Java Art Chapter 5. Program Sonification

Program sonification (also called auralization) is the transformation of an executing program into auditory information. I'm not talking about an application playing a sound clip, but the entire program *becoming* the clip or piece of music. The motivation for this unusual transformation is the same as for program visualization – as a way of better understanding what's happening inside code, to aid with its debugging and modification.

Music is inherently structured, hierarchical, and time-based, which suggests that it should be a good representation for structured and hierarchical code, whose execution is also time-based of course. Music offers many benefits as a notation, being both memorable and familiar. Even the simplest melody utilizes numerous attributes, such as sound location, loudness, pitch, sound quality (timbre), duration, rate of change, and ordering. These attributes can be variously matched to code attributes, such as data assignment, iteration and selection, and method calls and returns. Moving beyond a melody into more complex musical forms, lets us match recurring themes, orchestration, and multiple voices to programming ideas such as recursion, code reuse, and concurrency.

A drawback of music is the difficulty of representing quantitative information (e.g. that the integer x has the value 2), although qualitative statements are relatively easy to express (e.g. that the x value is increasing). One solution is lyrics: spoken (or sung) words to convey concrete details.

I'll be implementing program sonification using the tracer ideas discussed in the last two chapters (i.e. employing the Java Platform Debugger Architecture (JPDA), specifically its Java Debug Interface (JDI) API). The resulting system is shown in Figure 1.

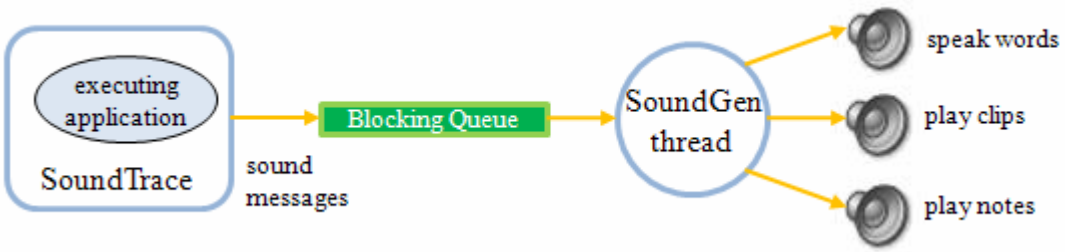

Figure 1. Sonification of a Java Application.

When a method is called in the monitored application, a word is spoken (an abbreviation of the method's name), when Java keywords are encountered in the code, musical notes are played, and when the program starts, ends, and when a method returns, sound clips are heard.

Sound generation is managed by the SoundGen thread, which reads messages from a queue filled by the tracer. The generator utilizes three sound APIs:

java.applet.AudioClip for playing clips, the MIDI classes in the Java Sound API for playing notes, and the FreeTTS speech synthesis system (http://freetts.sourceforge.net/), which is a partial implementation of the Java Speech API 1.0 (JSAPI, http://java.sun.com/products/java-media/speech/).

I'll start this chapter by explaining the three sound subsystems for playing clips, notes, and speaking. These are of general use outside of sonification; for example, the Speaker class (implemented with FreeTTS) can be used to pronounce any string, using a range of different voices.

The tracer code is not much different from that shown in previous chapters, but does illustrate two new ideas. Firstly, the queue between the tracer and sound generation thread deals with uneven execution speeds. The tracer generates messages at intervals of tens of milliseconds, but the generator is constrained to run much more slowly, waiting for each message to be sounded out before moving onto the next.

Secondly, the code shows the advantages of mixing static and dynamic analysis. Static analysis allows the Java keywords in the program text to be mapped to musical notes before the program begins execution (making it less costly to run). However, dynamic optimization of the message queue is also needed to reduce the amount of processing at runtime.

Although this chapter focuses on sonification, the implementation details are so similar to those used in the previous chapter on program visualization that it wouldn't be difficult to combine the two so that the application could be both visualized and heard at the same time.

More Information on Sonification

One of the best known sonification systems is CAITLIN (http://www.auralisation.org), which helps novice Pascal programmers with debugging. The auralization is based on point-of-interests (POIs), such as if-tests and loops. Each construct is represented by a musical motif, or a short recurring theme.

JListen auralizes Java program to report on events, which are specified by the programmer using a rule-based language called LSL (http://www.cs.purdue.edu/homes/apm/listen.html).

CodeSounding is a Java sonification library (http://www.codesounding.org/indexeng.html), which can add sound-generation methods to "if", "for", and other statements.

LYCAY is another Java library for the sonification of code, using a mix of parsing and execution strategies (http://lycay.sourceforge.net/textLetYourCodePlay.html).

The Wikipedia entry on sonification includes a good set of 'starter' references (http://en.wikipedia.org/wiki/Sonification). The *International Community for Auditory Display* (ICAD) is a more technical resource on sonification (http://www.icad.org/). Links to sonification-related research can be found at http://www.dcs.gla.ac.uk/~stephen/otherlinks.shtml and http://computing.unn.ac.uk/staff/cgpv1/caitlin/links.htm.

1. Playing a Clip

Java's AudioClip class offers a high-level means of loading and playing audio clips. It supports a large number of file formats, and multiple AudioClips can be played at the same time.

A major problem with AudioClip is that it doesn't report when a clip finishes. My SoundGen thread needs this information so that it can wait until the audio has finished before playing the next sound. There are several hacky work-arounds, such as calling sleep() for a period based on the audio file's byte size (which can be obtained via a File object).

Another issue is the lack of low-level access to the sound data (or the audio device it's playing on), to permit run-time effects like volume changing, panning between speakers, and echoing.

The best solution to these problems is to move to the Java Sound API, and utilize the javax.sound.sampled package for manipulating the audio clips. Unfortunately, this entails a lot of low-level programming involving sound formats (e.g. ALAW and ULAW), and an understanding of frame sizes, rates, and buffering. Once this high plateau of knowledge has been reached, calculating a clip's duration is a (long) one-liner:

```
double durationInSecs = clip.getBufferSize() /
  (clip.getFormat().getFrameSize()*clip.getFormat().getFrameRate());
```

For readers interested in this approach, I explain the details in an online chapter called "Chapters 7-10. Sound, Audio Effects, and Music Synthesis" at http://fivedots.coe.psu.ac.th/~ad/jg/. But in this chapter I'll take a much easier path: I'll assume that the user supplies the running time of a clip when requesting that it be played.

My ClipsPlayer class stores a collection of AudioClip objects in a HashMap whose keys are their filenames (minus the ".wav" extension).

```
// globals in the ClipsPlayer class
private final static String SOUND_DIR = "Sounds/";
                                  // directory holding the clips
private HashMap<String,AudioClip> clipsMap;
        // string is the filename (minus .wav) holding the clip


public ClipsPlayer()
{  clipsMap = new HashMap<String,AudioClip>();  }
```

When a clip is played, the play() method doesn't return until a specified delay time has passed.

```
public boolean play(String name, int delay)
// start playing the clip, and wait for delay ms before returning
{
  if (!clipsMap.containsKey(name))
    addClip(name);

  AudioClip clip = clipsMap.get(name);
```



```
  if (clip == null) {
    System.out.println("No clip found for " + name);
    return false;
  }
  clip.play();
  wait(delay);
  return true;
}  // end of play()


private boolean addClip(String name)
// store the AudioClip object for name in the hashmap
{
  AudioClip clip = loadClip(name);
  if (clip != null) {
    clipsMap.put(name, clip);
    System.out.println("Loaded clip for " + name);
    return true;
  }
  return false;
}  // end of addClip()


private AudioClip loadClip(String name)
// load name.wav from SOUND_DIR
{
  String fnm = SOUND_DIR + name + ".wav";
  AudioClip clip = null;
  try {
    clip = Applet.newAudioClip( getClass().getResource(fnm) );
  }
  catch (Exception e) {
    System.out.println("Could not load " + fnm);
  }
  return clip;
}  // end of loadClip()


private void wait(int delay)
{ try {
    Thread.sleep(delay); // in ms
  }
  catch (InterruptedException e)
  { e.printStackTrace(); }
}
```

ClipsPlayer can be called like so:

```
ClipsPlayer cp = new ClipsPlayer();
cp.play("windChime", 1930);   // a wind chime clip of 1.93 secs
cp.play("keyDrop", 430);      // a key dropping clip of 0.43 secs
cp.play("windChime", 1930);
```

When "windchime.wav" and "keyDrop.wav" are first played, they're loaded into ClipsPlayer's HashMap. This will delay their playing slightly, but I'm assuming that the clips are quite short, and so will be quick to load. Another drawback of the class is the need to supply a clip's duration every time it's played. It would be more

convenient to store it with the AudioClip when it's first loaded, and use that information each time the clip is played.

