Java Sound Programmer Guide

Introductory Material

Part I: Sampled Audio

Chapter 2: Overview of the Sampled Package

Chapter 3: Accessing Audio System Resources

Chapter 7: Using Files and Format Converters

Part II: MIDI

Chapter 8: Overview of the MIDI Package

Chapter 12: Synthesizing Sound

Part III: Service Provider Interfaces

Preface

For Whom This Guide Is Intended

This guide is intended for three groups of readers:

- Application developers: software programmers who want to write Java™ applications or applets using audio or MIDI. Most readers will fall into this category.
- Service providers: developers of software modules that extend the capabilities of an implementation of the Java Sound application programming interface (API). For example, a vendor might provide a new audio mixer or MIDI synthesizer, or the ability to read and write a new file format. The Java Sound API is designed to let programs automatically access all such "plug-in" modules available on a particular system.
- API implementors: developers creating new implementations of the Java Sound API.

It is assumed that the reader has a basic knowledge of programming in the Java language. Familiarity with audio and MIDI is helpful but not assumed.

What This Guide Describes

This is a largely conceptual description of the Java Sound API, with some code snippets as programming examples. The Java Sound API specifies a software layer that allows application programs to communicate with an audio and MIDI engine. The Java Sound API is part of the Java™ 2 Platform, Standard Edition (J2SE), version 1.3, which is the version described by this guide. The sound API is included in both the Java 2 Software Development Kit (SDK), Standard Edition, and the Java 2 Runtime Environment, Standard Edition. Earlier implementations of the Java Sound API were supplied as a separate products and their programming interfacea differ from the one described here.

Note:

This guide is not a description of any particular implementation of the Java Sound API. In particular, it does not specifically describe the reference implementation of the Java Sound API created by Sun Microsystems, Inc. For example, you will not find here a list of exactly which sound file formats are supported in the reference implementation. (Because the Java Sound API makes "plug-in" services possible, API implementors and/or third parties can add support for new formats, ports with special features, etc.) In general, this guide ignores unique features, extensions, limitations, or bugs of a particular implementation.

As a convenience for developers, it does make note of some current limitations of Sun's reference implemenation. If you are having problems, it is highly recommended that you consult:

- Sun's Bug Database in the Java Developer Connection
- Java Sound Home Page
- Java Sound Reference Implementation Notes

For More Information

See the links to the Java Sound API reference documentation at http://java.sun.com/products/jdk/1.4/docs/guide/sound/. This Web site also includes a brief description of the reference implementation, as well as links to other resources about the Java Sound API, such as demo programs, mailing lists, and Frequently Answered Questions (FAQs). Also see the Java Sound Home Page mentioned above.

Chapter 1: Introduction to the Java Sound API

Design Goals

The Java™ Sound API is a low-level API for effecting and controlling the input and output of sound media, including both audio and Musical Instrument Digital Interface (MIDI) data. The Java Sound API provides explicit control over the capabilities normally required for sound input and output, in a framework that promotes extensibility and flexibility.

Who is the Java Sound API For?

Because sound is so fundamental, the Java Sound API fulfills the needs of a wide range of application developers. Potential application areas include:

- Communication frameworks, such as conferencing and telephony
- End-user content delivery systems, such as media players and music using streamed content
- Interactive application programs, such as games and Web sites that use dynamic content
- Content creation and editing
- Tools, toolkits, and utilities

How Does the Java Sound API Relate to Other Interfaces?

The Java Sound API provides the lowest level of sound support on the Java platform. It provides application programs with a great amount of control over sound operations, and it is extensible. For example, the Java Sound API supplies mechanisms for installing, accessing, and manipulating system resources such as audio mixers, MIDI synthesizers, other audio or MIDI devices, file readers and writers, and sound format converters. The Java Sound API does not include sophisticated sound editors or graphical tools, but it provides capabilities upon which such programs can be built. It emphasizes low-level control beyond that commonly expected by the end user.

There are other Java platform APIs that have sound-related elements. The Java Media Framework (JMF) is a higher-level API that is currently available as a Standard Extension to the Java platform. JMF specifies a unified architecture, messaging protocol, and programming interface for capturing and playing back time-based media. JMF provides a simpler solution for basic media-player application programs, and it enables synchronization between different media types, such as audio and video. On the other hand, programs that focus on sound can benefit from the Java Sound API, especially if they require more advanced features, such as the ability to carefully control buffered audio playback or directly

manipulate a MIDI synthesizer. Other Java APIs with sound aspects include Java 3D and APIs for telephony and speech. An implementation of any of these APIs might use an implementation of the Java Sound API internally, but is not required to do so.

Packages

The Java Sound API includes support for both digital audio and MIDI data. These two major modules of functionality are provided in separate packages:

- `javax.sound.sampled`

  This package specifies interfaces for capture, mixing, and playback of digital (sampled) audio.
- `javax.sound.midi`

  This package provides interfaces for MIDI synthesis, sequencing, and event transport.

Two other packages permit service providers (as opposed to application developers) to create custom software components that extend the capabilities of an implementation of the Java Sound API:

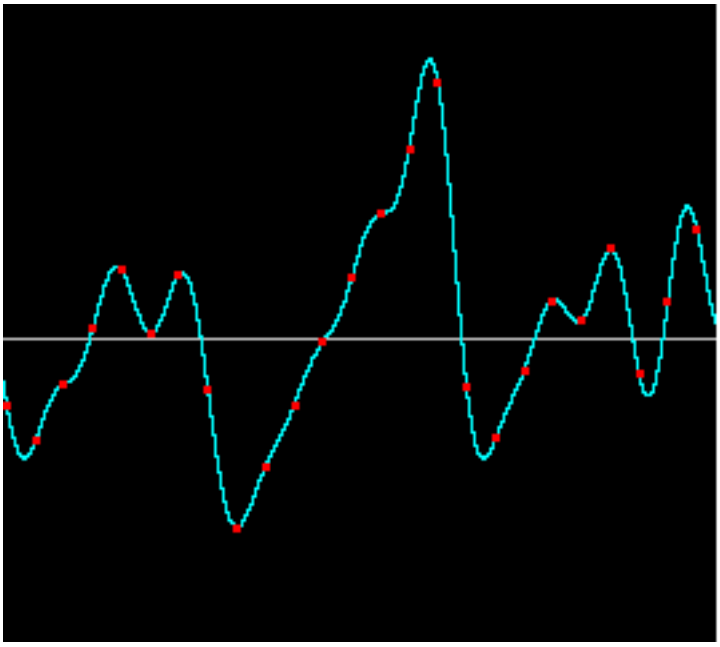
- `javax.sound.sampled.spi`
- `javax.sound.midi.spi`

The rest of this chapter briefly discusses the sampled-audio system, the MIDI system, and the SPI packages. Each of these is then discussed in detail in a subsequent part of the guide.

Sampled Audio

What Is Sampled Audio?

The `javax.sound.sampled` package handles digital audio data, which the Java Sound API refers to as sampled audio. *Samples* are successive snapshots of a signal. In the case of audio, the signal is a sound wave. A microphone converts the acoustic signal into a corresponding analog electrical signal, and an analog-to-digital converter transforms that analog signal into a sampled digital form. The following figure shows a brief moment in a sound recording.



A Sampled Sound Wave

This graph plots sound pressure (amplitude) on the vertical axis, and time on the horizontal axis. The amplitude of the analog sound wave is measured periodically at a certain rate, resulting in the discrete samples (the red data points in the figure) that comprise the digital audio signal. The center horizontal line indicates zero amplitude; points above the line are positive-valued samples, and points below are negative. The accuracy of the digital approximation of the analog signal depends on its resolution in time (the *sampling rate*) and its *quantization*, or resolution in amplitude (the number of bits used to represent each sample). As a point of reference, the audio recorded for storage on compact discs is sampled 44,100 times per second and represented with 16 bits per sample.

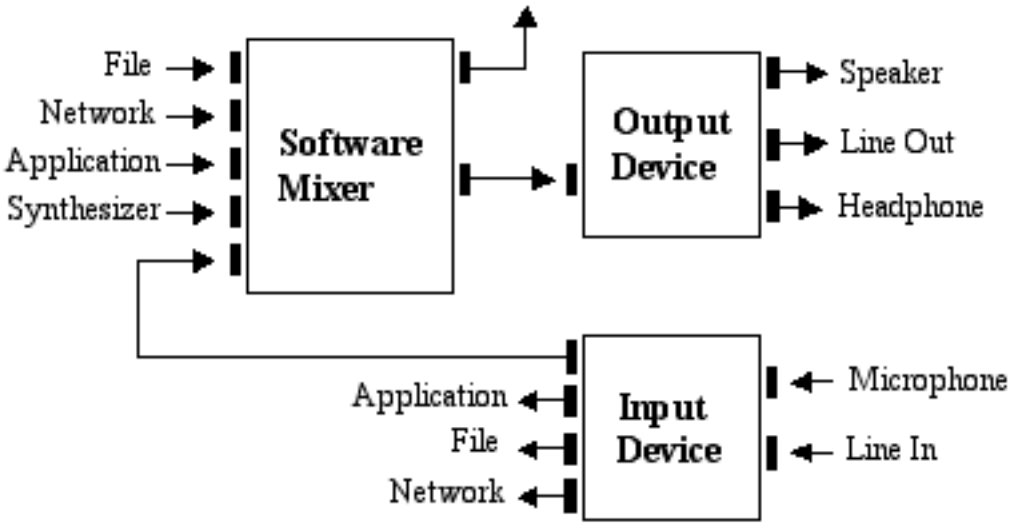
The term "sampled audio" is used here slightly loosely. A sound wave could be sampled at discrete intervals while being left in an analog form. For purposes of the Java Sound API, however, "sampled audio" is equivalent to "digital audio."

Typically, sampled audio on a computer comes from a sound recording, but the sound could instead be synthetically generated (for example, to create the sounds of a touch-tone telephone). The term "sampled audio" refers to the type of data, not its origin.

Further information about the structure of digital audio data is given under "What Is Formatted Audio Data?" in Chapter 2, "Overview of the Sampled Package."

Audio Configurations

The Java Sound API does not assume a specific audio hardware configuration; it is designed to allow different sorts of audio components to be installed on a system and accessed by the API. The Java Sound API supports common functionality such as input and output from a sound card (for example, for recording and playback of sound files) as well as mixing of multiple streams of audio. Here is one example of a typical audio architecture:



A Typical Audio Architecture

In this example, a device such as a sound card has various input and output ports, and mixing is provided in the software. The mixer might receive data that has been read from a file, streamed from a network, generated on the fly by an application program, or produced by a MIDI synthesizer. (The `javax.sound.midi` package, discussed next, supplies a Java language interface for synthesizers.) The mixer combines all its audio inputs into a single stream, which can be sent to an output device for rendering.

