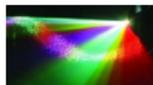Chapter 12
Described Auditory Content – Music

The Design and Implementation of Multimedia Software

David Bernstein

Jones and Bartlett Publishers

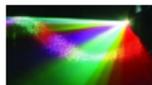www.jbpub.com

About this Chapter

- Described Auditory Content:
    - Described speech, described sounds, and described music

- This Chapter – Described Music:
    - 'Name' frequencies.

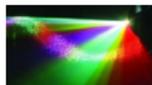   - Describe the duration of individual sounds.

    - Describe the 'voice' or 'instrument'.

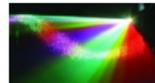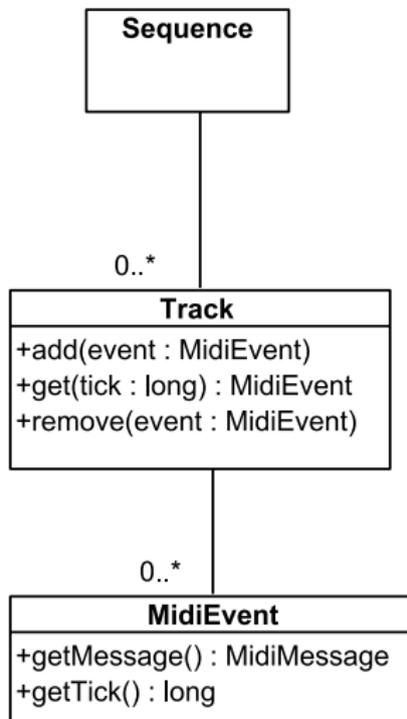   - Describe the loudness of those sounds.

Musical Instrument Digital Interface (MIDI)

- Original Purpose:

  A protocol for passing musical events between electronic instruments and sequencers.

- Our Interest:

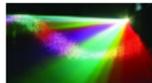 Sending messages to a sound card.

The `Sequence`, `Track` and `MidiEvent` Classes

Reading MIDI Events from a File

1. Create a `Sequence` from a `File` or an `InputStream`.

2. Create a `Sequencer`.

3. Open the `Sequencer`.

4. Associate the `Sequence` with the `Sequencer`.
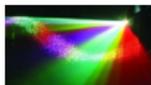
5. Start the `Sequencer`.

MidiPlayer

```java
import java.io.*;
import javax.sound.midi.*;

public class MidiPlayer
{
    public static void main(String[] args) throws Exception
    {
        Sequencer         sequencer;
        Sequence          seq;

        seq = MidiSystem.getSequence(new File(args[0]));

        sequencer = MidiSystem.getSequencer();
        sequencer.open();

        sequencer.setSequence(seq);
        sequencer.start();
    }
}
```
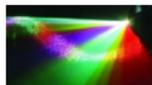
Requirements

F12.1 Encapsulate a description of music.
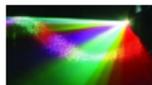
F12.2 Present/render this description.

Ways Synthesizers Can Generate Signals

- Using a Soundbank:

  The synthesizer has a database of 'sounds' (e.g., samplings of different instruments producing different tones) that it uses to produce the electrical signal.

- Using Waves:

 The synthesizer constructs the electrical signals from continuous waves.

Rendering in Java

- **Synthesizer**:

    Controls a set (typically with 16 members) of `MidiChannel` objects that encapsulate 'voices'.

- Setup:

   1. Obtain a `Synthesizer` object by calling the `MidiSystem.getSynthesizer()` method.

    2. Get a `Soundbank` object (e.g., by calling the `Synthesizer` object's `getDefaultSoundBank()` method).

    3. Call the `Synthesizer` object's `loadAllInstruments()` method (passing it the `Soundbank` object.

    4. Get the `MidiChannel` objects by calling the `Synthesizer` object's `getChannels()` method.
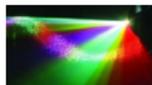
Renedering Notes – Alternative 1

1. Obtain a `Receiver` object by calling the `Synthesizer` object's `getReceiver()` method.

2. Construct a `ShortMessage` object that turns the note on.

3. Call the `Receiver` object's `send()` method (passing it the `ShortMessage` object).

4. Wait the appropriate amount of time.

5. Construct a `ShortMessage` object that turns the note off.

6. Call the `Receiver` object's `send()` method (passing it the `ShortMessage` object).
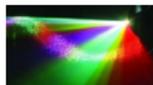
Renedering Notes – Alternative 2

1. Call the `MidiChannel` object's `noteOn()` method.

2. Wait the appropriate amount of time.

3. Call the `MidiChannel` object's `noteOff()` method.
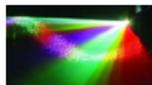
Evaluating the Alternatives

Which is better?

