would like to see the contents of the Pattern, the following methods are available:

- `getMusicString()` will return the full MusicString maintained by the Pattern
- `getTokens()` will return an array of Strings that represents each token of the MusicString

Observing Changes to a Pattern with a PatternListener

If you're interested in knowing when something has been added to a particular pattern, you can register a PatternListener on that pattern. Whenever a String, Pattern, or JFugueElement is added to a pattern, the `fragmentAdded(Pattern pattern)` method will be called on the listener. The only exception to this is if you add a Note that plays concurrently or in sequence with another note (i.e., the Note has a "+" or "_" associated with it, as discusses in Chapter 1). In this case, you will not receive an event through the `fragmentAdded()` callback.

Here's an example using an anonymous inner class for the listener.

```
Pattern pattern = new Pattern();
pattern.addPatternListener(new PatternListener() {
 public void fragmentAdded(Pattern addedFragment) {
  System.out.println("A new fragment has been added to this
 pattern: " + addedFragment);
 }
} );
```

Maintaining Properties within a Pattern

You can associate title, author, and other information with your pattern using the `setProperty(String key, String value)` and `getProperty()` methods. These methods are used to maintain a `Map<String, String>` of keys and values that is kept with each pattern.

You can also access the title independently, using `setTitle(String title)` and `getTitle()`. These methods will put a new value into the pattern's map using the key defined by `Pattern.TITLE`.

Here are some examples of using the properties on a Pattern:

```
Pattern pattern = new Pattern("C5q C5q G5q G5q A5q A5q Gh");
pattern.setTitle("Twinkle, Twinkle Little Star");
pattern.setProperty("Author", "unknown");
pattern.setProperty("Date Created", "May 8, 2008");
pattern.setProperty("Type", "nursery rhyme ");

System.out.println("Title is " + pattern.getTitle());
System.out.println("Type is " + pattern.getProperty("Type"));
```

The `getProperties()` method can be used to retrieve the actual `Map<String, String>` that contains the Pattern's property information. You may want to obtain the map to iterate over the keys or values using methods from the `java.util.Map` class.

The Pattern class also includes two methods that return String objects that contain the information that has been set in the pattern's property map. The `getPropertiesAsSentence()` method returns a single String that is composed of a concatenation of keys and values, separated my semicolons, as shown here:

```
key1: value1; key2: value2; key3: value3
```

The `getPropertiesAsParagraph()` method returns a single String that is composed of keys and values, separated my newline characters (\n), as shown here:

```
key1: value1\n
key2: value2\n
key3: value3\n
```

You could imagine using the properties on a pattern to facilitate management of many patterns. For example, you could create a jukebox of JFugue tunes that displays each song in a list, and that indexes each song with its author and title information.

Loading and Saving Patterns

Patterns can be saved and loaded. Saved pattern files, which should be named with a "`.jfugue`" file extension, are stored in JFugue's file format, which is simply this:

- A hash or pound character (`#`) as the first character of a line indicates a comment or a property
- Properties are noted by commented lines that contain a colon (`:`) anywhere in the line. The text to the left of the colon (except the `#` character) is the property key; the text to the right of the colon is the value
- All other lines are expected to be valid JFugue MusicStrings

Here's an example of a `.jfugue` file:

```
#
# Title: Twinkle, Twinkle Little Star
# Date Created: May 8, 2008
# Type: nursery rhyme
# Author: unknown
#

C5q C5q G5q G5q A5q A5q Gh F5q F5q E5q E5q D5q D5q C5h G5q
G5q F5q F5q E5q E5q D5h G5q G5q F5q F5q E5q E5q D5h C5q
C5q G5q G5q A5q A5q Gh F5q F5q E5q E5q D5q D5q C5h
```

To save a pattern, use the pattern's `savePattern(File)` method. This method will throw an `IOException` if there is a problem writing the file. Here's an example:

```
        pattern.savePattern(new File("twinkle.jfugue"));
```

You will notice that the MusicStrings saved into the `.jfugue` file are split into lines of approximately 80 characters each. This is intended to improve the human readability of the JFugue files. (If you create a file by hand, the lines can be as long or as short as you'd like).