2. Playing a Note

The simplest way to play a note is to employ a MIDI synthesizer via the Java Sound API's javax.sound.midi package.

A Java program can communicate with a synthesizer by sending it a stream of messages (as MidiMessage objects). Each message is routed to a particular channel, which represents a different kind of musical instrument. The idea is shown in greatly simplified form in Figure 2.

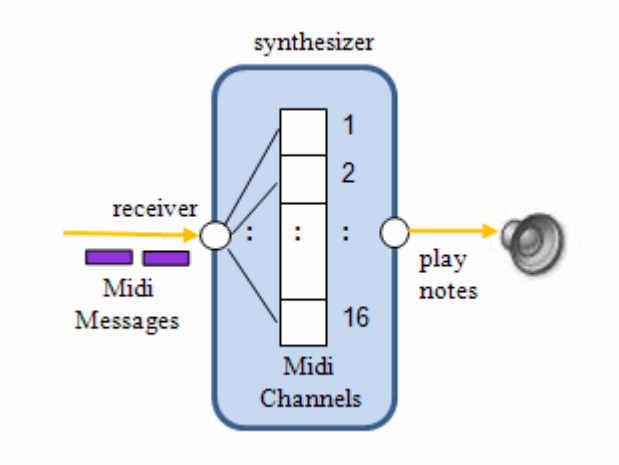

Figure 2. A MIDI Synthesizer.

The javax.sound.midi.MidiSystem class provides access to the MIDI resources on a machine. A synthesizer is sent MIDI messages via a receiver port, which is obtained like so:

```
Synthesizer synthesizer = MidiSystem.getSynthesizer();
synthesizer.open();
Receiver receiver = synthesizer.getReceiver();
```

MIDI messages can be encoded using three subclasses of MidiMessage: ShortMessage, SysexMessage, and MetaMessage. The ShortMessage is the most useful for my needs, since it includes the NOTE_ON and NOTE_OFF messages for starting and terminating note playing. The MidiChannel class offers noteOn() and noteOff() methods for building the messages:

```
void noteOn(int noteNumber, int velocity);
void noteOff(int noteNumber, int velocity);
void noteOff(int noteNumber);
```

A note number is a MIDI number assigned to a musical note, while velocity is equivalent to loudness. A note will keep playing after a noteOn() call, until it's

terminated with noteOff(). The two-argument form of noteOff() affects how quickly the note fades away.

Playing a note corresponds to sending a NOTE_ON message, letting it play for a while, and then killing it with a NOTE_OFF message. This can be wrapped up in a playNote() method (which I'll be revising a little later on in this chapter):

```
//globals
private static final int VOLUME = 90;  // fixed volume for notes

private Receiver receiver;


public void playNote(int note, int duration, int channel)
// first version; second version given below
{
  ShortMessage msg = new ShortMessage();
  try {
    msg.setMessage(ShortMessage.NOTE_ON, channel, note, VOLUME);
    receiver.send(msg, -1);  // -1 means play immediately

    wait(duration);

    // reuse the ShortMessage object
    msg.setMessage(ShortMessage.NOTE_OFF, channel, note, VOLUME);
    receiver.send(msg, -1);
  }
  catch (InvalidMidiDataException e)
  {  System.out.println(e);  }
}  // end of playNote()


private void wait(int duration)
{ try {
    Thread.sleep(duration); // in ms
  }
  catch (InterruptedException e)
  { e.printStackTrace(); }
}  // end of wait()
```

A MIDI message must be sent with a time-stamp. The -1 used above means that the message should be processed immediately.

The following code fragment plays a round of applause:

```
for (int i=0; i < 10; i++)
  playNote(39, 1000, 9); // note 39 sent to the drum channel, 9
```

Channel 9 plays different percussion and audio effects depending on the note numbers sent to it. Note 39 corresponds to a "hand clap". A complete list of the mappings from MIDI numbers to drum sounds can be found at http://www.midi.org/techspecs/gm1sound.php.

2.1. Note Names and Numbers

It's a lot easier (at least for programmers with some musical knowledge), to specify a note using a name derived from the piano keyboard rather than as a MIDI number.

A piano keyboard has a mix of black and white keys, as in Figure 3.

Figure 3. Part of a Piano Keyboard.

Keys are grouped into octaves, each octave consisting of twelve consecutive white and black keys. The white keys are labeled with the letters 'A' to 'G' and an octave number. For example, the note C4 is the white key closest to the center of the keyboard, often referred to as "middle C". The '4' means that the key is in the fourth octave, counting from the left of the keyboard.

A black key is labeled with the letter of the preceding white key and a sharp (#). For instance, the black key following C4 is C#4. A note for musicians: for simplicity's sake, I'll be ignoring flats in this discussion.

Figure 4 shows the keyboard fragment of Figure 3 again, but labeled with note names. I've assumed that the first white key is C4.

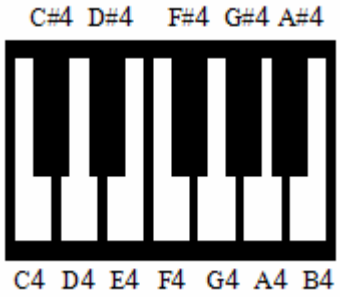

Figure 4. Partial Piano Keyboard with Note Names.

Figure 4 utilizes the C Major scale, where the letters appear in the order C, D, E, F, G, A, and B.

After B4, the fifth octave begins, starting with C5 and repeating the same sequence as in the fourth octave. Before C4 is the third octave, which ends with B3.

Having introduced note names, I can now explain their MIDI note numbers. MIDI notes can range between 0 and 127, extending well beyond the piano's scope, which only has 88 standard keys. This means that the note naming scheme gets a little strange below note 12 (C0), since we have to start talking about octave -1 (e.g. see the table at http://www.harmony-central.com/MIDI/Doc/table2.html). Additionally, a maximum value of 127 means that note names only go up to G9; there is no G#9.

Table 1 shows the mapping of MIDI numbers to notes for the 4th octave.

MIDI Number	Note Name
60	C4
61	C#4
62	D4
63	D#4
64	E4
65	F4
66	F#4
67	G4
68	G#4
69	A4
70	A#4
71	B4

Table 1. MIDI Numbers and Note Names.