Evaluating the Alternatives

The second method is used here because it is simpler and provides all of the necessary functionality.

Naming Frequencies

- Modern Music Notation:

  The frequencies in a single octave are denoted by the letters A-G.

  Letters can be followed by either a sharp indicator (i.e., a ♯ character) which denotes a half-step or half-tone up, or flat indicator (i.e., a ♭ character) which denotes a half-step or half-tone down.

 A complete octave contains A, A♯, B, C, C♯, D, D♯, E, F, F♯, G, and G♯.

- A Common Tuning:

  "A" is the name commonly given to the frequencies 55Hz, 110Hz, 220Hz, 440Hz, 880Hz, 1760Hz, etc.
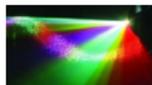
Describing Durations

- Durations are commonly given as multiples/fractions of a base duration, or *beat*.

- Common durations are whole notes, half notes, quarter notes, eighth notes and sixteenth notes.

- A standard duration can be 'dotted' to indicate that it should be increased by 50%.

- *Triplets* are notes that are grouped in sets of three (i.e., the three notes evenly divide an integral number of beats).
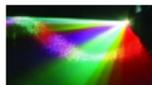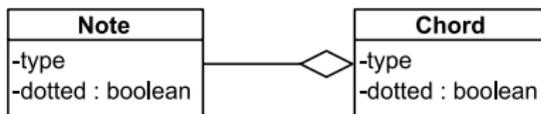
Alternative 1
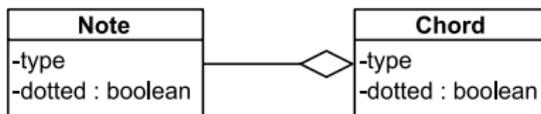
What are the shortcomings?

Alternative 1

It will probably result in code duplication.
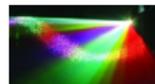
Alternative 2
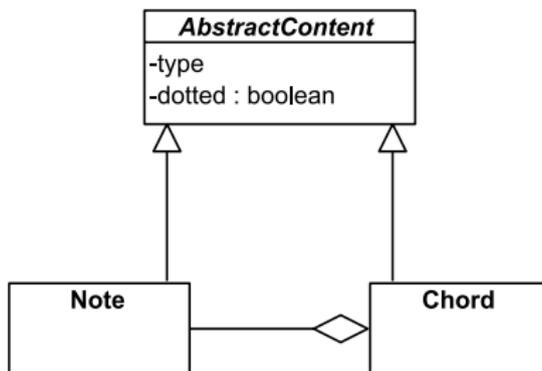
What are the shortcomings?
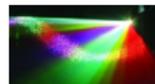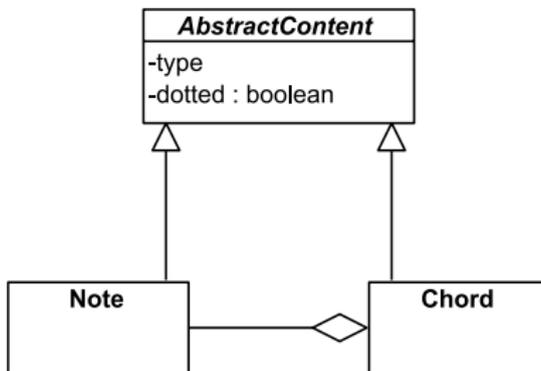
Alternative 2