Patterns may be loaded as well. The Pattern class contains a static method, `loadPattern(File)`, which returns a Pattern object, complete with MusicStrings and any available properties. This method will also throw an `IOException` if there is a problem reading the file.

```
      Pattern p = Pattern.loadPattern(new File("twinkle.jfugue"));
```

Why not just save() and load()?

You might be asking yourself why the Pattern class has methods called savePattern() and loadPattern(). After all, doesn't Pattern.loadPattern() sound a little redundant? Why not just call these things save() and load()?

The reason is that JFugue works with MIDI, too – and not just with MIDI Parsers and MIDI Renderers, but with loading and saving actual MIDI files, as you'll read about in the chapter about MIDI. Patterns have a role in that, too. Therefore, calling these methods simply save() and load() is ambiguous about what types of files are being saved and loaded. Pattern.loadPattern() might sound redundant, but at least it's not confusing!

Transforming Patterns with PatternTransformer

So far, you've seen how patterns can be created and managed, but you haven't seen some of the fun and intriguing things that can be done to patterns. That's about to change.

First, we need to provide a little background. When you create a Pattern or MusicString and pass it to `player.play()`, JFugue invokes a parser that converts your strings into musical events (This is discussed in more detail in Chapter 8). Those musical events are sent out to any class who wants to listen to them – that is, any class that implements the ParserListener interface, and has been registered with the parser. Typically, the MidiRenderer class, which is a type of ParserListener, is listening to the MusicStringParser, so your MusicString can be converted into actual MIDI music.

Now for the fun part: Any class can listen to the musical events, and that class can do whatever it wants with those events. The class could create MIDI; it could create sheet music; or it could change those events around and return a new pattern!

JFugue has an abstract class called a PatternTransformer, and these are exactly the kinds of things that it allows you to do. PatternTransformer implements the ParserListener interface, so it receives musical events when they are parsed. It also provides a foundation for creating a new Pattern object based on the musical events that are parsed by a Parser.

What are some types of things that you could do with a Pattern Transformer? Here are several ideas:

- Randomize the notes in a Pattern
- Perform certain notes in a Pattern with a tremolo effect
- Add an echo to each of the notes
- Increase the intervals between notes
- Increase the duration between notes
- Sort the notes in order of tone, in order of duration
- Reverse the order of the notes
- Replace all of one type of note with another note
- Replace specific sequences of notes with different sequences

Since JFugue allows you to read in MIDI files, you could even do these transformations on MIDI music that you might have created using other tools. Or, you could compose music in another tool, then bring it into JFugue to apply some interesting transformations.

PatternTransformers Included with with JFugue

JFugue comes with the following PatternTransformer classes, which can be found in the `org.jfugue.extras` package:

DiatonicIntervalPatternTransformer
Transposes all notes in the Pattern by a diatonic interval: 1 - unison, 2 - second, and so on; 8 - octave.

DurationPatternTransformer
Multiplies the duration of all notes in the given Pattern by a factor passed in as a parameter.

IntervalPatternTransformer
Changes the interval, or step, for each note in the given Pattern. For example, a C5 raised 3 steps would turn into a D#5. The interval is passed in as a parameter.

ReversePatternTransformer
Reverses the given Pattern.

How to Create a PatternTransformer

If you'd like to create your own PatternTransformer, it's easy to do. There are some key facts to know about the PatternTransformer class. The first is that PatternTransformer maintains a return pattern, which is the pattern to which the class's results should be added. The return pattern can be accessed by using the `getReturnPattern()` method. That return pattern will be used at the end of a transformation to receive the changed music.

Second, PatternTransformer implements all of the methods from ParserListener; for each, the default action is to simply add the incoming event to the return pattern, which means that PatternTransformer by itself would return an unaltered pattern. This also means that when creating your class that extends PatternTransformer, you only need to override those callbacks that you're specifically interested in. For example, if you're just interested in changing notes, you don't need to write code that handles instrument, voice, or controller events. In callbacks that you do override, be sure to make your modifications to the PatternTransformer's return pattern, which again you can access using the protected `getReturnPattern()` method.