This table can be used as the basis of the method getKey() which converts a note name string (e.g. "C4") into a MIDI note number (60):

```
// globals
/* The note offsets use the "C" major scale, which
   has the order "C D E F G A B", but the offsets are
   stored in the order "A B C D E F G" to simplify their
   lookup. */
private static final int[] cOffsets =  {9, 11, 0, 2, 4, 5, 7};
                                     // A   B  C  D  E  F  G
private static final int C4_KEY = 60;
      // C4 is the "C" in the 4th octave on a piano

private static final int OCTAVE = 12;    // note size of an octave



private int getKey(String noteStr)
// Convert a note string (e.g. "C4", "B5#") into a key.
{
  char[] letters = noteStr.toCharArray();
  if (letters.length < 2) {
    System.out.println("Incorrect note syntax; using C4");
    return C4_KEY;
  }

  // look at note letter in letters[0]
  int c_offset = 0;
  if ((letters[0] >= 'A') && (letters[0] <= 'G'))
    c_offset = cOffsets[letters[0] - 'A'];
```



```
    else
      System.out.println("Incorrect letter: " + letters[0] +
                                                ", using C");
    // look at octave number in letters[1]
    int range = C4_KEY;
    if ((letters[1] >= '0') && (letters[1] <= '9'))
      range = OCTAVE * (letters[1] - '0' + 1);  // plus 1 for midi
    else
      System.out.println("Incorrect number: " + letters[1] +
                                                  ", using 4");
    // look at optional sharp in letters[2]
    int sharp = 0;
    if ((letters.length > 2) && (letters[2] == '#'))
      sharp = 1;    // a sharp is 1 note higher
                    // (represented by the black keys on a piano)
    int key = range + c_offset + sharp;
    // System.out.println("note: " + noteStr + "; key: " + key);

    return key;
  }  // end of getKey()
```

The parsing of the string is simpler if I abuse the note name notation by moving the sharp '#' to the end of the string. Therefore, the user must write "C4#" rather than the standard C#4.

playNote() can now be revised to accept a note name:

```
// global
// piano channel used by the synthesizer
private static final int CHANNEL = 0;


public void play(String noteStr, int duration)
// second version; play note string for specified duration
{
  if (receiver == null) {
    System.out.println("No synthesizer to play note: " + noteStr);
    return;
  }

  int note = getKey(noteStr);

  ShortMessage message = new ShortMessage();
  try {
    message.setMessage(ShortMessage.NOTE_ON, CHANNEL, note, VOLUME);
    receiver.send(message, -1);
    wait(duration); // in ms
    message.setMessage(ShortMessage.NOTE_OFF, CHANNEL, note, VOLUME);
    receiver.send(message, -1);
  }
  catch (InvalidMidiDataException e)
  {  System.out.println(e);  }
}  // end of play()
```

I've also hard-wired the code to utilize channel 0, which plays notes as if on a piano.

2.2. The Rest of the NotesPlayer Class

The NotesPlayer class utilizes getKey() and the final version of playNote(). It also employs a findReceiver() method to find a named MIDI device and return its receiver port:

```
// global
private Receiver receiver;

public NotesPlayer()
{  receiver = findReceiver("Java Sound"); }


private Receiver findReceiver(String name)
// find the receiver MIDI device called name
{
  System.out.println("Searching for device: " + name);
  try {
    MidiDevice.Info[] devices = MidiSystem.getMidiDeviceInfo();
    for(MidiDevice.Info devInfo : devices)
      if (devInfo.getName().startsWith(name)) {
        MidiDevice dev = MidiSystem.getMidiDevice(devInfo);
        if (dev.getMaxReceivers() != 0) {
          System.out.println("Found: " + devInfo.getDescription());
          dev.open();
          System.out.println("Opening the Receiver");
          return dev.getReceiver();
        }
      }
    System.out.println("Device not found");
  }
  catch (MidiUnavailableException e)
  {  System.out.println(e);  }
  return null;
}  // end of findReceiver()
```

The call to findReceiver() from the NotesPlayer() constructor searches for "Java Sound" which begins the default name for the synthesizer used by Java.

NotesPlayer also has a method for closing the receiver port at the end of the note playing:

```
public void closeDown()
{ if (receiver != null)
    receiver.close();
}
```

A typical use of NotesPlayer:

```
NotesPlayer player = new NotesPlayer();
player.play("C4#", 500);
player.play("D4", 1000);
player.play("E4", 1000);
player.closeDown();
```

Three notes are played, the first for half a second, the others for a second.

3. Using a Speech Synthesizer

The Java Speech API (JSAPI, http://java.sun.com/products/java-media/speech/) covers two speech-related technologies: speech recognition, which converts spoken language into text, and speech synthesis that goes the other way, rendering text into spoken audio. I only need the latter, so can make use of FreeTTS, (http://freetts.sourceforge.net/), a fairly complete implementation of the speech synthesis parts of JSAPI. The main missing feature is JSML processing (the Java Speech Markup Language), which can be employed to fine-tune pronunciation, stress, pauses, and other aspects of speaking, to make a computerized voice sound more natural.

I downloaded the Windows binary version of FreeTTS v.1.2.2, unzipped it, and stored it in a convenient location (d:\freetts-1.2\ in my case).

Rather confusingly, there's initially no JSAPI JAR file in the FreeTTS lib\ directory (d:\freetts-1.2\lib\); it only appears after the user executes jsapi.exe in that directory.

Many FreeTTS programming examples will not work unless the speech.properties file in d:\freetts-1.2\ is copied either to your home directory or to the JRE's lib\ directory (which for me is c:\Program Files\Java\jre1.6.0_03\lib).

More details on these set-up issues can be found at http://freetts.sourceforge.net/docs/jsapi_setup.html

Having done all this, the easiest way of testing things is to go to d:\freetts-1.2\bin\, and type:

```
> java –jar HelloWorld.jar
```

It uses an US male voice called "kevin16" to say "hello world".

3.1. Adding Voices

FreeTTS's "kevin16" voice isn't that great, as the HelloWorld example illustrates. Fortunately, FreeTTS supports several better US voices, available from the MBROLA website (http://tcts.fpms.ac.be/synthesis/mbrola.html).

I downloaded four things: the PC/DOS version of the MBROLA software (in mbr301d.zip), and three voices called us1, us2, and us3. I moved the unzipped executable, mbrola.exe, to d:\freetts-1.2\mbrola\, along with the three unzipped voice folders (us1\, us2\, and us3\).

I checked if the voices had been correctly installed by writing a program that lists all the voices known to FreeTTS. My ListVoices.java application starts by obtaining a list of the machine's synthesizer engines by querying javax.speech.Central.

```
private static EngineList getEngines()
// first version – using JSAPI javax.speech.Central
{
  // Don't set any properties so all synthesizers will be returned
  SynthesizerModeDesc emptyDesc = new SynthesizerModeDesc();

  EngineList engineList = null;
  try {
    engineList = Central.availableSynthesizers(emptyDesc);
```

```
  }
  catch (Exception e)
  {  System.out.println(e);
     System.exit(1);
  }
  return engineList;
}  // end of getEngines()
```

Central.availableSynthesizers() is called with a description of the required synthesizer attributes, which in this case can be anything. Other examples of how to use Central can be found in its online documentation at http://java.sun.com/products/java-media/speech/forDevelopers/jsapi-doc/.

Central is a JSAPI class, which is implemented using lower-level FreeTTS classes. If you don't mind calling these classes directly, then FreeTTS's installation can be simplified. In particular, it's no longer necessary to move the speech.properties file to the JRE lib\ directory.

I rewrote getEngines() as:

```
private static EngineList getEngines()
// second version – uses FreeTTS FreeTTSEngineCentral
{
  // Don't set any properties so all synthesizers will be returned
  SynthesizerModeDesc emptyDesc = new SynthesizerModeDesc();

  EngineList engineList = null;
  try {
    FreeTTSEngineCentral central = new FreeTTSEngineCentral();
    engineList = central.createEngineList(emptyDesc);
  }
  catch (Exception e)
  {  System.out.println(e);
     System.exit(1);
  }
  return engineList;
}  // end of getEngines()
```

The use of the lower-level FreeTTSEngineCentral class means that I don't need Central, and so don't need to move the speech.properties file at FreeTTS installation time. The documentation for the FreeTTS-specific classes can be found at d:/freetts-1.2/javadoc/index.html.