MIDI

The `javax.sound.midi` package contains APIs for transporting and sequencing MIDI events, and for synthesizing sound from those events.

What Is MIDI?

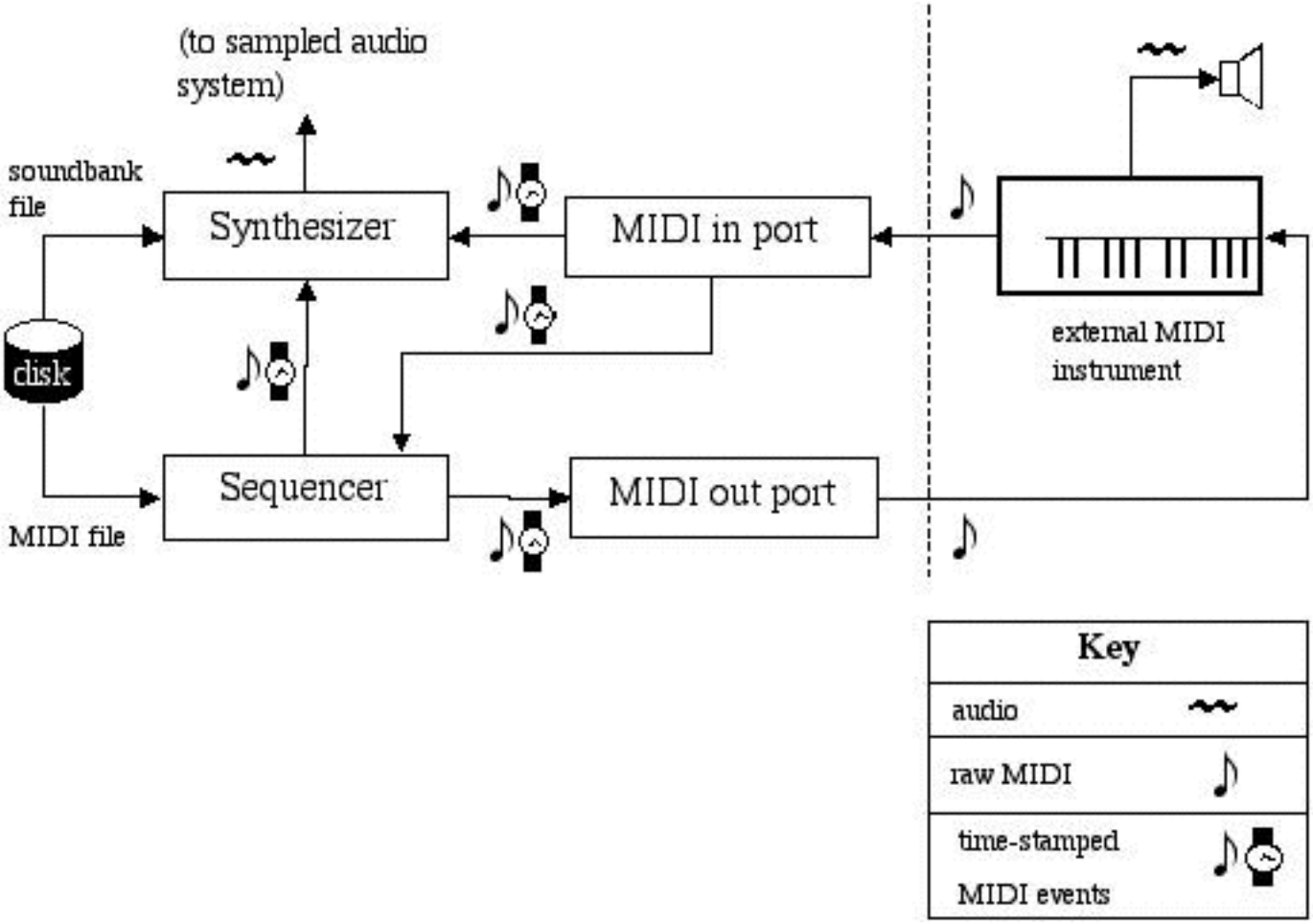
Whereas sampled audio is a direct representation of a sound itself, MIDI data can be thought of as a recipe for creating a sound, especially a musical sound. MIDI data, unlike audio data, does not describe sound directly. Instead, it describes events that affect the sound a synthesizer is making. MIDI data is analogous to a graphical user interface's keyboard and mouse events. In the case of MIDI, the events can be thought of as actions upon a musical keyboard, along with actions on various pedals, sliders, switches, and knobs on that musical instrument. These events need not actually originate with a hardware musical instrument; they can be simulated in software, and they can be stored in MIDI files. A program that can create, edit, and perform these files is called a sequencer. Many computer sound cards include MIDI-controllable music synthesizer chips to which sequencers can send their MIDI events. Synthesizers can also be implemented entirely in software. The synthesizers interpret the MIDI events that they receive and produce audio output. Usually the sound synthesized from MIDI data is musical sound (as opposed to speech, for example). MIDI synthesizers are also capable of generating various kinds of sound effects.

Some sound cards include MIDI input and output ports to which external MIDI hardware devices (such as keyboard synthesizers or other instruments) can be connected. From a MIDI input port, an application program can receive events generated by an external MIDI-equipped musical instrument. The program might play the musical performance using the computer's internal synthesizer, save it to disk as a MIDI file, or render it into musical notation. A program might use a MIDI output port to play an external instrument, or to control other external devices such as recording equipment.

More information about MIDI data is given in Chapter 8, "Overview of the MIDI Package," particularly in the section "A MIDI Refresher: Wires and Files."

MIDI Configurations

The diagram below illustrates the functional relationships between the major components in a possible MIDI configuration based on the Java Sound API. (As with audio, the Java Sound API permits a variety of MIDI software devices to be installed and interconnected. The system shown here is one potential scenario.) The flow of data between components is indicated by arrows. The data can be in a standard file format, or (as indicated by the key in the lower right corner of the diagram), it can be audio, raw MIDI bytes, or time-tagged MIDI messages.



A Possible MIDI Configuration

In this example, the application program prepares a musical performance by loading a musical score that's stored as a standard MIDI file on a disk (left side of the diagram). Standard MIDI files contain tracks, each of which is a list of time-tagged MIDI events. Most of the events represent musical notes

(pitches and rhythms). This MIDI file is read and then "performed" by a software sequencer. A sequencer performs its music by sending MIDI messages to some other device, such as an internal or external synthesizer. The synthesizer itself may read a soundbank file containing instructions for emulating the sounds of certain musical instruments. If not, the synthesizer will play the notes stored in the MIDI file using whatever instrument sounds are already loaded into the synthesizer.

As illustrated, the MIDI events must be translated into raw (non-time-tagged) MIDI before being sent through a MIDI output port to an external synthesizer. Similarly, raw MIDI data coming into the computer from an external MIDI source (a keyboard instrument, in the diagram) is translated into time-tagged MIDI messages that can control a synthesizer, or that a sequencer can store for later use. All these aspects of MIDI data flow are explained in detail in the subsequent chapters on MIDI (see Part II of this guide).

Service Provider Interfaces

The `javax.sound.sampled.spi` and `javax.sound.midi.spi` packages contain APIs that let software developers create new audio or MIDI resources that can be provided separately to the user and "plugged in" to an existing implementation of the Java Sound API. Here are some examples of services (resources) that can be added in this way:

- An audio mixer
- A MIDI synthesizer
- A file parser that can read or write a new type of audio or MIDI file
- A converter that translates between different sound data formats

In some cases, services are software interfaces to the capabilities of hardware devices, such as sound cards, and the service provider might be the same as the vendor of the hardware. In other cases, the services exist purely in software. For example, a synthesizer or a mixer could be an interface to a chip on a sound card, or it could be implemented without any hardware support at all.

An implementation of the Java Sound API contains a basic set of services, but the service provider interface (SPI) packages allow third parties to create new services. These third-party services are integrated into the system in the same way as the built-in services. The `AudioSystem` class in the `sampled` package and the `MidiSystem` class in the `midi` package act as coordinators that let application programs access the services explicitly or implicitly. Often the existence of a service is completely transparent to an application program that uses it. The service-provider mechanism benefits users of application programs based on the Java Sound API, because new sound features can be added to a program without requiring a new release of the Java SDK or runtime environment, and, in many cases, without even requiring a new release of the application program itself.

Chapter 2: Overview of the Sampled Package

This chapter provides an introduction to the Java™ Sound API's digital audio architecture, which is accessed through the `javax.sound.sampled` package. First, an explanation is given of the package's focus: playback and capture of formatted audio data. Then this chapter describes the three fundamental components required for playback or capture: an audio data format, a line, and a mixer. The `Line` interface and its subinterfaces are introduced briefly.

Design Goals

Before examining the elements of the Java Sound API, it helps to understand the orientation of the `javax.sound.sampled` package.

A Focus on Data Transport

The `javax.sound.sampled` package is fundamentally concerned with audio transport—in other words, the Java Sound API focuses on playback and capture. The central task that the Java Sound API addresses is how to move bytes of formatted audio data into and out of the system. This task involves opening audio input and output devices and managing buffers that get filled with real-time sound data. It can also involve mixing multiple streams of audio into one stream (whether for input or output). The transport of sound into or out of the system has to be correctly handled when the user requests that the flow of sound be started, paused, resumed, or stopped.

To support this focus on basic audio input and output, the Java Sound API provides methods for converting between various audio data formats, and for reading and writing common types of sound files. However, it does not attempt to be a comprehensive sound-file toolkit. A particular implementation of the Java Sound API need not support an extensive set of file types or data format conversions. Third-party service providers can supply modules that "plug in" to an existing implementation to support additional file types and conversions.

Buffered and Unbuffered Handling of Audio

The Java Sound API can handle audio transport in both a streaming, buffered fashion and an in-memory, unbuffered fashion. "Streaming" is used here in a general sense to refer to real-time handling of audio bytes; it does not refer to the specific, well-known case of sending audio over the Internet in a certain format. In other words, a stream of audio is simply a continuous set of audio bytes that arrive more or less at the same rate that they are to be handled (played, recorded, etc.). Operations on the bytes

commence before all the data has arrived. In the streaming model, particularly in the case of audio input rather than audio output, you do not necessarily know in advance how long the sound is and when it will finish arriving. You simply handle one buffer of audio data at a time, until the operation is halted. In the case of audio output (playback), you also need to buffer data if the sound you want to play is too large to fit in memory all at once. In other words, you deliver your audio bytes to the sound engine in chunks, and it takes care of playing each sample at the right time. Mechanisms are provided that make it easy to know how much data to deliver in each chunk.