Third, keep in mind that note events can come in from three separate callbacks: `noteEvent()`, `parallelNoteEvent()`, and `sequentialNoteEvent()`. Chances are that if you're playing with notes, you'll want to override each of these methods – and probably have each of them call the same new method that you'll create to do the work of adding the note to the return pattern.

Below is a sample from ReversePatternTransformer. Notice the way this transformer works: for each of the musical elements that is passed into a callback, this class inserts that element into the beginning of the return pattern. Notice that it has to do a little extra work to make parallel and sequential notes work correctly, so `A5q+B5q` becomes `B5q+A5q` and not `B5q A5q+`.

```
public class ReversePatternTransformer extends PatternTransformer
{
    public ReversePatternTransformer()
    {
        super();
    }

    public void voiceEvent(Voice voice)
    {
        insert(voice.getMusicString(), " ");
    }

    // Most of the other events follow the same steps as
    // voiceEvent: insert(element.getMusicString()," ");

    public void noteEvent(Note note)
    {
        insert(note.getMusicString(), " ");
    }

    public void sequentialNoteEvent(Note note)
    {
        insert(note.getMusicString().substring(1,
note.getMusicString().length()), "_");
    }

    public void parallelNoteEvent(Note note)
    {
        insert(note.getMusicString().substring(1,
note.getMusicString().length()), "+");
    }

    private void insert(String string, String connector)
    {
        StringBuilder buddy = new StringBuilder();
        buddy.append(string);
        buddy.append(connector);
        buddy.append(getReturnPattern().getMusicString());
        getReturnPattern().setMusicString(buddy.toString());
    }
}
```

It is not always necessary to start a new class file when creating a subclass of PatternTransformer. In fact, you can create a really simple anonymous transformer within your application in very few lines. For example, suppose you simply want to change all instances of a `G5` note to a `C5` note in any given pattern. This is so easy that you don't even need to create a new class file for it. The example below shows you how to do it with an anonymous inner class.

```
// Change G5 to C5
PatternTransformer noteTransformer = new PatternTransformer() {
    public void noteEvent(Note note)
    {
        change(note);
    }
    public void parallelNoteEvent(Note note)
    {
        change(note);
    }
    public void sequentialNoteEvent(Note note)
    {
        change(note);
    }
    private void change(Note note)
    {
        if (note.getValue() == 67)   // 67 is a G5
        {
            note.setValue(60);       // 60 is a C5
        }
        getReturnPattern().addElement(note);
    }
};
Pattern newPattern = noteTransformer.transform(originalPattern);
```

How to Use a PatternTransformer

Once you have a PatternTransformer, it is incredibly easy to use. You simply pass your pattern to its `transform()` method, and you get a Pattern back. The original pattern remains intact.

```
    YourPatternTransformer pt = new YourPatternTransformer();
    Pattern newPattern = pt.transform(originalPattern);
```

PatternTransformers In Action

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 uses two PatternTransformers. It takes advantage of JFugue’s ReversePatternTransformer to reverse the sequence of notes. It also uses an anonymous inner class “octave transformer” to lower the octave of all of the notes in one of the voices (note that lowering notes by one octave means subtracting 12 (not 8) from each note’s value). You may listen to the Crab Canon at http://www.jfugue.org.

```
// One voice of Bach’s Crab Canon
Pattern canon = new Pattern("D5h E5h A5h Bb5h C#5h Rq A5q "+
               "A5q Ab5h G5q G5q F#5h F5q F5q E5q Eb5q D5q "+
               "C#5q A3q 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");

// Create a new pattern that is the reverse of the first pattern
ReversePatternTransformer rpt = new ReversePatternTransformer();
Pattern reverseCanon = rpt.transform(canon);

// Lower the octaves of the reversed pattern
PatternTransformer octaveTransformer = new PatternTransformer() {
    public void noteEvent(Note note)
    {
        byte currentValue = note.getValue();
        note.setValue((byte)(currentValue - 12));
        getReturnPattern().addElement(note);
    }
};
Pattern octaveCanon = octaveTransformer.transform(reverseCanon);

// Combine the two patterns
Pattern pattern = new Pattern("T[VIVACE]");
pattern.add("V0 " + canon.getMusicString());
pattern.add("V1 " + octaveCanon.getMusicString());

// Play Bach's Crab Canon
Player player = new Player();
player.play(pattern);
```