Makes it difficult to add content that is not a specialization of the `AbstractContent` class.
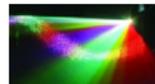
Alternative 3
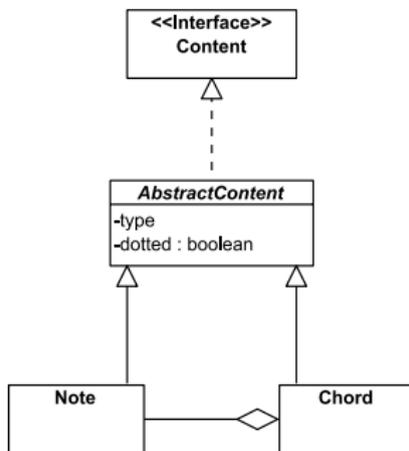
What are the advantages?
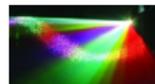
Alternative 3
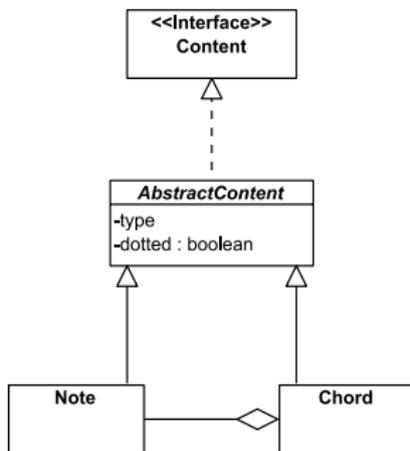
One can create other classes that implement the `Content` interface without having to specialize the `AbstractContent` class (e.g., a `BrokenChord` class).
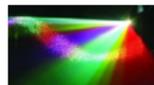
Alternative 4 – Composite Pattern
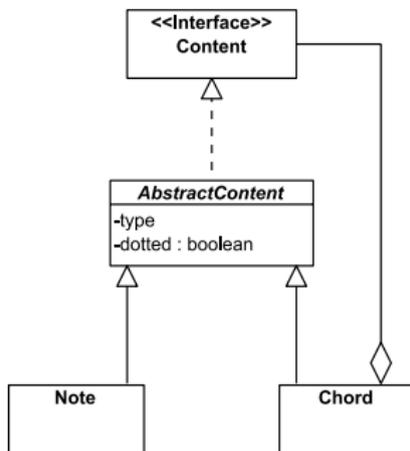
What are the shortcomings?

Alternative 4 – Composite Pattern
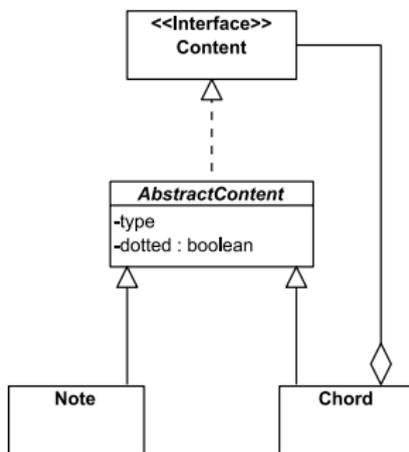
It is not consistent with the way the term "chord" is normally used in music theory.

Content

```
package auditory.described;

import javax.sound.midi.*;

public interface Content
{
    public abstract int  getType();

    public abstract boolean isDotted();

    public abstract void render(MidiChannel channel);

    public abstract void setAudible(boolean audible);
}
```
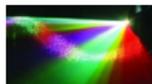
AbstractContent – Structure

```
package auditory.described;

import javax.sound.midi.*;

public abstract class       AbstractContent
              implements Content
{
    protected boolean                   audible, dotted, playing;
    protected int                       type;

    public AbstractContent()
    {
        this(1, false);
    }

    public AbstractContent(int type, boolean dotted)
    {
        this.type    = type;
        this.dotted  = dotted;
        this.audible = false;
        this.playing = false;
    }
}
```
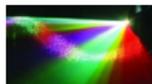
AbstractContent – Getters

```
public int getType()
{
   return type;
}

public boolean isDotted()
{
   return dotted;
}
```

AbstractContent – Setters

```
public void setAudible(boolean audible)
{
   this.audible = audible;
}

protected void setDotted(boolean dotted)
{
   this.dotted = dotted;
}

protected void setType(int type)
{
   this.type = type;
}
```
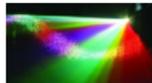
AbstractContent – Rendering

```
public void render(MidiChannel channel)
{
   if      ( audible && !playing)
   {
      playing = true;
      startPlaying(channel);
   }
   else if (!audible &&  playing)
   {
      playing = false;
      stopPlaying(channel);
   }
}
protected abstract void startPlaying(MidiChannel channel);

protected abstract void stopPlaying(MidiChannel channel);
```
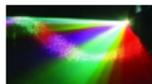
Note – Structure

```
package auditory.described;

import javax.sound.midi.*;

public class   Note
      extends AbstractContent
{
    private boolean               sharp;
    private char                  pitch;
    private int                   midiNumber;

    public Note()
    {
        this('C', false, 0, 1, false);
    }

    public Note(char pitch,   boolean sharp, int octave,
              int type, boolean dotted)
    {
        super(type, dotted);

        this.pitch = Character.toUpperCase(pitch);
        this.sharp = sharp;
        midiNumber = MidiCalculator.numberFor(pitch, sharp, octave);
    }
}
```