The Java Sound API also permits unbuffered transport in the case of playback only, assuming you already have all the audio data at hand and it is not too large to fit in memory. In this situation, there is no need for the application program to buffer the audio, although the buffered, real-time approach is still available if desired. Instead, the entire sound can be preloaded at once into memory for subsequent playback. Because all the sound data is loaded in advance, playback can start immediately—for example, as soon as the user clicks a Start button. This can be an advantage compared to the buffered model, where the playback has to wait for the first buffer to fill. In addition, the in-memory, unbuffered model allows sounds to be easily looped (cycled) or set to arbitrary positions in the data.

These two models for playback are discussed further in Chapter 4, "Playing Back Audio." Buffered recording is discussed in Chapter 5, "Capturing Audio."

The Essentials: Format, Mixer, and Line

To play or capture sound using the Java Sound API, you need at least three things: formatted audio data, a mixer, and a line. Each of these is explained below.

What Is Formatted Audio Data?

Formatted audio data refers to sound in any of a number of standard formats. The Java Sound API distinguishes between *data formats* and *file formats*.

Data Formats

A data format tells you how to interpret a series of bytes of "raw" sampled audio data, such as samples that have already been read from a sound file, or samples that have been captured from the microphone input. You might need to know, for example, how many bits constitute one sample (the representation of the shortest instant of sound), and similarly you might need to know the sound's sample rate (how fast the samples are supposed to follow one another). When setting up for playback or capture, you specify the data format of the sound you are capturing or playing.

In the Java Sound API, a data format is represented by an `AudioFormat` object, which includes the following attributes:

- Encoding technique, usually pulse code modulation (PCM)
- Number of channels (1 for mono, 2 for stereo, etc.)
- Sample rate (number of samples per second, per channel)
- Number of bits per sample (per channel)
- Frame rate

- Frame size in bytes
- Byte order (big-endian or little-endian)

PCM is one kind of encoding of the sound waveform. The Java Sound API includes two PCM encodings that use linear quantization of amplitude, and signed or unsigned integer values. Linear quantization means that the number stored in each sample is directly proportional (except for any distortion) to the original sound pressure at that instant—and similarly proportional to the displacement of a loudspeaker or eardrum that is vibrating with the sound at that instant. Compact discs, for example, use linear PCM-encoded sound. Mu-law encoding and a-law encoding are common nonlinear encodings that provide a more compressed version of the audio data; these encodings are typically used for telephony or recordings of speech. A nonlinear encoding maps the original sound's amplitude to the stored value using a nonlinear function, which can be designed to give more amplitude resolution to quiet sounds than to loud sounds.

A frame contains the data for all channels at a particular time. For PCM-encoded data, the frame is simply the set of simultaneous samples in all channels, for a given instant in time, without any additional information. In this case, the frame rate is equal to the sample rate, and the frame size in bytes is the number of channels multiplied by the sample size in bits, divided by the number of bits in a byte.

For other kinds of encodings, a frame might contain additional information besides the samples, and the frame rate might be completely different from the sample rate. For example, consider the MP3 (MPEG-1 Audio Layer 3) encoding, which is not explicitly mentioned in the current version of the Java Sound API, but which could be supported by an implementation of the Java Sound API or by a third-party service provider. In MP3, each frame contains a bundle of compressed data for a series of samples, not just one sample per channel. Because each frame encapsulates a whole series of samples, the frame rate is slower than the sample rate. The frame also contains a header. Despite the header, the frame size in bytes is less than the size in bytes of the equivalent number of PCM frames. (After all, the purpose of MP3 is to be more compact than PCM data.) For such an encoding, the sample rate and sample size refer to the PCM data that the encoded sound will eventually be converted into before being delivered to a digital-to-analog converter (DAC).

File Formats

A file format specifies the structure of a sound file, including not only the format of the raw audio data in the file, but also other information that can be stored in the file. Sound files come in various standard varieties, such as WAVE (also known as WAV, and often associated with PCs), AIFF (often associated with Macintoshes), and AU (often associated with UNIX systems). The different types of sound file have different structures. For example, they might have a different arrangement of data in the file's "header." A header contains descriptive information that typically precedes the file's actual audio samples, although some file formats allow successive "chunks" of descriptive and audio data. The header includes a specification of the data format that was used for storing the audio in the sound file. Any of these types of sound file can contain various data formats (although usually there is only one data format within a given file), and the same data format can be used in files that have different file formats.

In the Java Sound API, a file format is represented by an `AudioFileFormat` object, which contains:

- The file type (WAVE, AIFF, etc.)
- The file's length in bytes

- The length, in frames, of the audio data contained in the file
- An AudioFormat object that specifies the data format of the audio data contained in the file

The `AudioSystem` class (described in Chapter 3, "Accessing Audio System Resources") provides methods for reading and writing sounds in different file formats, and for converting between different data formats. Some of the methods let you access a file's contents through a kind of stream called an `AudioInputStream`. An `AudioInputStream` is a subclass of the generic Java `InputStream` class, which encapsulates a series of bytes that can be read sequentially. To its superclass, the `AudioInputStream` class adds knowledge of the bytes' audio data format (represented by an `AudioFormat` object). By reading a sound file as an `AudioInputStream`, you get immediate access to the samples, without having to worry about the sound file's structure (its header, chunks, etc.). A single method invocation gives you all the information you need about the data format and the file type.

What Is a Mixer?

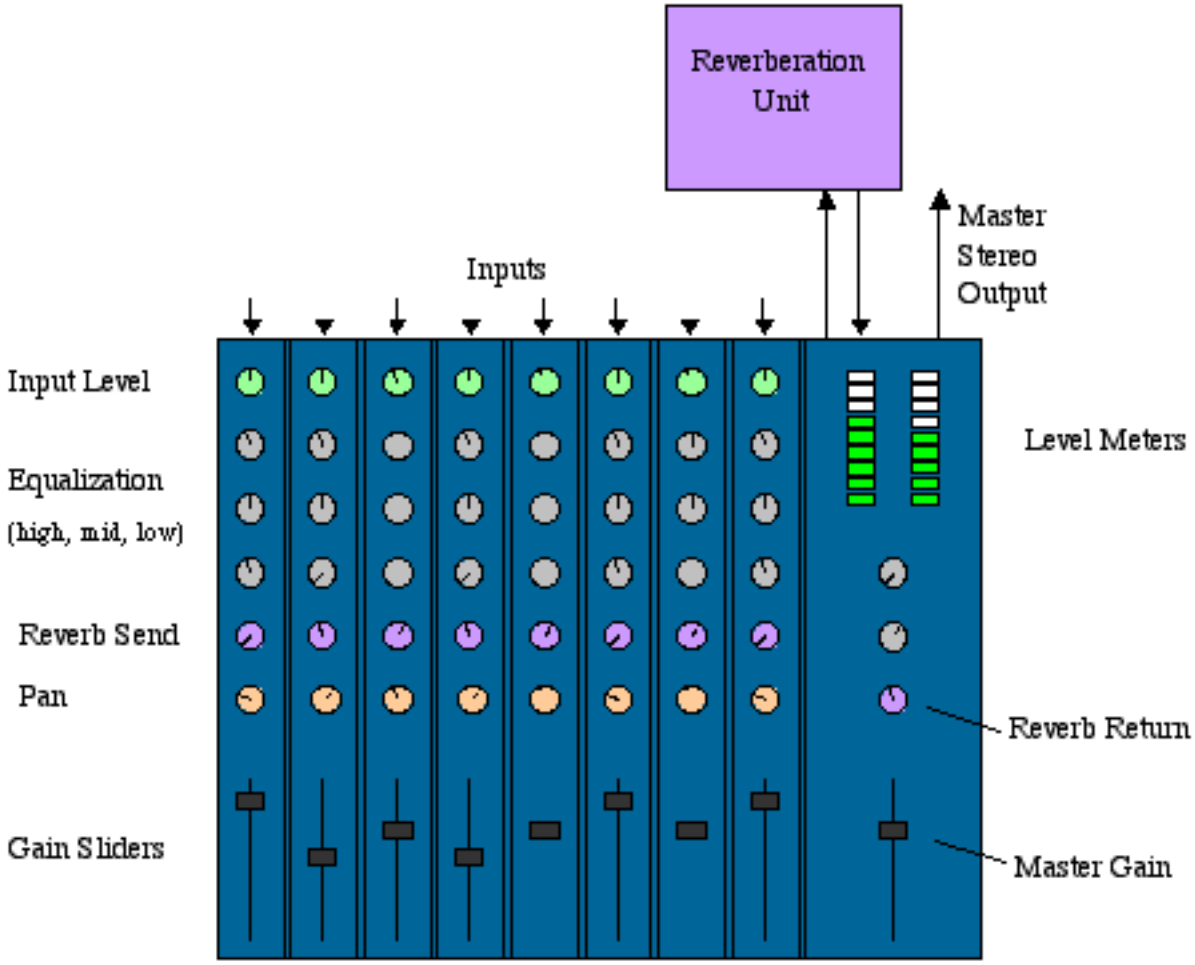
Many application programming interfaces (APIs) for sound make use of the notion of an audio *device*. A device is often a software interface to a physical input/output device. For example, a sound-input device might represent the input capabilities of a sound card, including a microphone input, a line-level analog input, and perhaps a digital audio input.

In the Java Sound API, devices are represented by `Mixer` objects. The purpose of a mixer is to handle one or more streams of audio input and one or more streams of audio output. In the typical case, it actually mixes together multiple incoming streams into one outgoing stream. A `Mixer` object can represent the sound-mixing capabilities of a physical device such as a sound card, which might need to mix the sound coming in to the computer from various inputs, or the sound coming from application programs and going to outputs.

Alternatively, a `Mixer` object can represent sound-mixing capabilities that are implemented entirely in software, without any inherent interface to physical devices.

In the Java Sound API, a component such as the microphone input on a sound card is not itself considered a device—that is, a mixer—but rather a *port* into or out of the mixer. A port typically provides a single stream of audio into or out of the mixer (although the stream can be multichannel, such as stereo). The mixer might have several such ports. For example, a mixer representing a sound card's output capabilities might mix several streams of audio together, and then send the mixed signal to any or all of various output ports connected to the mixer. These output ports could be (for example) a headphone jack, a built-in speaker, or a line-level output.

To understand the notion of a mixer in the Java Sound API, it helps to visualize a physical mixing console, such as those used in live concerts and recording studios. (See illustration that follows.)