Using a ParserListener to Analyze a Pattern

A ParserListener can be used to analyze a pattern as well. This is useful if you want to derive some insight into a pattern. Here are several examples:

- Get a list of instruments used in the pattern
- Extract all musical events that belong to a specific voice
- Find out what notes are used in the pattern
- Calculate the number of times each note is played
- Compute the duration of a single voice

For example, let's create a ParserListener that lists the instruments used in a pattern. This would be useful if we wanted to load a new set of sounds from a high-quality soundbank, but we want to make sure that we only load those instruments that need to be loaded, because high-quality sounds use a lot of memory.

We'll create a class that is a ParserListener, but we'll take advantage of the ParserListenerAdapter. This class implements ParserListener and provides implementations for each of the callbacks that simply do nothing. We can override the specific callbacks that we're interested in—and in this case, we're interested in the `instrumentEvent` callback.
The example below extends ParserListenerAdapter and overrides the `instrumentEvent` callback. When an instrument event comes in, the callback obtains the value of the instrument and, if that value doesn't already exist in a list of instruments, the callback adds the value to the list. That list of instruments is returned to the program that calls `getInstrumentsUsed(Pattern pattern)`.

```
public class GetInstrumentsUsedTool extends ParserListenerAdapter
{
    private List<Instrument> instruments;

    public GetInstrumentsUsedTool()
    {
        instruments = new ArrayList<Instrument>();
    }

    @Override
    public void instrumentEvent(Instrument instrument)
    {
        if (!instruments.contains(instrument)) {
            instruments.add(instrument);
        }
    }

    public List<Instrument> getInstrumentsUsed(Pattern pattern)
    {
        MusicStringParser parser = new MusicStringParser();
        parser.addParserListener(this);
        parser.parse(pattern);
        return instruments;
    }
}
```

To use this class, a program creates an instance of the `GetInstrumentsUsedTool` class, and calls its `getInstrumentsUsed()` method on the pattern of interest.

If you're interested in creating a ParserListener that will handle all musical events the same way, you can use the `CollatedParserListener` class. The CollatedParserListener is a ParserListener that takes all of the callbacks and funnels them to a single abstract method, `jfugueEvent(JFugueElement element)`. One example of how this is used is in the MusicStringParser, which uses a CollatedParserListener to handle the verification of events that are created from parsed MusicStrings.

Working with MIDI Patterns

You can convert MIDI files to JFugue Patterns, which also means that you can perform transformations and analyses on existing MIDI music. For more information, see Chapter 5.

4

The JFugue Player

Once you create MusicStrings or Patterns, you'll want to play them! This chapter discusses the various ways in which music may be played, and introduces some novel and interesting things you can do with a player.

Playing Music

You've already learned the most used method of the Player class: `player.play()`. In fact, this is such a central part of JFugue – after all, you want to play music! – that there are a number of variations of the method call, as follows:

- `play(String musicString)` – plays the given MusicString
- `play(Pattern pattern)` – plays the given Pattern
- `play(Rhythm rhythm)` – plays the given Rhythm (explained in Chapter 7), without needing to explicitly get a Pattern from the Rhythm

The Player class also has some methods to play MIDI sequences directly from a URL or File. These play methods do not parse or interpret the MIDI sequences, nor do they do any conversion. They simply obtain a Sequence from the MIDI source and play it (this may sound basic, but JFugue does a lot of opening, closing, and error catching behind the scenes).