Once I have an engines list, the voices they offer can be displayed:

```
EngineList engineList = getEngines();

// loop over the synthesizers
for (int e = 0; e < engineList.size(); e++) {
  SynthesizerModeDesc desc = (SynthesizerModeDesc) engineList.get(e);

  // loop over all the voices for this synthesizer
  Voice[] voices = desc.getVoices();
  for (Voice voice : voices)
    System.out.println( "  " + desc.getEngineName() +
                ": Voice: " + voice.getName() +
                "; Gender: " + genderStr( voice.getGender() ));
```

```
}
```

The gender information is represented by integer constants, so genderStr() changes it to more useful text:

```
private static String genderStr(int gender)
{
  switch (gender) {
    case Voice.GENDER_FEMALE: return "female";
    case Voice.GENDER_MALE: return "male";
    case Voice.GENDER_NEUTRAL: return "neutral";
    case Voice.GENDER_DONT_CARE:
    default: return "unknown";
  }
}  // end of genderStr()
```

All this code is wrapped up inside a ListVoices class, which can be compiled and executed like so:

```
> set "TTS=d:\freetts-1.2\lib"

> javac -cp "%TTS%\jsapi.jar;%TTS%\freetts.jar;
                %TTS%\freetts-jsapi10.jar;."  ListVoices.java

> java -Dmbrola.base=d:\freetts-1.2\mbrola
        -cp "%TTS%\jsapi.jar;%TTS%\freetts.jar;
             %TTS%\freetts-jsapi10.jar;."  ListVoices
```

Not every program requires jsapi.jar, freetts.jar, and freetts-jsapi10.jar, in the compilation and execution calls, but many do.

If we want to employ the three MBROLA US voices, then the call to java.exe must include the mbrola.base property (which states the location of the directory holding the mbrola.exe executable).

In general, its useful to package up these lengthy compilation and execution calls in DOS batch files, to save on typing.

The output of the call to ListVoices will be something like:

```
FreeTTS en_US time synthesizer: Voice: alan; Gender: male
FreeTTS en_US general synthesizer: Voice: kevin; Gender: male
FreeTTS en_US general synthesizer: Voice: kevin16; Gender: male
FreeTTS en_US general synthesizer: Voice: mbrola_us1; Gender: female
FreeTTS en_US general synthesizer: Voice: mbrola_us2; Gender: male
FreeTTS en_US general synthesizer: Voice: mbrola_us3; Gender: male
```

The alan, kevin, and kevin16 voices come with FreeTTS, and the three US voices from MBROLA. I need a good quality, general voice for speaking words, so will use mbrola_us1 from now on.

3.2. Making a Program Speak

I'll introduce the main elements of JSAPI by writing a Speaker class, which reads a string from the command line and speaks it. An example call to Speaker:

```
> java -Dmbrola.base=d:\freetts-1.2\mbrola
       -cp "%TTS%\jsapi.jar;%TTS%\freetts.jar;
            %TTS%\freetts-jsapi10.jar;."  Speaker "my name is jane"
```

Speaker is passed "my name is jane", and says it using the US female voice mbrola_us1.

A speech synthesis application (such as Speaker) typically passes through seven stages:

1) The desired properties for the synthesizer are collected together in a mode descriptor. The descriptor might include the name of the synthesizer that we require, its intended mode of use, and the chosen language.
2) The synthesizer engine is created, using the mode descriptor as a guide.
3) The synthesizer is moved to the ALLOCATED state, which causes it to load resources.
4) Once in the ALLOCATED state, various tweaks can be applied to the engine, such as changing its voice.
5) The synthesizer is moved into an ALLOCATION sub-state called RESUMED, which makes it ready to process text into speech. It's also a good idea to make sure that its speaking queue, where text is stored waiting for processing, is in the QUEUE_EMPTY state. This ensures that the supplied text will be processed as quickly as possible.
6) The text is passed to the engine, which renders it into speech. It’s possible to have the application wait for the speaking to finish by making it suspend until the speaking queue is once again empty.
7) The synthesizer is closed down by moving it to the DEALLOCATED state, making it release its resources.

All these stages are present in the Speaker class, called from main():

```
public static void main(String[] args)
{
  String msg;
  if (args.length != 1) {
    System.out.println("Usage: Speaker \"sentence to be said\"");
    msg = "You must tell me what to say";
  }
  else
     msg = args[0];

  Speaker speaker = new Speaker();
  speaker.say(msg);
  speaker.closeDown();
}  // end of main()
```


The Speaker constructor carries out stages 1-5, say() implements stage 6, and closeDown() performs stage 7.

The Speaker() method:

```
// globals
private static final String VOICE = "mbrola_us1";
private Synthesizer synthesizer;


public Speaker()
{
  try {
    System.out.println("General Synthesizer using \"" + VOICE +
                               "\" voice being initialized...");

    // specify the required synthesizer properties (stage 1)
    SynthesizerModeDesc desc = new SynthesizerModeDesc(
              null,          // engine name (don't care)
              "general",     // mode name -- general usage
              Locale.US,     // locale
              null,      // prefer a running synthesizer (don't care)
              null);     // voice (none specified yet)

    // create the synthesizer (stage 2)
    FreeTTSEngineCentral central = new FreeTTSEngineCentral();
                              // so no need for "speech.properties"
    EngineList list = central.createEngineList(desc);
    if(list.size() > 0) {
      EngineCreate creator = (EngineCreate)list.get(0);
      synthesizer = (Synthesizer)creator.createEngine();
    }

    if (synthesizer == null) {
      System.err.println("Cannot create synthesizer");
      System.exit(1);
    }

    // allocate resources (stage 3)
    synthesizer.allocate();
    synthesizer.waitEngineState(Synthesizer.ALLOCATED);

    // modify synthesizer properties (stage 4)
    SynthesizerProperties synProps =
              synthesizer.getSynthesizerProperties();
    synProps.setVoice( getVoice(VOICE));

    // get synthesizer ready to speak (stage 5)
    synthesizer.resume();

    // wait until the synthesizer is ready to speak
    synthesizer.waitEngineState(Synthesizer.RESUMED);
    synthesizer.waitEngineState(Synthesizer.QUEUE_EMPTY);
    System.out.println("Synthesizer ready");
  }
  catch (Exception e)
  {  System.out.println(e);
     System.exit(1);
  }
```



```
}  // end of Speaker()
```

The SynthesizerModeDesc class is used to create a mode descriptor. The null arguments in its constructor mean that the description doesn't care about the engine name, running behavior, or voices that the synthesizer offers. The descriptor only requires that the synthesizer can manage general speech, in American English.

The several calls to Synthesizer.waitEngineState() force the Speech object to wait until the synthesizer enters a desired state. For example, the call to Synthesizer.resume() in stage 5 requests that the synthesizer moves into the RESUMED state. The subsequent waitEngineState() calls waits for that state to be attained.

Once the synthesizer is in an ALLOCATED state, its behavior can be adjusted using a SynthesizerProperties object. The code above only sets the voice, but it’s also possible to change attributes such as the pitch and speaking rate. SynthesizerProperties settings are considered to be hints, so may have no effect, depending on the synthesizer.

Specifying a Voice

The getVoice() method searches through the voices associated with the synthesizer, looking for the specified voice. If it's present then its corresponding Voice object is returned, and subsequently employed by the synthesizer.

```
private Voice getVoice(String voiceName)
{
  // get the properties for this synthesizer engine
  SynthesizerModeDesc desc =
        (SynthesizerModeDesc) synthesizer.getEngineModeDesc();
  Voice[] voices = desc.getVoices();

  // check if voiceName is a known Voice for this engine
  Voice voice = null;
  for(int i = 0; i < voices.length; i++) {
    if (voices[i].getName().equals(voiceName)) {
      voice = voices[i];
      break;
    }
  }
    if (voice == null) {
    System.err.println("Synthesizer could not find the " +
                                        voiceName + " voice");
    System.exit(1);
  }
  return voice;
} // end of getVoice()
```

Speaking a Response

The say() method requests that a message is spoken, and returns when the speech is finished.

```
public void say(String msg)
```

```
{
  try {
    synthesizer.speakPlainText(msg, null);
                        // add msg to the speaking queue

   // wait for the queue to empty (i.e. until msg has been said)
   synthesizer.waitEngineState(Synthesizer.QUEUE_EMPTY);
  }
  catch (Exception e)
  {  System.out.println(e); }
}  // end of sayMessage()
```

The second argument of Synthesizer.speakPlainText() can be assigned a reference to a SpeakableListener object. The SpeakableListener interface processes SpeakableEvent events which report on the progress of the spoken output. That level of complexity is unnecessary here, since the application only needs to wait until the sentence has been said.