A Physical Mixing Console

A physical mixer has "strips" (also called "slices"), each representing a path through which a single audio signal goes into the mixer for processing. The strip has knobs and other controls by which you can control the volume and pan (placement in the stereo image) for the signal in that strip. Also, the mixer might have a separate bus for effects such as reverb, and this bus can be connected to an internal or external reverberation unit. Each strip has a potentiometer that controls how much of that strip's signal goes into the reverberated mix. The reverberated ("wet") mix is then mixed with the "dry" signals from the strips. A physical mixer sends this final mixture to an output bus, which typically goes to a tape recorder (or disk-based recording system) and/or speakers.

Imagine a live concert that is being recorded in stereo. Cables (or wireless connections) coming from the many microphones and electric instruments on stage are plugged into the inputs of the mixing console. Each input goes to a separate strip of the mixer, as illustrated. The sound engineer decides on the settings of the gain, pan, and reverb controls. The output of all the strips and the reverb unit are mixed together into two channels. These two channels go to two outputs on the mixer, into which cables are plugged that connect to the stereo tape recorder's inputs. The two channels are perhaps also sent via an amplifier to speakers in the hall, depending on the type of music and the size of the hall.

Now imagine a recording studio, in which each instrument or singer is recorded to a separate track of a multitrack tape recorder. After the instruments and singers have all been recorded, the recording engineer performs a "mixdown" to combine all the taped tracks into a two-channel (stereo) recording that can be

distributed on compact discs. In this case, the input to each of the mixer's strips is not a microphone, but one track of the multitrack recording. Once again, the engineer can use controls on the strips to decide each track's volume, pan, and reverb amount. The mixer's outputs go once again to a stereo recorder and to stereo speakers, as in the example of the live concert.

These two examples illustrate two different uses of a mixer: to capture multiple input channels, combine them into fewer tracks, and save the mixture, or to play back multiple tracks while mixing them down to fewer tracks.

In the Java Sound API, a mixer can similarly be used for input (capturing audio) or output (playing back audio). In the case of input, the *source* from which the mixer gets audio for mixing is one or more input ports. The mixer sends the captured and mixed audio streams to its *target*, which is an object with a buffer from which an application program can retrieve this mixed audio data. In the case of audio output, the situation is reversed. The mixer's source for audio is one or more objects containing buffers into which one or more application programs write their sound data; and the mixer's target is one or more output ports.

What Is a Line?

The metaphor of a physical mixing console is also useful for understanding the Java Sound API's concept of a *line*.

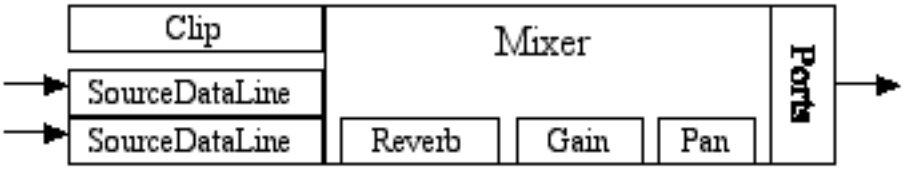
A line is an element of the digital audio "pipeline"—that is, a path for moving audio into or out of the system. Usually the line is a path into or out of a mixer (although technically the mixer itself is also a kind of line).

Audio input and output ports are lines. These are analogous to the microphones and speakers connected to a physical mixing console. Another kind of line is a data path through which an application program can get input audio from, or send output audio to, a mixer. These data paths are analogous to the tracks of the multitrack recorder connected to the physical mixing console.

One difference between lines in the Java Sound API and those of a physical mixer is that the audio data flowing through a line in the Java Sound API can be mono or multichannel (for example, stereo). By contrast, each of a physical mixer's inputs and outputs is typically a single channel of sound. To get two or more channels of output from the physical mixer, two or more physical outputs are normally used (at least in the case of analog sound; a digital output jack is often multichannel). In the Java Sound API, the number of channels in a line is specified by the `AudioFormat` of the data that is currently flowing through the line.

Lines in an Audio-output Configuration

Let's now examine some specific kinds of lines and mixers. The following diagram shows different types of lines in a simple audio-output system that could be part of an implementation of the Java Sound API:



A Possible Configuration of Lines for Audio Output

In this example, an application program has gotten access to some available inputs of an audio-input mixer: one or more *clips* and *source data lines*. A clip is a mixer input (a kind of line) into which you can load audio data prior to playback; a source data line is a mixer input that accepts a real-time stream of audio data. The application program preloads audio data from a sound file into the clips. It then pushes other audio data into the source data lines, a buffer at a time. The mixer reads data from all these lines, each of which may have its own reverberation, gain, and pan controls, and mixes the dry audio signals with the wet (reverberated) mix. The mixer delivers its final output to one or more output ports, such as a speaker, a headphone jack, and a line-out jack.

Although the various lines are depicted as separate rectangles in the diagram, they are all "owned" by the mixer, and can be considered integral parts of the mixer. The reverb, gain, and pan rectangles represent processing controls (rather than lines) that can be applied by the mixer to data flowing through the lines.

Note that this is just one example of a possible mixer that is supported by the API. Not all audio configurations will have all the features illustrated. An individual source data line might not support panning, a mixer might not implement reverb, and so on.

Lines in an Audio-input Configuration

A simple audio-input system might be similar:

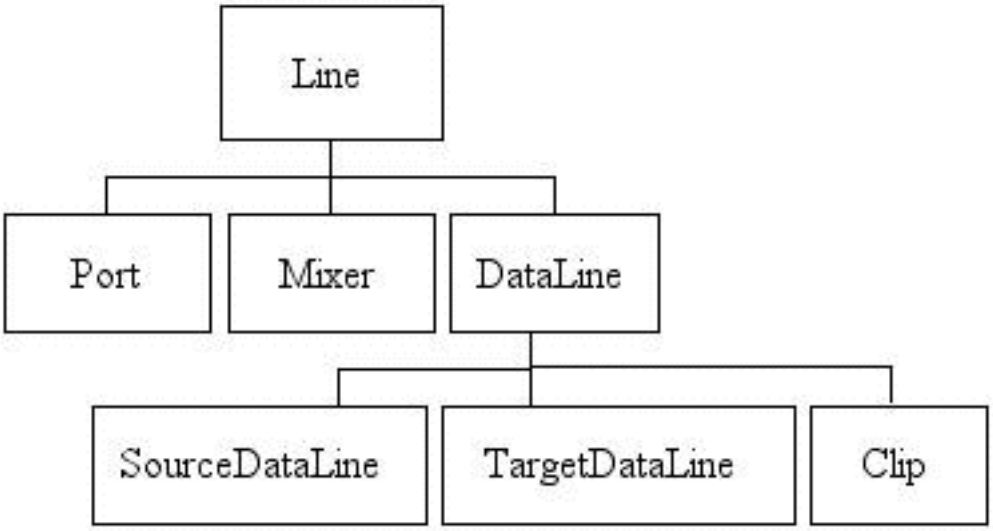
A Possible Configuration of Lines for Audio Input

Here, data flows into the mixer from one or more input ports, commonly the microphone or the line-in jack. Gain and pan are applied, and the mixer delivers the captured data to an application program via the mixer's target data line. A target data line is a mixer output, containing the mixture of the streamed input sounds. The simplest mixer has just one target data line, but some mixers can deliver captured data to multiple target data lines simultaneously.

The Line Interface Hierarchy

Now that we've seen some functional pictures of what lines and mixers are, let's discuss them from a slightly more programmatic perspective. Several types of line are defined by subinterfaces of the basic

`Line` interface. The interface hierarchy is shown below.



The Line Interface Hierarchy

The base interface, `Line`, describes the minimal functionality common to all lines:

- Controls

  Data lines and ports often have a set of controls that affect the audio signal passing through the line. The Java Sound API specifies control classes that can be used to manipulate aspects of sound such as: gain (which affects the signal's volume in decibels), pan (which affects the sound's right-left positioning, reverb (which adds reverberation to the sound to emulate different kinds of room acoustics), and sample rate (which affects the rate of playback as well as the sound's pitch).

- Open or closed status

  Successful opening of a line guarantees that resources have been allocated to the line. A mixer has a finite number of lines, so at some point multiple application programs (or the same one) might vie for usage of the mixer's lines. Closing a line indicates that any resources used by the line may now be released.

- Events

  A line generates events when it opens or closes. Subinterfaces of `Line` can introduce other types of events. When a line generates an event, the event is sent to all objects that have registered to "listen" for events on that line. An application program can create these objects, register them to listen for line events, and react to the events as desired.

We'll now examine the subinterfaces of the `Line` interface.

`Ports` are simple lines for input or output of audio to or from audio devices. As mentioned earlier, some common types of ports are the microphone, line input, CD-ROM drive, speaker, headphone, and line output.

The `Mixer` interface represents a mixer, of course, which as we have seen represents either a hardware or a software device. The `Mixer` interface provides methods for obtaining a mixer's lines. These include source lines, which feed audio to the mixer, and target lines, to which the mixer delivers its mixed audio.

For an audio-input mixer, the source lines are input ports such as the microphone input, and the target lines are `TargetDataLines` (described below), which deliver audio to the application program. For an audio-output mixer, on the other hand, the source lines are `Clips` or `SourceDataLines` (described below), to which the application program feeds audio data, and the target lines are output ports such as the speaker.

A `Mixer` is defined as having one or more source lines and one or more target lines. Note that this definition means that a mixer need not actually mix data; it might have only a single source line. The `Mixer` API is intended to encompass a variety of devices, but the typical case supports mixing.

The `Mixer` interface supports synchronization; that is, you can specify that two or more of a mixer's lines be treated as a synchronized group. Then you can start, stop, or close all those data lines by sending a single message to any line in the group, instead of having to control each line individually. With a mixer that supports this feature, you can obtain sample-accurate synchronization between lines.

The generic `Line` interface does not provide a means to start and stop playback or recording. For that you need a data line. The `DataLine` interface supplies the following additional media-related features beyond those of a `Line`:

- Audio format

  Each data line has an audio format associated with its data stream.

- Media position

  A data line can report its current position in the media, expressed in sample frames. This represents the number of sample frames captured by or rendered from the data line since it was opened.

- Buffer size

  This is the size of the data line's internal buffer in bytes. For a source data line, the internal buffer is one to which data can be written, and for a target data line it's one from which data can be read.

- Level (the current amplitude of the audio signal)
- Start and stop playback or capture
- Pause and resume playback or capture
- Flush (discard unprocessed data from the queue)
- Drain (block until all unprocessed data has been drained from the queue, and the data line's buffer has become empty)
- Active status

  A data line is considered active if it is engaged in active presentation or capture of audio data to or from a mixer.