- `playMidiDirectly(URL url)` – Obtains a MIDI sequence from the URL, and plays it directly

- `playMidiDirectly(File file)` – Opens the given file, obtains a MIDI Sequence, and plays it directly

The Player class also has methods that let you save music as a MIDI file, and load music from a MIDI file *and* convert it to a JFugue Pattern. These are discussed in more detail in Chapter 5, but for completeness, they are listed here:

- `saveMidi(String musicString, File file)` – saves the music in the MusicString to the given file.
- `saveMidi(Pattern pattern, File file)` – saves the music in the Pattern to the given file.
- `loadMidi(File file)` – Opens the given MIDI file and reads in the sequence, converts the sequence to a MusicString, wraps the MusicString in a Pattern, and returns the Pattern.

Starting a Player with a Known Sequencer or Synthesizer

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.

The no-argument `Player` constructor obtains a sequencer using Java's `MidiSystem.getSequencer()` call. There are two other constructors for the Player class that will use the sequencer or synthesizer you provide:

- `Player(Sequencer sequencer)` – creates a new player, and associates it with the given sequencer
- `Player(Synthesizer synth)` – creates a new player. Obtains a sequencer connected to the given synthesizer, and associates that sequencer with the player.

In addition, the Player class has a static method,
`Player.getSequencerConnectedToSynthesizer(Synthesizer synth)`,
which will create a new Sequencer instance and connect it to the given synthesizer by setting the sequencer's transmitter's receiver to the synth's receiver.

Pausing, Rewinding, and Forwarding the Player

The Player class supports pausing, stopping, resuming, and changing the current position of the sequencer. This is made possible by leveraging methods on Java's Sequencer class, which supports repositioning the millisecond position of a sequence.

JFugue's `play()` methods do not return until a sequence has finished playing. If you have a single-threaded application, the application will wait until the `play()` method is done playing its sequence before continuing. Therefore, if you would like to control the player, you'll need to do it in a separate thread.

The methods on the `Player` class that support pausing, repositioning, and so on, are as follows:

- `pause()` – pause the playback of the sequence. Calls to the `play()` method continue to wait for the `play()` method to finish, which can't happen until the player is resumed and finishes playing its song, or it is stopped.
- `resume()` – resumes a paused player
- `stop()` – stops the player. The `play()` method will return immediately.
- `jumpTo(long microseconds)` – repositions the microsecond position on the sequencer to the desired value. Note that this is expressed in *microseconds*, not *milliseconds*!

Additionally, there are methods available to inspect the state of the player:

- `isStarted()` – returns true if the player has started; if the player has played a sequence which has been completed or stopped, `isStarted()` returns false
- `isPaused()` – returns true is `pause()` has been called on the player, and playback has not resumed
- `isPlaying()` – returns true if the player's sequencer is running – this is a passthrough to `sequencer.isRunning()`
- `isFinished()` – returns true if the player's sequencer has finished playing a sequence, or if the `stop()` has been called
- `getSequencePosition()` – this is a passthrough to the player's sequencer's `getMicrosecondPosition()` method

The Streaming Player

You can send music to JFugue in real-time, instead of constructing a MusicString and waiting for the `player.play()` method to return.

So far, you have learned how to work with JFugue in the following way:

1. You call `player.play()` with a MusicString or Pattern
2. JFugue converts your musical instructions into a MIDI sequence
3. The MIDI sequence is played
4. When you're done, call `close()` on the Player.

This arrangement supports cases in which you specify, either manually or through an algorithm, the content of your music. You can fully specify the music, and then have it played.

However, you may find cases in which you want to add new music at runtime. For example, suppose you're making a virtual musical instrument – a graphical piano, for example, that lets one press the keys and hear music immediately. For this, you need a Player that can play music in real-time. That is the motivation behind the Streaming Player.

The Streaming Player works in the following way:

1. You create a `StreamingPlayer()` and add fragments or tokens of a MusicString using the `stream(String musicString)` or `stream(Pattern pattern)` method on the Streaming Player
2. JFugue converts your musical instructions into MIDI commands
3. The MIDI commands are played as they arrive
4. When you're done, call `close()` on the StreamingPlayer.