Closing Down the Synthesizer

As the application finishes, it calls Speaker's closeDown() method to terminate the synthesizer:

```
public void closeDown()
{
  try {
    synthesizer.cancel();        // cancel any speaking
    synthesizer.deallocate();    // free the engine's resources
  }
  catch (Exception e)
  {  System.out.println(e); }
}
```

Any currently executing speech is cancelled before the resources are deallocated; this corresponds to stage 7 in the list above.

4. Back to Sonification

Over the last 15 pages or so, I've described three audio 'building blocks': the ClipsPlayer class for playing clips, NotesPlayer which uses a MIDI synthesizer to generate notes, and Speaker which reads out strings using the FreeTTS implementation of JSAPI. Now it's time to return to my Sonification application, first shown in Figure 1 (and repeated here as Figure 5 to save you having to flick back to the beginning).

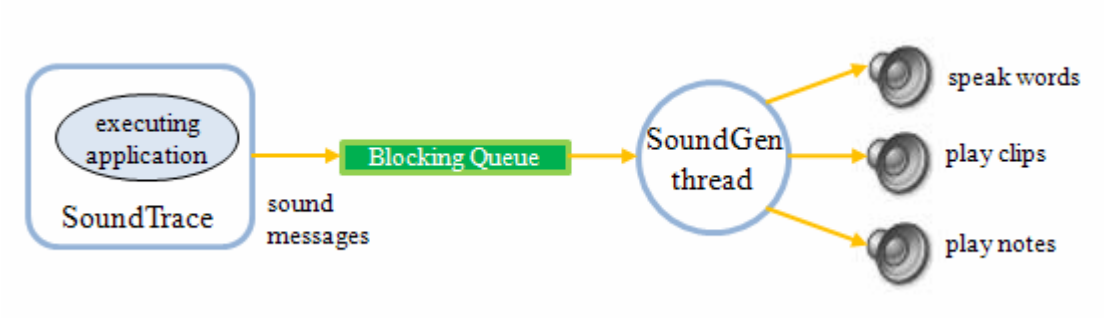

Figure 5. Sonification of a Java Application (Again).

SoundTrace is a tracer employing the Java Debug Interface (JDI) API, while sound generation is managed by a thread which reads messages from a queue filled by the tracer. SoundGen utilizes ClipsPlayer, NotesPlayer, and Speaker to generate audio output.

Figure 6 shows the UML class diagrams for the sonifier, with only public methods listed for each class.

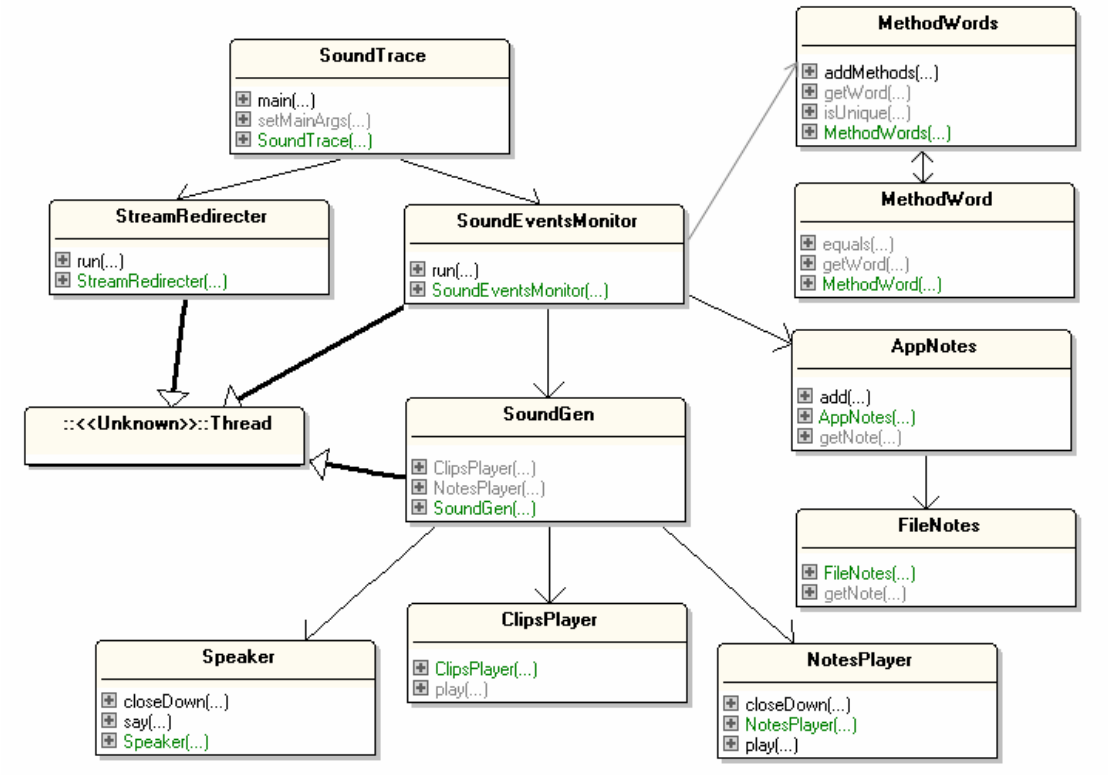

Figure 6. SoundTrace Class Diagrams

The three classes at the top left of Figure 6 (SoundTrace, StreamRedirecter, and SoundEventsMonitor) contain the tracer code.

SoundTrace.java is little more than a renamed SimpleTrace.java from Java Art Chapter 3. It sets up the command-line launching connection which starts the JVM and creates a local link with the JVM on the same machine. It passes the application's name and input arguments over to the JVM, and employs the StreamRedirecter class to redirect the JVM's output and error streams to stdout and stderr. StreamRedirecter is unchanged from previous chapters.

SoundEventsMonitor is a subset of JDIEventMonitor from Java Art Chapter 3, and I'll only describe the parts which are new or differ significantly from before. SoundEventsMonitor monitors incoming JDI events for a program running in the JVM. When a method is called, a word is spoken, when keywords in the code are encountered, musical notes are played, and when the program starts, ends, and when a method returns, sound clips are played.

SoundEventsMonitor maintains two types of audio information:

1) it uses the MethodWords and MethodWord classes to store the words that will be spoken when methods are called in the application being traced;
2) it calls AppNotes and FileNotes to analyze the application's code, and all the lines containing keywords are assigned musical notes. This is a form of static analysis since the program files are scanned before their code is executed, at class load time.

Three types of messages are placed onto the blocking queue going to the SoundGen thread:

```
$w <word to be said>
$n <note to be played>
$c <wav filename (a clip) to be played>  <duration(ms)>
```

For example:

```
$w foo        // say the word "foo"
$n C4         // play the note C4
$c zap 500    // play zap.wav which lasts for 500ms
```

SoundGen reads these audio messages, passing them to the relevant audio object (ClipsPlayer, NotesPlayer, or Speaker).