- Events

  `START` and `STOP` events are produced when active presentation or capture of data from or to the data line starts or stops.

A `TargetDataLine` receives audio data from a mixer. Commonly, the mixer has captured audio data from a port such as a microphone; it might process or mix this captured audio before placing the data in

the target data line's buffer. The `TargetDataLine` interface provides methods for reading the data from the target data line's buffer and for determining how much data is currently available for reading.

A `SourceDataLine` receives audio data for playback. It provides methods for writing data to the source data line's buffer for playback, and for determining how much data the line is prepared to receive without blocking.

A `Clip` is a data line into which audio data can be loaded prior to playback. Because the data is pre-loaded rather than streamed, the clip's duration is known before playback, and you can choose any starting position in the media. Clips can be looped, meaning that upon playback, all the data between two specified loop points will repeat a specified number of times, or indefinitely.

This chapter has introduced most of the important interfaces and classes of the sampled-audio API. Subsequent chapters show how you can access and use these objects in your application program.

Chapter 3: Accessing Audio System Resources

Notes:

1. For Linux, `Port` is currently not implemented in the Sun reference implementation of JavaSound. (See http://developer.java.sun.com/developer/bugParade/bugs/4384401.html.)
2. Looping a portion of a `Clip` from a start point to an end point doesn't work. (See http://developer.java.sun.com/developer/bugParade/bugs/4386052.html.)

The Java™ Sound API takes a flexible approach to system configuration. Different sorts of audio devices (mixers) can be installed on a computer. The API makes few assumptions about what devices have been installed and what their capabilities are. Instead, it provides ways for the system to report about the available audio components, and ways for your program to access them.

This section shows how your program can learn what sampled-audio resources have been installed on the computer, and how it can gain access to the available resources. Among other things, the resources include mixers and the various types of lines owned by the mixers.

The AudioSystem Class

The `AudioSystem` class acts as a clearinghouse for audio components, including built-in services and separately installed services from third-party providers. `AudioSystem` serves as an application program's entry point for accessing these installed sampled-audio resources. You can query the `AudioSystem` to learn what sorts of resources have been installed, and then you can obtain access to them. For example, an application program might start out by asking the `AudioSystem` class whether there is a mixer that has a certain configuration, such as one of the input or output configurations illustrated earlier in the discussion of lines. From the mixer, the program would then obtain data lines, and so on.

Here are some of the resources an application program can obtain from the `AudioSystem`:

- Mixers

  A system typically has multiple mixers installed. There is usually at least one for audio input and one for audio output. There might also be mixers that don't have I/O ports but instead accept audio from an application program and deliver the mixed audio back to the program. The `AudioSystem` class provides a list of all of the installed mixers.

- Lines

  Even though every line is associated with a mixer, an application program can get a line directly from the `AudioSystem`, without dealing explicitly with mixers.

- Format conversions

  An application program can use format conversions to translate audio data from one format to another. Conversions are described in Chapter 7, "Using Files and Format Converters."

- Files and streams

  The `AudioSystem` class provides methods for translating between audio files and audio streams. It can also report the file format of a sound file and can write files in different formats. These facilities are discussed in Chapter 7, "Using Files and Format Converters."

Information Objects

Several classes in the Java Sound API provide useful information about associated interfaces. For example, `Mixer.Info` provides details about an installed mixer, such as the mixer's vendor, name, description, and version. `Line.Info` obtains the class of a specific line. Subclasses of `Line.Info` include `Port.Info` and `DataLine.Info`, which obtain details relevant to a specific port and data line, respectively. Each of these classes is described further in the appropriate section below. It's important not to confuse the `Info` object with the mixer or line object that it describes.

Getting a Mixer

Usually, one of the first things a program that uses the Java Sound API needs to do is to obtain a mixer, or at least one line of a mixer, so that you can get sound into or out of the computer. Your program might need a specific kind of mixer, or you might want to display a list of all the available mixers so that the user can select one. In either case, you need to learn what kinds of mixers are installed. `AudioSystem` provides the following method:

```
static Mixer.Info[] getMixerInfo()
```

Each `Mixer.Info` object returned by this method identifies one type of mixer that is installed. (Usually a system has at most one mixer of a given type. If there happens to be more than one of a given type, the returned array still only has one `Mixer.Info` for that type.) An application program can iterate over the `Mixer.Info` objects to find an appropriate one, according to its needs. The `Mixer.Info` includes the following strings to identify the kind of mixer:

- Name
- Version
- Vendor
- Description

These are arbitrary strings, so an application program that needs a specific mixer must know what to expect and what to compare the strings to. The company that provides the mixer should include this information in its documentation. Alternatively, and perhaps more typically, the application program will

display all the `Mixer.Info` objects' strings to the user and let the user choose the corresponding mixer.

Once an appropriate mixer is found, the application program invokes the following `AudioSystem` method to obtain the desired mixer:

```
static Mixer getMixer(Mixer.Info info)
```

What if your program needs a mixer that has certain capabilities, but it doesn't need a specific mixer made by a specific vendor? And what if you can't depend on the user's knowing which mixer should be chosen? In that case, the information in the `Mixer.Info` objects won't be of much use. Instead, you can iterate over all the `Mixer.Info` objects returned by get`MixerInfo`, get a mixer for each by invoking `getMixer`, and query each mixer for its capabilities. For example, you might need a mixer that can write its mixed audio data to a certain number of target data lines simultaneously. In that case, you would query each mixer using this Mixer method:

```
int getMaxLines(Line.Info info)
```

Here, the `Line.Info` would specify a `TargetDataLine`. The `Line.Info` class is discussed in the next section.

Getting a Line of a Desired Type

There are two ways to get a line:

- Directly from the `AudioSystem` object
- From a mixer that you have already obtained from the `AudioSystem` object (see "Getting a Mixer," in this chapter)

Getting a Line Directly from the AudioSystem

Let's assume you haven't obtained a mixer, and your program is a simple one that really only needs a certain kind of line; the details of the mixer don't matter to you. You can use the `AudioSystem` method:

```
static Line getLine(Line.Info info)
```

which is analogous to the `getMixer` method discussed above. Unlike `Mixer.Info`, the `Line.Info` used as an argument doesn't store textual information to specify the desired line. Instead, it stores information about the class of line desired.

`Line.Info` is an abstract class, so you use one of its subclasses (`Port.Info` or `DataLine.Info`) to obtain a line. The following code excerpt uses the `DataLine.Info` subclass to obtain and open a target data line:

```
TargetDataLine line;
DataLine.Info info = new DataLine.Info(TargetDataLine.class,
    format); // format is an AudioFormat object
if (!AudioSystem.isLineSupported(info)) {
    // Handle the error.
    }
    // Obtain and open the line.
try {
```

```
        line = (TargetDataLine) AudioSystem.getLine(info);
        line.open(format);
    } catch (LineUnavailableException ex) {
            // Handle the error.
        //...
    }
```

This code obtains a `TargetDataLine` object without specifying any attributes other than its class and its audio format. You can use analogous code to obtain other kinds of lines. For a `SourceDataLine` or a `Clip`, just substitute that class for `TargetDataLine` as the class of the line variable, and also in the first argument to the `DataLine.Info` constructor.

For a `Port`, you can use static instances of `Port.Info`, in code like the following:

```
    if (AudioSystem.isLineSupported(Port.Info.MICROPHONE)) {
        try {
            line = (Port) AudioSystem.getLine(
                Port.Info.MICROPHONE);
        }
    }
```

Note the use of the method `isLineSupported` to see whether the mixer even has a line of the desired type.

Recall that a source line is an input to a mixer—namely, a `Port` object if the mixer represents an audio-input device, and a `SourceDataLine` or `Clip` object if the mixer represents an audio-output device. Similarly, a target line is an output of the mixer: a `Port` object for an audio-output mixer, and a `TargetDataLine` object for an audio-input mixer. What if a mixer doesn't connect to any external hardware device at all? For example, consider an internal or software-only mixer that gets audio from an application program and delivers its mixed audio back to the program. This kind of mixer has `SourceDataLine` or `Clip` objects for its input lines and `TargetDataLine` objects for its output lines.

You can also use the following `AudioSystem` methods to learn more about source and target lines of a specified type that are supported by any installed mixer:

```
        static Line.Info[] getSourceLineInfo(Line.Info info)
        static Line.Info[] getTargetLineInfo(Line.Info info)
```

Note that the array returned by each of these methods indicates unique types of lines, not necessarily all the lines. For example, if two of a mixer's lines, or two lines of different mixers, have identical `Line.Info` objects, the two lines will represented by only one `Line.Info` in the returned array.

Getting a Line from a Mixer

The `Mixer` interface includes variations on the `AudioSystem` access methods for source and target lines, described above. These `Mixer` methods include ones that take `Line.Info` arguments, just as `AudioSystem's` methods do. However, `Mixer` also includes these variants, which take no arguments:

```
Line.Info[] getSourceLineInfo()
Line.Info[] getTargetLineInfo()
```

These methods return arrays of all the `Line.Info` objects for the particular mixer. Once you've obtained the arrays, you can iterate over them, calling `Mixer's getLine` method to obtain each line, followed by `Line's open` method to reserve use of each line for your program.

Selecting Input and Output Ports

The previous section, regarding how to obtain a line of a desired type, applies to ports as well as other types of lines. You can obtain all of the source (i.e., input) and target (i.e, output) ports by passing a `Port.Info` object to the `AudioSystem` (or `Mixer`) methods `getSourceLineInfo` and `getTargetLineInfo` that take a `Line.Info` argument. You then iterate over the returned array of objects and invoke Mixer's `getLine` method to get each port.

You can then open each `Port` by invoking `Line's open` method. Opening a port means you turn it on—that is, you allow sound to come in or out the port. Similarly, you can close ports that you don't want sound to travel through, because some ports might already be open before you even obtain them. Some platforms leave all ports on by default; or a user or system administrator might have selected certain ports to be on or off, using another application program or operating-system software.

Warning: If you want to select a certain port and make sure that the sound is actually going in or out the port, you can open the port as described. However, this can be considered user-hostile behavior! For example, a user might have the speaker port turned off so as not to disturb her co-workers. She would be rather upset if your program suddenly overrode her wishes and started blaring music. As another example, a user might want to be assured that his computer's microphone is never turned on without his knowledge, to avoid eavesdropping. In general, it is recommended not to open or close ports unless your program is responding to the user's intentions, as expressed through the user interface. Instead, respect the settings that the user or the operating system has already selected.