Behind the scenes, JFugue renders music slightly differently compared to the traditional Player. The MusicStringParser is still responsible for parsing the new tokens that are streamed to the StreamingPlayer, but instead of a MidiRenderer listening for the resulting musical events, a StreamingMidiRenderer is listening. Instead of creating a sequence, the StreamingMidiEventManager is used. This sends MIDI events directly to the MidiChannels associated with a Synthesizer. (This also means that you won't have a sequence object at the end that you can save, because a sequence is never created.)

Like the Player, the Streaming Player can be created with a sequencer or synthesizer that you may have already set up. The constructors for the `StreamingPlayer` class are as follows:

- `StreamingPlayer()` – creates a new streaming player, which will get a Sequencer from the MidiSystem
- `StreamingPlayer(Sequencer sequencer)` – creates a new streaming player, and associates it with the given sequencer
- `StreamingPlayer(Synthesizer synth)` – creates a new streaming player. Obtains a sequencer connected to the given synthesizer, and associates that sequencer with the player.

When the Stream Doesn't Flow

If you don't hear any music from the Streaming Player, try changing the Synthesizer that you're using. For example, I wasn't hearing any music when I was using `com.sun.media.sound.MixerSynth` (you can get the name of your synthesizer by calling `synthesizer.toString()`). To change the synthesizer, I linked to the gervill.jar library and got a synthesizer of `com.sun.media.sound.SoftSynthesizer.` This worked – with a different synthesizer, I could now hear streaming music.

How to Simulate a Pipe Organ

Suppose you want to write a program that lets the user press keys to make various notes sound; like a pipe organ, the sound starts when the key is pressed, and stops when the key is released. It is not immediately obvious how this might work, but with a little ingenuity, it's not hard at all.

The issue is that you don't know how long of a duration to specify the new note, because you can't predict when the user will release the key. Fortunately, with the tie commands in the MusicString, you can indicate the start of a note separately from the end of a note.

Here's an example. Create an class that extends JFrame and implements KeyListener, and implement the KeyListener methods in this way (and don't forget to call `addKeyListener(this)`):

```
public void keyPressed(KeyEvent e)
{
    String noteString = {figure out the note} + "o-";
    player.stream(noteString);
}

public void keyReleased(KeyEvent e)
{
    String noteString = {figure out the note} + "-o";
    player.stream(noteString);
}

public void keyTyped(KeyEvent e)
{
    // Do nothing
}
```

When the user presses a key, a MusicString like `C3o-` is streamed to the Streaming Player. Recall that `o-` is a duration that says, "one-hundred-twenty-eighth note, beginning of a tie". This note will start, and it won't end until another note comes along to complete the tie. The tie will be completed when the user releases the key, which will stream a fragment like `C3-o`, which will end the tie with a 128th note. The 128th notes (the shortest durations available to JFugue) provide a way to start and stop the playing of a note, while adding as little additional duration to the note as possible. For example, if we ended with a quarter duration, the music would seem to continue after the user has released the key.

Note that this will only work with instruments that won't decay naturally without an end note. For example, the sound created by the MIDI piano instrument starts to decay after it has sounded, so the note will sound like it is complete before the end note is delivered (which may be correct behavior, if that's the effect your going for). Instruments like the rock organ or flute will continue to sound at the initial volume level until an end event is sent. Of course, this can be annoying if the program crashes before the end note can be sent, because the note will still sound!

Stop the music!

If you happen to create a program that ends or crashes before an ending tie note is delivered, the music will continue to sound. To stop all notes on all channels instantly, use the static method `Player.allNotesOff(Synthesizer synth)`.

Throttling the Delivery of New Fragments

The Streaming Player will not wait until a fragment is done playing before allowing you to stream more fragments. That means that if you were to write a program that, from within a loop, sends new MusicStrings or Patterns to `stream()` without pausing, you probably won't hear any music, because all of the messages will be delivered nearly instantaneously (or, at least as fast as the program can run).