4.1. Initializing Event Monitoring

The SoundEventsMonitor constructor initializes the AppNotes and MethodWords objects, and sets up the blocking queue connection to the SoundGen thread.

```
// globals
private final VirtualMachine vm;   // the JVM

// musical data objects
private AppNotes appNotes;
       // stores musical notes representing the application code

private MethodWords methodWords;
       // stores words that will be spoken when methods are called
```



```
// sound generation objects
private BlockingQueue<String> sndQueue;   // holds sound messages
private SoundGen soundGen;     // processes the sound messages


public SoundEventsMonitor(VirtualMachine jvm)
{
  super("SoundEventsMonitor");
  vm = jvm;

  appNotes = new AppNotes();
  methodWords = new MethodWords();

  // initialize the message queue and the sound generation thread
  sndQueue = new LinkedBlockingQueue<String>();
  soundGen = new SoundGen(sndQueue);
  soundGen.start();

  setEventRequests();
}  // end of SoundEventsMonitor()
```

Four types of program events are requested in setEventRequests().

```
private void setEventRequests()
{
  EventRequestManager mgr = vm.eventRequestManager();

  // report 'method entries'
  MethodEntryRequest menr = mgr.createMethodEntryRequest();
  for (int i = 0; i < excludes.length; ++i)
    menr.addClassExclusionFilter(excludes[i]);
  menr.setSuspendPolicy(EventRequest.SUSPEND_EVENT_THREAD);
  menr.enable();

  // report 'method exits'
  MethodExitRequest mexr = mgr.createMethodExitRequest();
  for (int i = 0; i < excludes.length; ++i)
    mexr.addClassExclusionFilter(excludes[i]);
  mexr.setSuspendPolicy(EventRequest.SUSPEND_EVENT_THREAD);
  mexr.enable();

  // report 'class loads'
  ClassPrepareRequest cpr = mgr.createClassPrepareRequest();
  for (int i = 0; i < excludes.length; ++i)
    cpr.addClassExclusionFilter(excludes[i]);
  cpr.enable();

  // report 'thread starts'
  ThreadStartRequest tsr = mgr.createThreadStartRequest();
  tsr.enable();
}  // end of setEventRequests()
```

Method entry and exits are monitored so that various sounds can be generated. Class loads must be observed so that the corresponding Java file can be analyzed by AppNotes and FileNotes to create musical notes for its keywords. A thread start triggers code stepping so the correct note can be played when a line is executed.

4.2. Handling Events

As in JDIEventMonitor in Java Art Chapter 3, SoundEventsMonitor calls handleEvent() to process an incoming event:

```
private void handleEvent(Event event)
{
  // method entry/exit events
  if (event instanceof MethodEntryEvent)
    methodEntryEvent((MethodEntryEvent) event);
  else if (event instanceof MethodExitEvent)
    methodExitEvent((MethodExitEvent) event);

  // class load events
  else if (event instanceof ClassPrepareEvent)
    classPrepareEvent((ClassPrepareEvent) event);

  // thread start events
  else if (event instanceof ThreadStartEvent)
    threadStartEvent((ThreadStartEvent) event);

  // step event -- a line of code is about to be executed
  else if (event instanceof StepEvent)
    stepEvent((StepEvent) event);

  // VM events
  else if (event instanceof VMStartEvent)
    vmStartEvent((VMStartEvent) event);
  else if (event instanceof VMDeathEvent)
    vmDeathEvent((VMDeathEvent) event);
  else if (event instanceof VMDisconnectEvent)
    vmDisconnectEvent((VMDisconnectEvent) event);

  else
    throw new Error("Unexpected event type");
}  // end of handleEvent()
```

handleEvent() deals with methods, classes, threads, and step events. The JVM-related processing in vmStartEvent(), vmDeathEvent(), and vmDisconnectEvent() is unchanged from JDIEventMonitor.

Entering a Method

Method entry triggers the playing of a clip if the method is main(), and a method word is spoken.

```
// global
private MethodWords methodWords;
    // stores the words that will be spoken when methods are called


private void methodEntryEvent(MethodEntryEvent event)
{
  Method meth = event.method();
  String className = meth.declaringType().name();

  System.out.println();
```



```
  if (meth.isConstructor())     // use class name for the constructor
    System.out.println("entered " + className + " constructor");
  else
    System.out.println("entered " + className+"."+meth.name() +"()");

  if (meth.name().equals("main"))   // program starting so play clip
    saySound("$c windChime 1930");

  // report method entry by saying a word
  String word = methodWords.getWord(className, meth);
  if (word != null) {
    System.out.println("  saying " + word);
    saySound("$w " + word);
  }
  else
    System.out.println("Could not find method word to say");
}  // end of methodEntryEvent()
```

The clip played when main() starts is a wind chime which lasts for 1.93 seconds. The word spoken for a method is retrieved from the MethodWords object.

saySound() doesn't say the sound directly; it puts a message onto the blocking queue so it can be processed by the SoundGen thread.

```
// global
private BlockingQueue<String> sndQueue;   // holds sound messages


private void saySound(String msg)
{ try {
    sndQueue.put(msg);
  }
  catch(InterruptedException e) {}
}
```

Leaving a Method

methodExitEvent() is called when all the code in a method has been executed, and the application is about to return to the calling function. A return from main() (i.e. the termination of the application) is signaled by playing a clip of a key dropping, while other returns are represented by a 'zapping' sound.

```
private void methodExitEvent(MethodExitEvent event)
{
  Method meth = event.method();
  String className = meth.declaringType().name();

  if (meth.isConstructor())
    System.out.println("exiting " + className + " constructor");
  else
    System.out.println("exiting " + className + "."
                                        + meth.name() + "()" );

  // report method return/ program exit
  if (meth.name().equals("main"))   // program is exiting
    saySound("$c keyDrop 430");
  else              // the method is returning
```

```
    saySound("$c zap 500");
  System.out.println();
}  // end of methodExitEvent()
```

Clip playing is only requested in methodEntryEvent() and methodExitEvent(), and so I've hard-wired the details of the clip messages into those methods.

If more clips were used, across more methods, then it would be better to store the clip details (i.e. the event trigger name, wave filename, and duration) in a separate class, which could perhaps load it's information at start-up time.

Loading a Class

classPrepareEvent() is called after a new class has been loaded, which is a good time to parse the class's file to build a collection of musical notes, and use the class's method names to create the words stored in methodWords.

```
// globals
private AppNotes appNotes;
private MethodWords methodWords;


private void classPrepareEvent(ClassPrepareEvent event)
{
  ReferenceType ref = event.referenceType();

  List<Field> fields = ref.fields();
  List<Method> methods = ref.methods();
  String className = ref.name();

  String fnm;
  try {
    fnm = ref.sourceName();  // get filename of the class
    appNotes.add(fnm);       // create musical notes for code in fnm
  }
  catch (AbsentInformationException e)
  {  fnm = "??"; }

  System.out.println("loaded class: " + className + " from " +
                fnm + " - fields=" + fields.size() +
                      ", methods=" + methods.size() );

  methodWords.addMethods(className, methods);
                                // convert method names into words
}  // end of classPrepareEvent()
```

Starting a Thread

When a new thread starts running, the tracer needs to switch on single stepping so it can monitor which lines are being executed.

```
private void threadStartEvent(ThreadStartEvent event)
{
   ThreadReference thr = event.thread();
```



```
  if (thr.name().equals("Signal Dispatcher") ||
      thr.name().equals("DestroyJavaVM") ||
      thr.name().startsWith("AWT-") )     // AWT threads
    return;

  if (thr.threadGroup().name().equals("system"))
    return;    //ignore system threads

  setStepping(thr);
} // end of threadStartEvent()
```

setStepping() asks the JVM to issue step events, which are sent out just before each line is executed.

```
private void setStepping(ThreadReference thr)
{
  EventRequestManager mgr = vm.eventRequestManager();
  StepRequest sr = mgr.createStepRequest(thr, StepRequest.STEP_LINE,
                                                StepRequest.STEP_INTO);
  sr.setSuspendPolicy(EventRequest.SUSPEND_EVENT_THREAD);
  for (int i = 0; i < excludes.length; ++i)
    sr.addClassExclusionFilter(excludes[i]);
  sr.enable();
}  // end of setStepping()
```

There are a few different kinds of step requests, but the most common is a combination of STEP_INTO and STEP_LINE which means that every line in every method will be examined.

Single Stepping Event Handling

After setStepping() has requested single stepping, StepEvents will start to be added to the event queue, and handleEvent() will process them by calling stepEvent().