`Note` – Rendering

```
protected void startPlaying(MidiChannel channel)
{
    channel.noteOn(midiNumber, 127);
}

protected void stopPlaying(MidiChannel channel)
{
    channel.noteOff(midiNumber, 127);
}
```
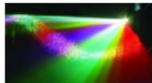
MidiCalculator

```
package auditory.described;

public class MidiCalculator
{
    public static int numberFor(char     pitch,
                                boolean  sharp,
                                int      octave)
    {
        int     midiBase, midiNumber;


        // Handle special cases (B sharp and E sharp)
        if ((pitch == 'B') && sharp)
        {
            pitch = 'C';
            sharp = false;
        }
        else if ((pitch == 'E') && sharp)
        {
            pitch = 'F';
            sharp = false;
        }

        // Calculate the MIDI value
        midiBase = 60;
        if       (pitch == 'A') midiNumber = midiBase +  9;
        else if  (pitch == 'B') midiNumber = midiBase + 11;
        else if  (pitch == 'C') midiNumber = midiBase +  0;
```
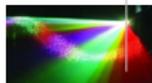
MidiCalculator (cont.)

```
        else if (pitch == 'D') midiNumber = midiBase +  2;
        else if (pitch == 'E') midiNumber = midiBase +  4;
        else if (pitch == 'F') midiNumber = midiBase +  5;
        else if (pitch == 'G') midiNumber = midiBase +  7;
        else                    midiNumber = -1;  // Rest

        if (sharp) midiNumber = midiNumber + 1;

        midiNumber = midiNumber + (octave * 12);

        return midiNumber;
    }
}
```
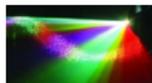
Chord – Structure

```
package auditory.described;

import java.util.*;
import javax.sound.midi.*;

public class        Chord
      extends      AbstractContent
{
    private ArrayList<Note>        notes;

    public Chord()
    {
        this(1, false);
    }

    public Chord(int type, boolean dotted)
    {
        super(type, dotted);

        notes = new ArrayList<Note>();
    }

    public void addNote(Note note)
    {
        notes.add(note);
    }
}
```
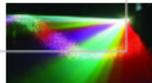
Chord – startPlaying()

```
protected void startPlaying(MidiChannel channel)
{
    Iterator<Note>      i;
    Note                note;

    i = notes.iterator();
    while (i.hasNext())
    {
        note = i.next();
        if (note != null)
        {
            note.setType(type);
            note.setDotted(dotted);
            note.startPlaying(channel);
        }
    }
}
```
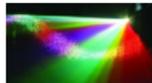
Chord – stopPlaying()

```
protected void stopPlaying(MidiChannel channel)
{
    Iterator<Note>     i;
    Note               note;

    i = notes.iterator();
    while (i.hasNext())
    {
        note = i.next();
        if (note != null)
        {
            note.stopPlaying(channel);
        }
    }
}
```
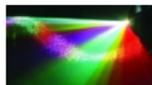
Music Notation

Music notation consists of one or more *staffs* (a set of five horizontal lines) that contain multiple measures (delimited by *bar lines*), each of which contains multiple notes (that indicate both the pitch and the duration). The end of a song is usually indicated with a double bar line.
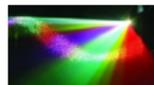
Eight Measures of Beethoven's "Ode to Joy"
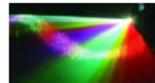
A More Convenient Notation

A comma delimited `String` with a field for the 'name' including an optional `'#'` character, the octave (with 0 being the octave that contains middle C), the type of the note (i.e., 1 for whole, 2 for half, 4 for quarter, etc) which includes an optional `'.'` character.