If you decide that you want to loop through a number of fragments and add each of them individually to the StreamingPlayer, you'll need to throttle your delivery of the events. You can do this by sleeping for the duration of your fragment multiplied by `TimeFactor.QUARTER_DURATIONS_IN_WHOLE`.

The Anticipator: Know Upcoming Events Before They Happen

Suppose you'd like to know what musical events are going to sound in the future. There are a number of potential applications for this; the most obvious one is graphical animation based on musical events.

You know how musicians in a live orchestra get ready for the music they're about to play – the violinists bring their violins to their chins, the flutists bring their flutes to their lips, the percussionists make their way to the xylophone or the tubular bells, and pick up a mallet? They know the music that's coming next: they are anticipating the next musical events.

Now, suppose you are creating a graphical animation that will also need to prepare for upcoming musical events. For fun, let's say that your animation will be composed of cannons that launch balls, and the balls follow the laws of physics as they fly through the air and eventually land in a pool of water; various locations in the pool correspond to different notes. And, for fun, let's say that this animation is meant to be displayed when playing… are you ready for it?... canons[4]!

Here's the tricky part: The cannon has to be angled in such a way that a ball launched from it will arc correctly through the air and land in exactly the right place in the pool to give the illusion that it has generated a note. As soon as the ball hits the water, the note will sound. But the cannon needs to know what note that will be, so it can aim the ball correctly!

Let's say that the time it takes for the ball to be launched from the cannon and hit the water is a constant 300 milliseconds[5], and the time it takes for the cannon to change its angle is 10 milliseconds (for simplicity, we'll assume that movement to and from any angle takes the same 10 milliseconds, regardless if the change is 1 degree or 90 degrees). So, we need to know what note will occur 400 milliseconds in the future. Can we know this? With JFugue, as always, the answer is, "Of course!"

To make this work, we need JFugue's Anticipator class. The Anticipator is used to deliver musical events before they happen. It works by launching a separate thread, and processing the same Pattern or MusicString that the Player will eventually play. But instead of causing musical events to be fired, the Anticipator will deliver the events to whatever listener you have defined.

Here's how the Anticipator works:

1. Create an instance of the Anticipator class
2. Add a ParserListener to the Anticipator. For the sake of simplicity, JFugue provides an EasyAnticipatorListener that you can use as the ParserListener (more on that below)
3. Call `player.play(Anticipator anticipator, Pattern pattern, long msOffset)`, passing the anticipator, the pattern, and the millisecond delay indicating how much earlier you'd like the Anticipator listener to know about the upcoming event.

Here's an example:

```
Player player = new Player();
Pattern pattern = new Pattern("your music");

Anticipator anticipator = new Anticipator();
```

[4] For a beautiful rendition of Pachelbel's "Canon in D" created using JFugue, see http://www.jfugue.org/exchange.html

[5] Please disregard any facts of physics that may be bent for the sake of this example. This is a book on music programming.

```
anticipator.addParserListener(new EasyAnticipatorListener() {
    public void extendedNoteEvent(Voice voice, Instrument
instrument, Note note)
    {
        System.out.println("Voice: "+voice.getVoice());
        System.out.println("Instrument: " +
           instrument.getInstrument());
        System.out.println("Note "+note.getValue()+", duration "
           + note.getDuration());
    }
} );

player.play(anticipator, pattern, 400);
```

The example above uses the EasyAnticipatorListener, which is a ParserListener that does some extra things behind the scenes. It keeps track of voice and instrument changes as they occur, and takes the three different types of note events (regular note event, parallel note event, and sequential note event), and combines those into a single callback, `extendedNoteEvent()`, which is only triggered when one of the three note events is called. It comes with the voice and instrument in which the note will be played.

Since the Anticipator isn't actually playing music, it needs to simulate the delays that would occur as notes are played. For this, the Anticipator relies in the `TimeFactor.sortAndDeliverMidiMessages()` static method. The job of this method is to arrange the MIDI messages according to when in time they should be sent (MIDI messages do not necessarily follow a linear sequence), and send off the events at the correct time.