An event contains the location of the line that's about to be executed, which includes the code's filename and line number. These are passed to the AppNotes object to help retrieve a note string if that line contains a keyword. If a note is found (e.g. "C4"), then it's sent as an "$n" message over to SoundGen.

```
private void stepEvent(StepEvent event)
{
  Location loc = event.location();
  try {   // get the line of code
    String fnm = loc.sourceName();  // get filename of code
    int lineNum = loc.lineNumber();
    String note = appNotes.getNote(fnm, lineNum);

    // play note for the line
    if (note != null) {
      System.out.println(fnm + "." + lineNum + ": play " + note);
      saySound("$n " + note);
    }
  }
  catch (AbsentInformationException e) {}
}  // end of stepEvent()
```


4.3. Extracting Words from Method Names

The MethodWords class is little more than a wrapper around an ArrayList of MethodWord objects, along with several methods for adding to and searching through the list:

```
// global in MethodWords class
private ArrayList<MethodWord> mWords;
    // stores the words that will be spoken when methods are called

public MethodWords()
{  mWords = new ArrayList<MethodWord>();  }
```

addMethod() is typical of the class's code. It adds a MethodWord object to the list, if that object isn't already there.

```
public void addMethod(String className, Method meth)
{
  if (getWord(className, meth) != null)  // already present
    return;  // don't add

  MethodWord mw;
  if (meth.isConstructor())   // use class name for the constructor
    mw = new MethodWord(className, className, this);
  else
    mw = new MethodWord(className, meth.name(), this);
  mWords.add(mw);
}  // end of addMethod()
```

A minor complication of addMethod() (and several other methods in MethodWords) is the need to map the Method object into a method name. I set the constructor's method name to be its class name, rather than the "<init>" string returned by Method.name().

The primary aim of a MethodWord object is to map the class and method names onto a word, which can be spoken by the FreeTTS speech synthesizer. The tricky part is that each word must be unique and preferable short.

```
// globals in MethodWord class
private String className, methodName;
private String wordToSay;


public MethodWord(String cName, String mName, MethodWords mWords)
{
  className = cName;
  methodName = mName;
  wordToSay = buildWord(methodName, mWords);
  System.out.println(className + ":" + methodName +
       "() is represented by the word \"" + wordToSay + "\"");
}  // end of MethodWord()
```


The trickiness is hidden inside buildWord() which tries a number of different word creation strategies, testing each one for uniqueness against the existing words in the MethodWords object, mWords.

```
private String buildWord(String mName, MethodWords mWords)
{
  if (mName.length() < MIN_LEN)
    // if method name is short, use it as the word to say
    if (mWords.isUnique(mName))
      return mName;

  // shorten method name
  String shortName;
  for (int i= MIN_LEN; i <= mName.length(); i++) {
    shortName = mName.substring(0, i);
    if (mWords.isUnique(shortName))
      return shortName;
  }

  // if we get to here then no substring of mName is unique
  // so add class name to it's front, and try again

  String longName = className + " " + mName;
  for (int i= (className.length()+1)+MIN_LEN;
                           i <= longName.length(); i++) {
    shortName = longName.substring(0, i);
    if (mWords.isUnique(shortName))
      return shortName;
  }

  // if we get here then no substring of "className mName" is unique
  // so keep adding a number to mName, until it's unique
  int count = 1;
  String numName = mName + " " + count;
  while (!mWords.isUnique(numName)) {
    count++;
    numName = mName + " " + count;
  }
  return numName;
}  // end of buildWord()
```

buildWords() tries four ways of making a unique word from a method name:

- if the name is short enough then use that unchanged;
- if the name isn't short enough then try increasingly longer substrings of the method name;
- if no subsequence of the name is unique then try again with the class name appended to the front;
- if a class+method name isn't unique then add a digit to the end of the method name.

4.4. Converting Files into Musical Notes

The AppNotes and FileNotes classes parse the files involved in the application, converting the lines containing specified keywords into musical note strings. At run time, when the tracer executes a line of code, its corresponding note is played (if one exists).

AppNotes is mostly just a wrapper around a TreeMap of FileNotes objects, and several methods for adding to and searching through the map:

```
// global in AppNotes class
private TreeMap<String,FileNotes> filesNotesMap;


public AppNotes()
{  filesNotesMap = new TreeMap<String,FileNotes>();  }
```

The String used as the map's key is the filename.

add() adds a FileNotes object to the map, if that object isn't already there.

```
public void add(String fnm)
{
  if (filesNotesMap.containsKey(fnm)) {
    System.out.println(fnm + "already stored");
    return;
  }
  filesNotesMap.put(fnm, new FileNotes(fnm));
  System.out.println(fnm + " added to files Notes");
}  // end of add()
```

FileNotes does all the real work. It loads a file line-by-line, analyzing each line of code. Those lines containing a keyword are converted into musical note strings, while "" is stored for the lines without keywords.

```
// global in FileNotes class
private ArrayList<String> codeNotes;
     /* each line of the input file is represented by
        a musical note string, or "" */

public FileNotes(String fileName)
{
  codeNotes = new ArrayList<String>();

  int lineNum = 0;
  String line = null;
  BufferedReader in = null;
  String[] words;

  System.out.println("Analyzing " + fileName);
  try {
    in = new BufferedReader(new FileReader(fileName));
    while ((line = in.readLine()) != null) {
      lineNum++;
      words = line.split("\\W+");    // split on word boundaries
      storeNote(words, lineNum);
    }
```



```
  }
  catch (IOException ex) {
    System.out.println("Problem reading " + fileName);
  }
  finally {
    try {
      if (in != null)
        in.close();
    }
    catch (IOException e) {}
  }
}  // end of showLines()
```

A line of input is split into tokens based on word boundaries, and the resulting array is examined for keywords by storeNote().

If storeNote() finds a keyword in the words array, then a corresponding note is added to the codeNotes list. If no keyword is found, then "" is stored instead, which ensures that every program line has an entry in the list.

```
// globals
private final String[] keywords =
    { "else", "break", "switch", "case", "finally", "catch", "for",
      "if", "try", "continue", "return", "default", "do", "while"
    };

// C D E F G A B  in the 4th and 5th octave on a piano
private final String[] notes =
    { "C4", "D4", "E4", "F4", "G4", "A4", "B4",
      "C5", "D5", "E5", "F5", "G5", "A5", "B5"
    };


private void storeNote(String[] words, int lineNum)
{
  for(String word : words)
    for (int i=0; i < keywords.length; i++)
      if (word.equalsIgnoreCase(keywords[i])) {
        codeNotes.add(notes[i]);
        System.out.println("  " + lineNum +
                           ": found " + keywords[i]);
        return;
      }
  codeNotes.add("");      // no keyword found
}  // end of storeNote()
```

The correspondence between keywords and notes is encoded by the ordering of the keywords[] and notes[] arrays. For example, if the "break" keyword is found on a line, which is the second string in keywords[], then the second note in notes[] ("D4") will be added to the codeNotes list.

I've restricted myself to control structure keywords, so that the played notes will give an indication of the control flow in the application. A complete list of Java keywords can be found at http://java.sun.com/docs/books/tutorial/java/nutsandbolts/_keywords.html.

To simplify matters a little, storeNote() stops searching a line as soon as a single keyword is found. Even when a line has multiple keywords, it will only be represented by one note in the list.

4.5. Static and Dynamic Analysis

AppNotes and FileNotes carry out a form of static analysis – the text of the application code is converted into lists of musical notes. This can be carried out at any time, for example in a separate pre-processing stage before the tracer begins, so it needn't have any speed impact.

Although static analysis is a great tool, it must often be combined with dynamic analysis (i.e. analysis performed at run-time), since it's only at run-time that certain information comes available, such as the user's input, the results of random-number generation, or the processing order between threads.

The dual use of static and dynamic analysis can be seen in SoundTrace – static analysis is performed in AppNotes and FileNotes (as we've just seen), and dynamic analysis of audio messages is carried out by SoundGen.

4.6. Generating Sounds

The SoundGen thread monitors a blocking queue which stores audio messages sent to it from SoundEventMonitor.

Three types of messages arrive on the BlockingQueue, with the formats

```
$w <word to be said>
$n <note to be played>
$c <wav filename (a clip) to be played>  <duration(ms)>
```

Every 0.5 seconds, SoundGen empties the queue of all the messages currently there, storing them in a separate list. Optimizations are applied to the list at run-time (i.e. dynamic analysis), before the messages are converted into either:

- a word spoken using the Speaker class (described in section 3.2)
- a musical note performed by the NotesPlayer class (see section 2)
- a WAV audio clip played with the ClipPlayer class (see section 1).