Eight Measures of Beethoven's 'Ode to Joy"

```
            .           .           .
            .           .           .
            .           .           .
F#,-1,4     D ,-1,4     F#,-1,4     D ,-1,4
F#,-1,4     D ,-1,4     F#,-1,4     D ,-1,4
G ,-1,4     E ,-1,4     G ,-1,4     E ,-1,4
A ,-1,4     F#,-1,4     A ,-1,4     F#,-1,4

A ,-1,4     F#,-1,4.    A ,-1,4     E ,-1,4.
G ,-1,4     E ,-1,8     G ,-1,4     D ,-1,8
F#,-1,4     E ,-1,2     F#,-1,4     D ,-1,4
E ,-1,4                 E ,-1,4     R , 0,4
.           .           .
.           .           .
.           .           .
```
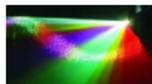
NoteFactory

```
package auditory.described;

import java.util.*;

null
public class NoteFactory
{
    public static Note parseNote(String s)
    {
        boolean          dotted, sharp;
        char             pitch, sharpChar;
        int              duration, octave, octaveEnd;
        Note             theNote;
        String           durationString, octaveString, token;
        StringTokenizer  st;

        st = new StringTokenizer(s, ", ");

        try
        {
            token = st.nextToken();

            // Determine the pitch
            pitch = token.charAt(0);

            // Determine if this is a sharp or a natural
            sharp = false;
            if (token.length() == 2)
```
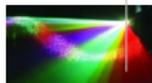
NoteFactory (cont.)

```
        {
           sharpChar = token.charAt(1);
           if (sharpChar == '#') sharp = true; // ASCII 35
        }

        // Detemine the octave (relative to middle C)
        octaveString = st.nextToken();
        octave       = Integer.parseInt(octaveString);

        // Determine the duration (which has an arbitrary length)
        dotted         = false;
        durationString = st.nextToken();
        if (durationString.endsWith(".")) dotted = true;
        duration = (int)Double.parseDouble(durationString);

        // Construct a new Note
        theNote = new Note(pitch, sharp, octave, duration, dotted);
     }
     catch (NoSuchElementException nsee)
     {
        theNote = null;
     }

     return theNote;
   }
}
```

Scales

- An (ascending) *scale* is a sequence of notes that starts and ends on the same note, moving up in a consistent way.

- A *major scale* is a scale that uses the T-T-S-T-T-T-S pattern when ascending (where T denotes a tone or whole step and S denotes a semi-tone or half step).

- A *minor scale* is a scale that uses the T-S-T-T-S-T-T pattern when ascending.
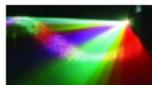
Scales – The C-Major Scale

```
C    D    E    F    G    A    B    C
  T    T    S    T    T    T    S
```
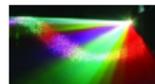
Transposition

- A song that uses the notes from a particular scale is said to be in the *key* of that scale.

- Transposition is an operation that changes a song from one key to another.

- Since the representation used here does not include a *key signature*, that is, all notes include explicit sharps/flats, transposition can be performed directly on the MIDI numbers.

What's Left?

- The `Audition` components of the conceptual model.

- The `AuditionView` component of the conceptual model

Organizing Notes and Chords

- Notes are commonly grouped into durations of equal length, called *measures*.

- The *time signature* indicates how many beats are in each measure (in the 'numerator') and which type of note gets a whole beat (in the 'denominator').

- A collection of notes/chords ordered over time that is intended to be played by a single instrument (or sung by a single voice) is often called a *part*.

- A collection of parts (for different instruments/voices) is referred to as a *score*.
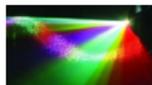
Alternative 1

- Aproach:

  Each `Part` has it's own `Metronome` object and renders its content (independently) in its own thread of execution.
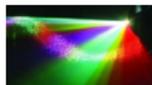
- Shortcomings:

  What are the shortcomings?

Alternative 1

- Aproach:

    Each `Part` has it's own `Metronome` object and renders its content (independently) in its own thread of execution.

- Shortcomings:

    The different `Part` objects may not stay synchronized since the timing of `handleTick()` messages is not precise.
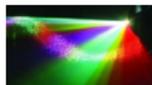
Alternative 2

- Approach:

  Have a single `Metronome` object that synchronizes the `Part` objects.

- `handleTick()` Method:

  Determines which `Note`/`Chord` objects needs to be rendered.
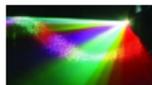
Alternative 2.1

- Approach:

    Include an attribute for the instrument in `Part`.

- Shortcomings:

    What are the shortcomings?
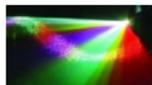
Alternative 2.1

- Approach:

  Include an attribute for the instrument in `Part`.