It isn't necessary to open or close a port before the mixer it's attached to will function correctly. For example, you can start playing back sound into an audio-output mixer, even though all its output ports are closed. The data still flows into the mixer; the playback isn't blocked. The user just won't hear anything. As soon as the user opens an output port, the sound will be audible through that port, starting at whatever point in the media the playback has already reached.

Also, you don't need to access the ports to learn whether the mixer has certain ports. To learn whether a mixer is actually an audio-output mixer, for example, you can invoke `getTargetLineInfo` to see whether it has output ports. There's no reason to access the ports themselves unless you want to change their settings (such as their open-or-closed state, or the settings of any controls they might have).

Permission to Use Audio Resources

The Java Sound API includes an `AudioPermission` class that indicates what kinds of access an applet (or an application running with a security manager) can have to the sampled-audio system. Permission to record sound is controlled separately. This permission should be granted with care, to help

prevent security risks such as unauthorized eavesdropping. By default, applets and applications are granted permissions as follows:

- An *applet* running with the applet security manager can play, but not record, audio.
- An *application* running with no security manager can both play and record audio.
- An application running with the default security manager can play, but not record, audio.

In general, applets are run under the scrutiny of a security manager and aren't permitted to record sound. Applications, on the other hand, don't automatically install a security manager, and are able to record sound. (However, if the default security manager is invoked explicitly for an application, the application isn't permitted to record sound.)

Both applets and applications can record sound even when running with a security manager if they have been granted explicit permission to do so.

If your program doesn't have permission to record (or play) sound, an exception will be thrown when it attempts to open a line. There is nothing you can do about this in your program, other than to catch the exception and report the problem to the user, because permissions can't be changed through the API. (If they could, they would be pointless, because nothing would be secure!) Generally, permissions are set in one or more policy configuration files, which a user or system administrator can edit using a text editor or the Policy Tool program.

For more information on security and permissions, see "Security Architecture" and "Policy Permissions" in the Security guide and the specialized trail on security in the Java Tutorial.

Chapter 4: Playing Back Audio

Playback is sometimes referred to as *presentation* or *rendering*. These are general terms that are applicable to other kinds of media besides sound. The essential feature is that a sequence of data is delivered somewhere for eventual perception by a user. If the data is time-based, as sound is, it must be delivered at the correct rate. With sound even more than video, it's important that the rate of data flow be maintained, because interruptions to sound playback often produce loud clicks or irritating distortion. The Java™ Sound API is designed to help application programs play sounds smoothly and continuously, even very long sounds.

The previous chapter discussed how to obtain a line from the audio system or from a mixer. This chapter shows how to play sound through a line.

There are two kinds of line that you can use for playing sound: a `Clip` and a `SourceDataLine`. These two interfaces were introduced briefly under "The Line Interface Hierarchy" in Chapter 2, "Overview of the Sampled Package." The chief difference between the two is that with a `Clip` you specify all the sound data at one time, before playback, whereas with a `SourceDataLine` you keep writing new buffers of data continuously during playback. Although there are many situations in which you could use either a `Clip` or a `SourceDataLine`, the following criteria help identify which kind of line is better suited for a particular situation:

- Use a `Clip` when you have non-real-time sound data that can be preloaded into memory.

  For example, you might read a short sound file into a clip. If you want the sound to play back more than once, a `Clip` is more convenient than a `SourceDataLine`, especially if you want the playback to loop (cycle repeatedly through all or part of the sound). If you need to start the playback at an arbitrary position in the sound, the `Clip` interface provides a method to do that easily. Finally, playback from a `Clip` generally has less latency than buffered playback from a `SourceDataLine`. In other words, because the sound is preloaded into a clip, playback can start immediately instead of having to wait for the buffer to be filled.
- Use a `SourceDataLine` for streaming data, such as a long sound file that won't all fit in memory at once, or a sound whose data can't be known in advance of playback.

  As an example of the latter case, suppose you're monitoring sound input—that is, playing sound back as it's being captured. If you don't have a mixer that can send input audio right back out an output port, your application program will have to take the captured data and send it to an audio-output mixer. In this case, a `SourceDataLine` is more appropriate than a `Clip`. Another example of sound that can't be known in advance occurs when you synthesize or manipulate the sound data interactively in response to the user's input. For example, imagine a game that gives aural feedback by "morphing" from one sound to another as the user moves the mouse. The

dynamic nature of the sound transformation requires the application program to update the sound data continuously during playback, instead of supplying it all before playback starts.

Using a Clip

You obtain a `Clip` as described earlier under "Getting a Line of a Desired Type" in Chapter 3, "Accessing Audio System Resources": Construct a `DataLine.Info` object with `Clip.class` for the first argument, and pass this `DataLine.Info` as an argument to the `getLine` method of `AudioSystem` or `Mixer`.

Setting Up the Clip for Playback

Obtaining a line just means you've gotten a way to refer to it; `getLine` doesn't actually reserve the line for you. Because a mixer might have a limited number of lines of the desired type available, it can happen that after you invoke `getLine` to obtain the clip, another application program jumps in and grabs the clip before you're ready to start playback. To actually use the clip, you need to reserve it for your program's exclusive use by invoking one of the following `Clip` methods:

```
void open(AudioInputStream stream)
void open(AudioFormat format, byte[] data, int offset,
int bufferSize)
```

Despite the `bufferSize` argument in the second `open` method above, `Clip` (unlike `SourceDataLine`) includes no methods for writing new data to the buffer. The `bufferSize` argument here just specifies how much of the byte array to load into the clip. It's not a buffer into which you can subsequently load more data, as you can with a `SourceDataLine's` buffer.

After opening the clip, you can specify at what point in the data it should start playback, using `Clip's setFramePosition` or `setMicroSecondPosition` methods. Otherwise, it will start at the beginning. You can also configure the playback to cycle repeatedly, using the `setLoopPoints` method.

Starting and Stopping Playback

When you're ready to start playback, simply invoke the `start` method. To stop or pause the clip, invoke the `stop` method, and to resume playback, invoke `start` again. The clip remembers the media position where it stopped playback, so there's no need for explicit pause and resume methods. If you don't want it to resume where it left off, you can "rewind" the clip to the beginning (or to any other position, for that matter) using the frame- or microsecond-positioning methods mentioned above.

A `Clip's` volume level and activity status (active versus inactive) can be monitored by invoking the `DataLine` methods `getLevel` and `isActive`, respectively. An active `Clip` is one that is currently playing sound.

Using a SourceDataLine

Obtaining a `SourceDataLine` is similar to obtaining a `Clip`. See "Getting a Line of a Desired Type" in Chapter 3, "Accessing Audio System Resources."

Setting Up the SourceDataLine for Playback

Opening the `SourceDataLine` is also similar to opening a `Clip`, in that the purpose is once again to reserve the line. However, you use a different method, inherited from `DataLine`:

```
void open(AudioFormat format)
```

Notice that when you open a `SourceDataLine`, you don't associate any sound data with the line yet, unlike opening a `Clip`. Instead, you just specify the format of the audio data you want to play. The system chooses a default buffer length.

You can also stipulate a certain buffer length in bytes, using this variant:

```
void open(AudioFormat format, int bufferSize)
```

For consistency with similar methods, the `buffersize` argument is expressed in bytes, but it must correspond to an integral number of frames.

How would you select a buffer size? It depends on your program's needs.

To start with, shorter buffer sizes mean less latency. When you send new data, you hear it sooner. For some application programs, particularly highly interactive ones, this kind of responsiveness is important. For example, in a game, the onset of playback might need to be tightly synchronized with a visual event. Such programs might need a latency of less than 0.1 second. As another example, a conferencing application needs to avoid delays in both playback and capture. However, many application programs can afford a greater delay, up to a second or more, because it doesn't matter exactly when the sound starts playing, as long as the delay doesn't confuse or annoy the user. This might be the case for an application program that streams a large audio file using one-second buffers. The user probably won't care if playback takes a second to start, because the sound itself lasts so long and the experience isn't highly interactive.

On the other hand, shorter buffer sizes also mean a greater risk that you'll fail to write data fast enough into the buffer. If that happens, the audio data will contain discontinuities, which will probably be audible as clicks or breakups in the sound. Shorter buffer sizes also mean that your program has to work harder to keep the buffers filled, resulting in more intensive CPU usage. This can slow down the execution of other threads in your program, not to mention other programs.

So an optimal value for the buffer size is one that minimizes latency just to the degree that's acceptable for your application program, while keeping it large enough to reduce the risk of buffer underflow and to avoid unnecessary consumption of CPU resources. For a program like a conferencing application, delays are more annoying than low-fidelity sound, so a small buffer size is preferable. For streaming music, an initial delay is acceptable, but breakups in the sound are not. Thus for streaming music a larger buffer size—say, a second—is preferable. (Note that high sample rates make the buffers larger in terms of the number of bytes, which are the units for measuring buffer size in the `DataLine` API.)

Instead of using the open method described above, it's also possible to open a `SourceDataLine` using

Line's open() method, without arguments. In this case, the line is opened with its default audio format and buffer size. However, you can't change these later. If you want to know the line's default audio format and buffer size, you can invoke DataLine's getFormat and getBufferSize methods, even before the line has ever been opened.

Starting and Stopping Playback

Once the SourceDataLine is open, you can start playing sound. You do this by invoking DataLine's start method, and then writing data repeatedly to the line's playback buffer.

The start method permits the line to begin playing sound as soon as there's any data in its buffer. You place data in the buffer by the following method:

```
int write(byte[] b, int offset, int length)
```

The offset into the array is expressed in bytes, as is the array's length.

The line begins sending data as soon as possible to its mixer. When the mixer itself delivers the data to its target, the SourceDataLine generates a START event. (In a typical implementation of the Java Sound API, the delay between the moment that the source line delivers data to the mixer and the moment that the mixer delivers the data to its target is negligible—that is, much less than the time of one sample.) This START event gets sent to the line's listeners, as explained below under "Monitoring a Line's Status." The line is now considered active, so the isActive method of DataLine will return true. Notice that all this happens only once the buffer contains data to play, not necessarily right when the start method is invoked. If you invoked start on a new SourceDataLine but never wrote data to the buffer, the line would never be active and a START event would never be sent. (However, in this case, the isRunning method of DataLine would return true.)