The SoundGen constructor initializes audio objects:

```
// globals
// sound processing objects
private Speaker speaker;
private NotesPlayer notesPlayer;
private ClipsPlayer clipsPlayer;

private BlockingQueue<String> queue;   // for incoming messages


public SoundGen(BlockingQueue<String> q)
{
  queue = q;
```

```
  // initialize sound playing tools
  speaker = new Speaker();
  notesPlayer = new NotesPlayer();
  clipsPlayer = new ClipsPlayer();
}  // end of SoundGen()
```

The queue emptying, analysis of the resulting list, and message processing is carried out in the thread's run() method:

```
// globals
private static final int DELAY = 500;
               // time delay (in ms) between queue examinations
private BlockingQueue<String> queue;   // for incoming messages
private boolean progFinished = false;


public void run()
// periodically drain and process messages
{
  ArrayList<String> msgs = new ArrayList<String>();
          // to store messages removed from the queue
  try {
    while (!progFinished) {
      msgs.clear();
      queue.drainTo(msgs);    // the draining may not return anything
      combineMsgs(msgs);
      for(String msg : msgs)
        processMsg(msg);
      wait(DELAY);
    }

    // finish off
    System.out.println("thread exiting");
    speaker.closeDown();
    notesPlayer.closeDown();
  }
  catch (Exception e)
  {  System.out.println(e); }
  }  // end of run()


private void wait(int duration)
{ try {
    Thread.sleep(duration); // in ms
  }
  catch (InterruptedException e)
  { e.printStackTrace(); }
}
```

Queue draining occurs every DELAY milliseconds, and the messages are moved over to a temporary list.

4.7. Analyzing the Messages

There are many ways of analyzing the messages, but SoundGen only looks for repeats, replacing them with a single message.

For example, assume there are five C4 note messages in the middle of the list:

```
..."$s push" "$n C4" "$n C4" "$n C4" "$n C4" "$n C4" "$c zap 500"...
```

The five messages are reduced to one, with the number of repeats appended to it:

```
..."$s push" "$n C4 5" "$c zap 500"...
```

This optimization is carried out frequently because of the way that note messages denote keywords, including iterative keywords such as "while" and "for". If a loop is repeatedly executing, then the queue will quickly become lengthy.

There's no reliable way to simplify the repetition information by static analysis since the number of loop iterations will often depend on run-time values, such as user input; message reduction must be carried out dynamically.

combineMsgs() looks at each message in the list, and removes any duplicates that follow it.

```
private void combineMsgs(ArrayList<String> msgs)
{
  int totalRemoved = 0;
  for(int i=0; i < msgs.size(); i++)
    totalRemoved += removeDups(i, msgs); //msgs list may get smaller
  System.out.println("Total removed: " + totalRemoved);
}


private int removeDups(int i, ArrayList<String> msgs)
/* Remove any duplicates of the message at position i, starting
   from position i+1 in the list. Add the number of repetitions
   to the message. */
{
  String currMsg = msgs.get(i);
  int pos = i+1;

  int numDups = 0;
  while ((pos+numDups) < msgs.size()) {
    if (!currMsg.equals( msgs.get(pos+numDups) ))
      break;
    numDups++;
  }

  if (numDups > 0) {
    System.out.println("Removing " + numDups + " duplicates" );
    int count = 0;
    while (count != numDups) {
      msgs.remove(pos);
      count++;
    }

    // add no of duplicates to end of current message at i
    msgs.set(i, currMsg + (" " + (numDups+1)) );
  }

  return numDups;
}  // end of removeDups()
```


A basic drawback of this approach is that the optimizations are time-dependent. The list contents depends on how often the message queue is drained, and this affects the number of duplicated messages detected and removed..

removeDups() checks if a single message appears multiple times, one after another. A more complicated optimization would be to search for repeating *sequences* of messages. For instance, assume there's a repeating sequence containing a "push", note playing, and a 'zap' clip:

```
.."$s push" "$n C4 5" "$c zap 500" "$s push" "$n C4 5" "$c zap 500"..
```

The repeating three-message sequence could be reduced to something like:

```
.. "[" "$s push" "$n C4 5" "$c zap 500" "]2" ..
```

The "[" and "]2" denotes that the sequence inside the brackets appeared twice in the original list. This requires more complicated pattern matching and parsing, which I haven't implemented.

4.8. Processing a Message

Processing a message is a three-way branch which converts a message to a spoken word, a musical note, or a sound clip.

```
// global
private Speaker speaker;


private void processMsg(String msg)
{
  System.out.println("--> " + msg);
  if (msg.startsWith("$w "))          // speak a word
    speaker.say( msg.substring(3) );
  else if (msg.startsWith("$n "))     // play a note
    processNote( msg.substring(3) );
  else if (msg.startsWith("$c "))     // play a clip
    processClip( msg.substring(3) );
  else
    System.out.println("Could not understand " + msg);
}  // end of processMsg()
```

The string in the "$w" message is passed to the Speaker object to be spoken.

Processing a note involves the pulling apart of the "$n" data, which takes the form "<note to be played> [< number of repeats >]" (e.g. "C4 2"), where the number of repeats is optional.

```
// globals
private static final int NOTE_DURATION = 800;
                     // fixed duration for playing a note
private static final int MAX_REPEATS = 3;
                     // max multiple for playing repeated notes

private NotesPlayer notesPlayer;
```

```
private void processNote(String msg)
{
  String[] toks = msg.split("\\s+");
  int numTimes = 0;
  if (toks.length == 2) {   // there is a number
    try {
      numTimes = Integer.parseInt(toks[1]);
    }
    catch (Exception e)
    { System.out.println("Number is incorrect for " + toks[1]);  }
  }

  if (numTimes != 0) {  // play a note repeatedly, and more quickly
    if (numTimes > MAX_REPEATS)
      numTimes = MAX_REPEATS;
    for (int i=0; i < numTimes; i++)
      notesPlayer.play(toks[0], NOTE_DURATION/2);
  }
  else                  // play a single note
    notesPlayer.play(toks[0], NOTE_DURATION);
}  // end of processNote()
```

It's irritating if a note is played too many times in succession. So, I've fixed it that a note can only be played at most MAX_REPEATS times, and at increased speed (by halving its duration).

Processing a clip involves a similar sort of parsing as for a note. "$c" data has the form "<wav filename (a clip) to be played> <duration(ms)> [< number of repeats >]" (e.g. "zap 500 2"), where the number of repetitions is optional.

```
// globals
private static final int CLIP_DURATION = 2000;
                          // fixed duration for playing a clip

private boolean progFinished = false;
private ClipsPlayer clipsPlayer;


private void processClip(String msg)
{
  String[] toks = msg.split("\\s+");

  if (toks[0].equals("keyDrop"))
    /* denotes end of main() in the tracing program, which means the
       end of tracing, and of sound generation */
    progFinished = true;

  // get duration
  int duration = 0;
  if (toks.length >= 2) {   // there is a duration
    try {
      duration = Integer.parseInt(toks[1]);
    }
    catch (Exception e)
    { System.out.println("Duration is incorrect for " + toks[1]);  }
```



```
  }
  if (duration == 0)    // problem with duration, so use a guess
    duration = CLIP_DURATION;


  // get number of times
  int numTimes = 1;
  if (toks.length == 3) {   // there is a number
    try {
      numTimes = Integer.parseInt(toks[2]);
    }
    catch (Exception e)
    { System.out.println("Repeats is incorrect for " + toks[2]);  }
  }

  // play a clip repeatedly
  for (int i=0; i < numTimes; i++)
    clipsPlayer.play(toks[0], duration);
}  // end of processClip()
```

If a clip's duration is incorrectly parsed, then it's assumed to be 2 seconds long.

When the "keyDrop" clip turns up, it means that the application's main() method is exiting, and so the global boolean progFinished is set to true. This will cause the processing loop in run() to terminate, allowing the thread to finish.