- Shortcomings:

  One can not have the same `Part` object rendered using different instruments.
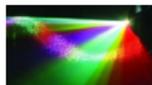
Alternative 2.2

- Approach:

  A `Part` object can be told what instrument to use when its
  `render()` method is called.

- Advantage:

  Adds flexibility at no 'cost'.
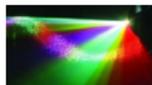
The Design
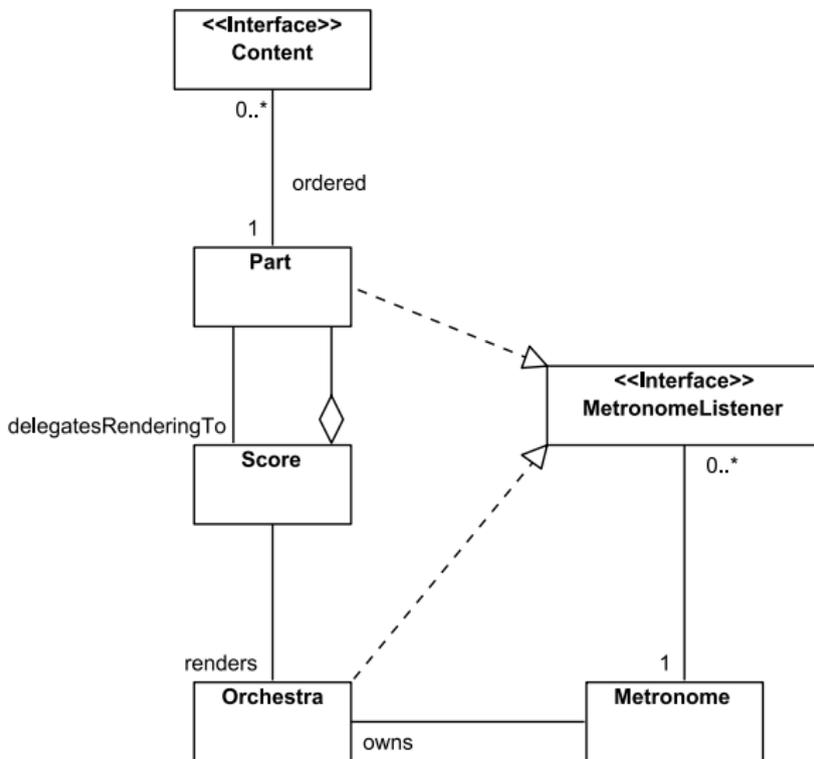
Part – Structure

```
package auditory.described;

import java.util.*;
import javax.sound.midi.*;

import event.*;

public class Part implements MetronomeListener
{
    private ArrayList<Content>     sounds;
    private double                 timeSignatureDenominator,
                                   timeSignatureNumerator;
    private int                    millisPerMeasure, stopTime;
    public Part()
    {
        sounds       = new ArrayList<Content>();
    }
}
```

Part – add()

```
public void add(Content c)
{
    if (c != null) sounds.add(c);
}
```

Part – Setters

```
public void setTempo(int millisPerMeasure)
{
    this.millisPerMeasure = millisPerMeasure;
}


public void setTimeSignature(int numerator, int denominator)
{
    this.timeSignatureNumerator   = numerator;
    this.timeSignatureDenominator = denominator;
}
```
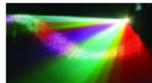
Part – Global Variables

```
    private Content             currentContent,
                                previousContent;
    private Iterator<Content>   iterator;
    private Metronome           metronome; // Not Owned
```
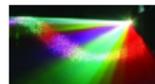
Part – upbeat()

```
public void upbeat(Metronome metronome)
{
   this.metronome   = metronome; // For later removal

   iterator         = sounds.iterator();
   currentContent   = null;
   previousContent  = null;
   stopTime         = -1;

   metronome.addListener(this);
}
```