So how do you know how much data to write to the buffer, and when to send the second batch of data? Fortunately, you don't need to time the second invocation of write to synchronize with the end of the first buffer! Instead, you can take advantage of the write method's blocking behavior:

- The method returns as soon as the data has been written to the buffer. It doesn't wait until all the data in the buffer has finished playing. (If it did, you might not have time to write the next buffer without creating a discontinuity in the audio.)
- It's all right to try to write more data than the buffer will hold. In this case, the method blocks (doesn't return) until all the data you requested has actually been placed in the buffer. In other words, one buffer's worth of your data at a time will be written to the buffer and played, until the remaining data all fits in the buffer, at which point the method returns. Whether or not the method blocks, it returns as soon as the last buffer's worth of data from this invocation can be written. Again, this means that your code will in all likelihood regain control before playback of the last buffer's worth of data has finished.
- While in many contexts it is fine to write more data than the buffer will hold, if you want to be certain that the next write issued does not block, you can limit the number of bytes you write to the number that DataLine's available method returns.

Here's an example of iterating through chunks of data that are read from a stream, writing one chunk at a time to the SourceDataLine for playback:

```
// read chunks from a stream and write them to a source data
line
line.start();
while (total < totalToRead && !stopped)}
    numBytesRead = stream.read(myData, 0, numBytesToRead);
    if (numBytesRead == -1) break;
    total += numBytesRead;
    line.write(myData, 0, numBytesRead);

}
```

If you don't want the `write` method to block, you can first invoke the `available` method (inside the loop) to find out how many bytes can be written without blocking, and then limit the `numBytesToRead` variable to this number, before reading from the stream. In the example given, though, blocking won't matter much, since the write method is invoked inside a loop that won't complete until the last buffer is written in the final loop iteration. Whether or not you use the blocking technique, you'll probably want to invoke this playback loop in a separate thread from the rest of the application program, so that your program doesn't appear to freeze when playing a long sound. On each iteration of the loop, you can test whether the user has requested playback to stop. Such a request needs to set the `stopped` boolean, used in the code above, to `true`.

Since `write` returns before all the data has finished playing, how do you learn when the playback has actually completed? One way is to invoke the `drain` method of `DataLine` after writing the last buffer's worth of data. This method blocks until all the data has been played. When control returns to your program, you can free up the line, if desired, without fear of prematurely cutting off the playback of any audio samples:

```
line.write(b, offset, numBytesToWrite);
//this is the final invocation of write
line.drain();
line.stop();
line.close();
line = null;
```

You can intentionally stop playback prematurely, of course. For example, the application program might provide the user with a Stop button. Invoke `DataLine's stop` method to stop playback immediately, even in the middle of a buffer. This leaves any unplayed data in the buffer, so that if you subsequently invoke `start`, the playback resumes where it left off. If that's not what you want to happen, you can discard the data left in the buffer by invoking `flush`.

A `SourceDataLine` generates a `STOP` event whenever the flow of data has been stopped, whether this stoppage was initiated by the drain method, the stop method, or the flush method, or because the end of a playback buffer was reached before the application program invoked `write` again to provide new data. A `STOP` event doesn't necessarily mean that the `stop` method was invoked, and it doesn't necessarily mean that a subsequent invocation of `isRunning` will return `false`. It does, however, mean that `isActive` will return `false`. (When the `start` method has been invoked, the `isRunning` method will return `true`, even if a `STOP` event is generated, and it will begin to return

`false` only once the `stop` method is invoked.) It's important to realize that `START` and `STOP` events correspond to `isActive`, not to `isRunning`.

Monitoring a Line's Status

Once you have started a sound playing, how do you find when it's finished? We saw one solution above—invoking the `drain` method after writing the last buffer of data—but that approach is applicable only to a `SourceDataLine`. Another approach, which works for both `SourceDataLines` and `Clips`, is to register to receive notifications from the line whenever the line changes its state. These notifications are generated in the form of `LineEvent` objects, of which there are four types: `OPEN`, `CLOSE`, `START`, and `STOP`.

Any object in your program that implements the `LineListener` interface can register to receive such notifications. To implement the `LineListener` interface, the object simply needs an update method that takes a `LineEvent` argument. To register this object as one of the line's listeners, you invoke the following `Line` method:

```
public void addLineListener(LineListener listener)
```

Whenever the line opens, closes, starts, or stops, it sends an `update` message to all its listeners. Your object can query the `LineEvent` that it receives. First you might invoke `LineEvent.getLine` to make sure the line that stopped is the one you care about. In the case we're discussing here, you want to know if the sound is finished, so you see whether the `LineEvent` is of type `STOP`. If it is, you might check the sound's current position, which is also stored in the `LineEvent` object, and compare it to the sound's length (if known) to see whether it reached the end and wasn't stopped by some other means (such as the user's clicking a Stop button, although you'd probably be able to determine that cause elsewhere in your code).

Along the same lines, if you need to know when the line is opened, closed, or started, you use the same mechanism. `LineEvents` are generated by different kinds of lines, not just `Clips` and `SourceDataLines`. However, in the case of a `Port` you can't count on getting an event to learn about a line's open or closed state. For example, a `Port` might be initially open when it's created, so you don't invoke the `open` method and the `Port` doesn't ever generate an `OPEN` event. (See "Selecting Input and Output Ports" in Chapter 3, "Accessing Audio System Resources.")

Synchronizing Playback on Multiple Lines

If you're playing back multiple tracks of audio simultaneously, you probably want to have them all start and stop at exactly the same time. Some mixers facilitate this behavior with their `synchronize` method, which lets you apply operations such as `open`, `close`, `start`, and `stop` to a group of data lines using a single command, instead of having to control each line individually. Furthermore, the degree of accuracy with which operations are applied to the lines is controllable.

To find out whether a particular mixer offers this feature for a specified group of data lines, invoke the `Mixer` interface's `isSynchronizationSupported` method:

```
boolean isSynchronizationSupported(Line[] lines,
```

```
        boolean  maintainSync)
```

The first parameter specifies a group of specific data lines, and the second parameter indicates the accuracy with which synchronization must be maintained. If the second parameter is `true`, the query is asking whether the mixer is capable of maintaining sample-accurate precision in controlling the specified lines *at all times*; otherwise, precise synchronization is required only during start and stop operations, not throughout playback.

Processing the Outgoing Audio

Some source data lines have signal-processing controls, such as gain, pan, reverb, and sample-rate controls. Similar controls, especially gain controls, might be present on the output ports as well. For more information on how to determine whether a line has such controls, and how to use them if it does, see Chapter 6, "Processing Audio with Controls."

Chapter 5: Capturing Audio

Capturing refers to the process of obtaining a signal from outside the computer. A common application of audio capture is recording, such as recording the microphone input to a sound file. However, capturing isn't synonymous with recording, because recording implies that the application always saves the sound data that's coming in. An application that captures audio doesn't necessarily store the audio. Instead it might do something with the data as it's coming in—such as transcribe speech into text—but then discard each buffer of audio as soon as it's finished with that buffer.

As discussed in Chapter 2, "Overview of the Sampled Package," a typical audio-input system in an implementation of the Java™ Sound API consists of:

1. An input port, such as a microphone port or a line-in port, which feeds its incoming audio data into:
2. A mixer, which places the input data in:
3. One or more target data lines, from which an application can retrieve the data.

   Commonly, only one input port can be open at a time, but an audio-input mixer that mixes audio from multiple ports is also possible. Another scenario consists of a mixer that has no ports but instead gets its audio input over a network.

   The `TargetDataLine` interface was introduced briefly under "The Line Interface Hierarchy" in Chapter 2, "Overview of the Sampled Package." `TargetDataLine` is directly analogous to the `SourceDataLine` interface, which was discussed extensively in Chapter 4, "Playing Back Audio." Recall that the `SourceDataLine` interface consists of:

- A `write` method to send audio to the mixer
- An `available` method to determine how much data can be written to the buffer without blocking

Similarly, `TargetDataLine` consists of:

- A `read` method to get audio from the mixer
- An `available` method to determine how much data can be read from the buffer without blocking

Setting Up a TargetDataLine

The process of obtaining a target data line was described in Chapter 3, "Accessing Audio System Resources," but we repeat it here for convenience:

```
TargetDataLine line;
```

```
DataLine.Info info = new DataLine.Info(TargetDataLine.class,

    format); // format is an AudioFormat object
if (!AudioSystem.isLineSupported(info)) {
    // Handle the error ...
}
// Obtain and open the line.
try {

    line = (TargetDataLine) AudioSystem.getLine(info);
    line.open(format);
} catch (LineUnavailableException ex) {
    // Handle the error ...
}
```

You could instead invoke `Mixer's getLine` method, rather than `AudioSystem's`.

As shown in this example, once you've obtained a target data line, you reserve it for your application's use by invoking the `DataLine` method `open`, exactly as was described in the case of a source data line in Chapter 4, "Playing Back Audio." The single-parameter version of the `open` method causes the line's buffer to have the default size. You can instead set the buffer size according to your application's needs by invoking the two-parameter version:

```
void open(AudioFormat format, int bufferSize)
```

When choosing a buffer size, keep in mind the tradeoff between the delays incurred by long buffers, on the one hand, and the risk of discontinuities in the audio if the buffers are so short that you can't retrieve the data fast enough, on the other hand. When capturing audio, you risk data overflow if you don't pull data from the filled buffers fast enough. If overflow occurs, some of the captured data will be discarded, which will probably cause audible clicks and skips in the sound. This is the opposite situation from playback, where you risk data underflow, which can result in gaps in the sound. (See Chapter 4, "Playing Back Audio," for more on choosing buffer sizes.)

Reading the Data from the TargetDataLine

Once the line is open, it is ready to start capturing data, but it isn't active yet. To actually commence the audio capture, use the `DataLine` method `start`. This begins delivering input audio data to the line's buffer for your application to read. Your application should invoke start only when it's ready to begin reading from the line; otherwise a lot of processing is wasted on filling the capture buffer, only to have it overflow (that is, discard data).

To start retrieving data from the buffer, invoke `TargetDataLine's` read method:

```
int read(byte[] b, int offset, int length)
```

This method attempts to read `length` bytes of data into the array `b`, starting at the byte position `offset` in the array. The method returns the number of bytes actually read.

As with `SourceDataLine's write` method, you can request more data than actually fits in the buffer, because the method blocks until the requested amount of data has been delivered, even if you request many buffers' worth of data.