Part – `handleTick()`

```
public void handleTick(int millis)
{
   double    beats, millisPerBeat, type;

   if (iterator == null)
      throw(new IllegalStateException("No upbeat()"));

   if (millis >= stopTime)
   {
      if (currentContent != null)
         currentContent.setAudible(false);

      if (iterator.hasNext())
      {
         previousContent = currentContent;
         currentContent  = iterator.next();

         // This calculation needn't really be done each iteration
         millisPerBeat = 1.0/(double)timeSignatureNumerator *
                         millisPerMeasure;

         beats         =  (1.0/(double)currentContent.getType()) *
                          (double)timeSignatureDenominator;

         if (currentContent.isDotted()) beats = beats * 1.5;

         stopTime = millis + (int)(beats * millisPerBeat);
```
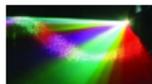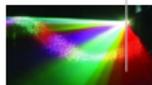
Part – handleTick() (cont.)

```
            currentContent.setAudible(true);
        }
        else
        {
            metronome.removeListener(this);
        }
    }
}
```
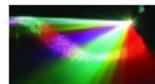
Part – render()

```
public void render(MidiChannel channel)
{
    if (previousContent != null)
        previousContent.render(channel);

    if (currentContent   != null)
        currentContent.render(channel);
}
```

PartFactory

```
package auditory.described;

import java.io.*;

public class PartFactory
{
    public static Part createPart(BufferedReader in)
                    throws IOException
    {
        Note                    note;
        Part                    part;
        String                  line;

        part = new Part();

        while ((line = in.readLine()) != null &&
               (!line.equals("X"))                    )
        {
            if (!line.equals(""))
            {
                note = NoteFactory.parseNote(line);
                if (note != null) part.add(note);
            }
        }

        return part;
    }
```
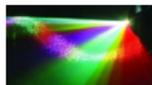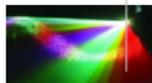
PartFactory (cont.)

```
    public static Part createPart(String filename)
                    throws IOException
    {
        BufferedReader            in;

        in = new BufferedReader(new FileReader(filename));

        return createPart(in);
    }
}
```

Score – Structure

```
package auditory.described;

import java.util.*;
import javax.sound.midi.*;

import event.*;

public class Score
{
    private Hashtable<Part, MidiChannel>        channelTable;
    private Hashtable<Part, String>             parts;
    private int                                 timeSignatureDenominator,
                                                timeSignatureNumerator,
                                                millisPerMeasure;
    public void addPart(Part part, String instrument)
    {
        parts.put(part, instrument);
    }

    public Enumeration<Part> getParts()
    {
        return parts.keys();
    }

    public String getInstrumentName(Part part)
    {
        return parts.get(part);
    }
```

Score – Structure (cont.)

```
    public void removePart(Part part)
    {
        parts.remove(part);
    }
}
```
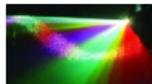
Score – Setters

```
public void setChannel(Part part, MidiChannel channel)
{
   channelTable.put(part, channel);
}

public void setTempo(int millisPerMeasure)
{
   this.millisPerMeasure = millisPerMeasure;
}

public void setTimeSignature(int numerator, int denominator)
{
   this.timeSignatureNumerator   = numerator;
   this.timeSignatureDenominator = denominator;
}
```
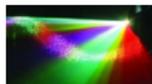
Score – upbeat()

```java
public void upbeat(Metronome metronome)
{
    Enumeration<Part>           e;
    Part                        part;

    e = parts.keys();
    while (e.hasMoreElements())
    {
        part        = e.nextElement();
        part.upbeat(metronome);
        part.setTimeSignature(timeSignatureNumerator,
                              timeSignatureDenominator);
        part.setTempo(millisPerMeasure);
    }
}
```
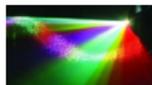
Score – `render()`
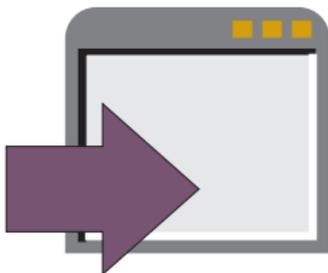
```
public void render()
{
    Enumeration<Part>           e;
    MidiChannel                 channel;
    Part                        part;

    e = parts.keys();
    while (e.hasMoreElements())
    {
        part        = e.nextElement();
        channel     = channelTable.get(part);
        part.render(channel);
    }
}
```

Orchestra –Demonstration

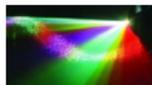## In examples/chapter:

Orchestra.html

```
java -cp Orchestra.jar OrchestraApplication frerejacques.txt

java -cp Orchestra.jar OrchestraApplication daytripper.txt

java -cp Orchestra.jar OrchestraApplication OdeToJoy.txt
```

Orchestra – Structure

```
package auditory.described;

import java.io.InputStream;
import java.net.URL;
import java.util.*;
import javax.sound.midi.*;

import event.*;
import io.*;

public class Orchestra implements MetronomeListener
{
    private Hashtable<String, Instrument>      instruments;
    private Metronome                          metronome;
    private MidiChannel[]                      channels;
    private Score                              score;
}
```
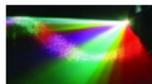
Orchestra – Constructor

```
public Orchestra(Score score) throws MidiUnavailableException
{
   this(score, new Metronome(10));
}


public Orchestra(Score score, Metronome metronome)
                          throws MidiUnavailableException
{
   Instrument[]   loaded;
   Soundbank      soundbank;
   Synthesizer    synthesizer;

   this.score     = score;
   this.metronome = metronome;
   metronome.addListener(this);

   instruments  = new Hashtable<String, Instrument>();

   synthesizer = MidiSystem.getSynthesizer();
   synthesizer.open();
   soundbank = synthesizer.getDefaultSoundbank();

   if (soundbank == null) soundbank = findSoundbank();

   synthesizer.loadAllInstruments(soundbank);

   channels     = synthesizer.getChannels();
```
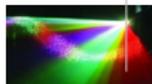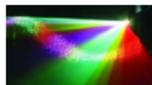
Orchestra – Constructor (cont.)

```
        loaded = synthesizer.getLoadedInstruments();
        for (int i=0; i<loaded.length; i++)
        {
            instruments.put(loaded[i].getName(), loaded[i]);
        }
    }
```

Orchestra – start()

```
public void start()
{
    Enumeration<Part>        e;
    Instrument               instrument;
    int                      i;
    MidiChannel              channel;
    String                   name;
    Patch                    patch;
    Part                     part;


    e = score.getParts();
    i = 0;

    while (e.hasMoreElements())
    {
        part       = e.nextElement();
        name       = score.getInstrumentName(part);
        instrument = instruments.get(name);

        // Have the channel use the appropriate instrument
        if (instrument == null)
        {
            channels[i].programChange(0, 0);
        }
        else
        {
            patch = instrument.getPatch();
```
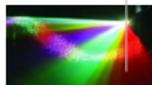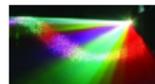
Orchestra – start() (cont.)

```
        channels[i].programChange(patch.getBank(),
                                  patch.getProgram());
    }
    score.setChannel(part, channels[i]);

    score.upbeat(metronome);
}


// Start the metronome
metronome.start();
}
```
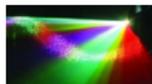
Orchestra – `handleTick()`

```
public void handleTick(int millis)
{
    score.render();

    if (metronome.getNumberOfListeners() == 1)
    {
        metronome.stop();
    }
}
```

ScoreFactory

```
package auditory.described;

import java.io.*;
import java.util.StringTokenizer;
import javax.sound.midi.MidiUnavailableException;

import io.*;

public class ScoreFactory
{
    private ResourceFinder        finder;

    public ScoreFactory()
    {
        finder = ResourceFinder.createInstance();
    }

    public ScoreFactory(ResourceFinder finder)
    {
        this.finder = finder;
    }

    public Score createScore(InputStream is)
                   throws IOException,
                          MidiUnavailableException
    {
        BufferedReader        in;
        int                   denominator, numerator, tempo;
```
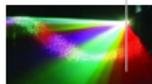
ScoreFactory (cont.)

```
Note                    note;
Part                    part;
Score                   score;
String                  line, voice;
StringTokenizer         st;

in = new BufferedReader(new InputStreamReader(is));

// Read the time signature and tempo
line        = in.readLine();
st          = new StringTokenizer(line, ",/");
numerator   = Integer.parseInt(st.nextToken());
denominator = Integer.parseInt(st.nextToken());
tempo       = Integer.parseInt(st.nextToken());

score = new Score();

score.setTimeSignature(numerator, denominator);
score.setTempo(tempo);

while ((voice = in.readLine()) != null)
{
    part = PartFactory.createPart(in);
    score.addPart(part, voice);
}

return score;
}
```
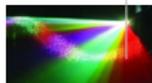
ScoreFactory (cont.)

```
    public Score createScore(String filename)
                    throws IOException,
                           MidiUnavailableException
    {
        InputStream          is;

        is = finder.findInputStream(filename);

        return createScore(is);
    }
}
```