To avoid having your application hang during recording, you can invoke the read method within a loop, until you've retrieved all the audio input, as in this example:

```
// Assume that the TargetDataLine, line, has already
// been obtained and opened.
ByteArrayOutputStream out  = new ByteArrayOutputStream();
int numBytesRead;
byte[] data = new byte[line.getBufferSize() / 5];

// Begin audio capture.
line.start();

// Here, stopped is a global boolean set by another thread.
while (!stopped) {
   // Read the next chunk of data from the TargetDataLine.
   numBytesRead =  line.read(data, 0, data.length);
   // Save this chunk of data.
   out.write(data, 0, numBytesRead);
}
```

Notice that in this example, the size of the byte array into which the data is read is set to be one-fifth the size of the line's buffer. If you instead make it as big as the line's buffer and try to read the entire buffer, you need to be very exact in your timing, because data will be dumped if the mixer needs to deliver data to the line while you are reading from it. By using some fraction of the line's buffer size, as shown here, your application will be more successful in sharing access to the line's buffer with the mixer.

The `read` method of `TargetDataLine` takes three arguments: a byte array, an offset into the array, and the number of bytes of input data that you would like to read. In this example, the third argument is simply the length of your byte array. The `read` method returns the number of bytes that were actually read into your array.

Typically, you read data from the line in a loop, as in this example. Within the `while` loop, each chunk of retrieved data is processed in whatever way is appropriate for the application—here, it's written to a `ByteArrayOutputStream`. Not shown here is the use of a separate thread to set the boolean `stopped`, which terminates the loop. This boolean's value might be set to `true` when the user clicks a Stop button, and also when a listener receives a `CLOSE` or `STOP` event from the line. The listener is necessary for `CLOSE` events and recommended for `STOP` events. Otherwise, if the line gets stopped somehow without stopped being set to `true`, the `while` loop will capture zero bytes on each iteration, running fast and wasting CPU cycles. A more thorough code example would show the loop being re-entered if capture becomes active again.

As with a source data line, it's possible to drain or flush a target data line. For example, if you're recording the input to a file, you'll probably want to invoke the `drain` method when the user clicks a

Stop button. The `drain` method will cause the mixer's remaining data to get delivered to the target data line's buffer. If you don't drain the data, the captured sound might seem to be truncated prematurely at the end.

There might be some cases where you instead want to flush the data. In any case, if you neither flush nor drain the data, it will be left in the mixer. This means that when capture recommences, there will be some leftover sound at the beginning of the new recording, which might be undesirable. It can be useful, then, to flush the target data line before restarting the capture.

Monitoring the Line's Status

Because the `TargetDataLine` interface extends `DataLine`, target data lines generate events in the same way source data lines do. You can register an object to receive events whenever the target data line opens, closes, starts, or stops. For more information, see "Monitoring the Line's Status" in Chapter 4, "Playing Back Audio."

Processing the Incoming Audio

Like some source data lines, some mixers' target data lines have signal-processing controls, such as gain, pan, reverb, or sample-rate controls. The input ports might have similar controls, especially gain controls. For more information on how to determine whether a line has such controls, and how to use them if it does, see Chapter 6, "Processing Audio with Controls."

Chapter 6: Processing Audio with Controls

Previous chapters have discussed how to play or capture audio samples. The implicit goal has been to deliver samples as faithfully as possible, without modification (other than possibly mixing the samples with those from other audio lines). Sometimes, however, you want to be able to modify the signal. The user might want it to sound louder, quieter, fuller, more reverberant, higher or lower in pitch, and so on. This chapter discusses the Java™ Sound API features that provide these kinds of signal processing.

There are two ways to apply signal processing:

- You can use any processing supported by the mixer or its component lines, by querying for `Control` objects and then setting the controls as the user desires. Typical controls supported by mixers and lines include gain, pan, and reverberation controls.
- If the kind of processing you need isn't provided by the mixer or its lines, your program can operate directly on the audio bytes, manipulating them as desired.

This chapter discusses the first technique in greater detail, because there is no special API for the second technique.

Introduction to Controls

A mixer can have various sorts of signal-processing controls on some or all of its lines. For example, a mixer used for audio capture might have an input port with a gain control, and target data lines with gain and pan controls. A mixer used for audio playback might have sample-rate controls on its source data lines. In each case, the controls are all accessed through methods of the `Line` interface.

Because the `Mixer` interface extends `Line`, the mixer itself can have its own set of controls. These might serve as master controls affecting all the mixer's source or target lines. For example, the mixer might have a master gain control whose value in decibels is added to the values of individual gain controls on its target lines.

Others of the mixer's own controls might affect a special line, neither a source nor a target, that the mixer uses internally for its processing. For example, a global reverb control might choose the sort of reverberation to apply to a mixture of the input signals, and this "wet" (reverberated) signal would get mixed back into the "dry" signal before delivery to the mixer's target lines.

If the mixer or any of its lines have controls, you might wish to expose the controls via graphical objects in your program's user interface, so that the user can adjust the audio characteristics as desired. The controls are not themselves graphical; they just allow you to retrieve and change their settings. It's up to

you to decide what sort of graphical representations (sliders, buttons, etc.), if any, to use in your program.

All controls are implemented as concrete subclasses of the abstract class `Control`. Many typical audio-processing controls can be described by abstract subclasses of `Control` based on a data type (such as boolean, enumerated, or float). Boolean controls, for example, represent binary-state controls, such as on/off controls for mute or reverb. Float controls, on the other hand, are well suited to represent continuously variable controls, such as pan, balance, or volume.

The Java Sound API specifies the following abstract subclasses of `Control`:

- `BooleanControl`—represents a binary-state (true or false) control. For example, mute, solo, and on/off switches would be good candidates for `BooleanControls`.
- `FloatControl`—data model providing control over a range of floating-point values. For example, volume and pan are `FloatControls` that could be manipulated via a dial or slider.
- `EnumControl`—offers a choice from a set of objects. For example, you might associate a set of buttons in the user interface with an `EnumControl` to select one of several preset reverberation settings.
- `CompoundControl`—provides access to a collection of related items, each of which is itself an instance of a `Control` subclass. `CompoundControls` represent multi-control modules such as graphic equalizers. (A graphic equalizer would typically be depicted by a set of sliders, each affecting a `FloatControl`.)

Each subclass of `Control` above has methods appropriate for its underlying data type. Most of the classes include methods that set and get the control's current value(s), get the control's label(s), and so on.

Of course, each class has methods that are particular to it and the data model represented by the class. For example, `EnumControl` has a method that lets you get the set of its possible values, and `FloatControl` permits you to get its minimum and maximum values, as well as the precision (increment or step size) of the control.

Each subclass of `Control` has a corresponding `Control.Type` subclass, which includes static instances that identify specific controls.

The following table shows each `Control` subclass, its corresponding `Control.Type` subclass, and the static instances that indicate specific kinds of controls:

`Control`	`Control.Type`	`Control.Type` instances

`BooleanControl`	`BooleanControl.Type`	`MUTE`
		Mute status of line
		`APPLY_REVERB`
		Reverberation on/off

`CompoundControl`	`CompoundControl.Type`	(none)

`EnumControl`	`EnumControl.Type`	`REVERB` Access to reverb settings (each is an instance of ReverbType)

`FloatControl`	`FloatControl.Type`	`AUX_RETURN`
		Auxiliary return gain on a line
		`AUX_SEND`
		Auxiliary send gain on a line
		`BALANCE`
		Left-right volume balance
		`MASTER_GAIN`
		Overall gain on a line
		`PAN`
		Left-right position
		`REVERB_RETURN`
		Post-reverb gain on a line
		`REVERB_SEND`
		Pre-reverb gain on a line
		`SAMPLE_RATE`
		Playback sample rate
		`VOLUME`
		Volume on a line

An implementation of the Java Sound API can provide any or all of these control types on its mixers and lines. It can also supply additional control types not defined in the Java Sound API. Such control types could be implemented via concrete subclasses of any of these four abstract subclasses, or via additional `Control` subclasses that don't inherit from any of these four abstract subclasses. An application program can query each line to find what controls it supports.

Getting a Line that Has the Desired Controls

In many cases, an application program will simply display whatever controls happen to be supported by the line in question. If the line doesn't have any controls, so be it. But what if it's important to find a line that has certain controls? In that case, you can use a `Line.Info` to obtain a line that has the right characteristics, as described under "Getting a Line of a Desired Type" in Chapter 3, "Accessing Audio System Resources."

For example, suppose you prefer an input port that lets the user set the volume of the sound input. The following code excerpt shows how one might query the default mixer to determine whether it has the desired port and control:

```
Port lineIn;
FloatControl volCtrl;
try {
  mixer = AudioSystem.getMixer(null);
  lineIn = (Port)mixer.getLine(Port.Info.LINE_IN);
  lineIn.open();
  volCtrl = (FloatControl) lineIn.getControl(

      FloatControl.Type.VOLUME);
  // Assuming getControl call succeeds,
  // we now have our LINE_IN VOLUME control.
} catch (Exception e) {
  System.out.println("Failed trying to find LINE_IN"
    + " VOLUME control: exception = " + e);
}
if (volCtrl != null)
  // ...
```

Getting the Controls from the Line

An application program that needs to expose controls in its user interface might simply query the available lines and controls, and then display an appropriate user-interface element for every control on every line of interest. In such a case, the program's only mission is to provide the user with "handles" on the control; not to know what those controls do to the audio signal. As long as the program knows how to map a line's controls into user-interface elements, the Java Sound API architecture of `Mixer`, `Line`, and `Control` will generally take care of the rest.

For example, suppose your program plays back sound. You're using a `SourceDataLine`, which you've obtained as described under "Getting a Line of a Desired Type" in Chapter 3, "Accessing Audio System Resources." You can access the line's controls by invoking the `Line` method:

```
Control[] getControls()
```

Then, for each of the controls in this returned array, you then use the following `Control` method to get the control's type:

```
Control.Type getType()
```

Knowing the specific `Control.Type` instance, your program can display a corresponding user-interface element. Of course, choosing "a corresponding user-interface element" for a specific `Control.Type` depends on the approach taken by your program. On the one hand, you might use the same kind of element to represent all `Control.Type` instances of the same class. This would require you to query the *class* of the `Control.Type` instance using, for example, the `Object.getClass` method. Let's say the result matched `BooleanControl.Type`. In this case, your program might display a generic checkbox or toggle button, but if its class matched `FloatControl.Type`, then you might display a graphic slider.

On the other hand, your program might distinguish between different types of controls—even those of the same class—and use a different user-interface element for each one. This would require you to test the *instance* returned by `Control's getType` method. Then if, for example, the type matched `BooleanControl.Type.APPLY_REVERB`, your program might display a checkbox; while if the type matched `BooleanControl.Type.MUTE`, you might instead display a toggle button.

Using a Control to Change the Audio Signal

> **Note**
>
> The current implementation requires that to change the value of a `Control` its corresponding `Line` must be open.

Now that you know how to access a control and determine its type, this section will describe how to use `Controls` to change aspects of the audio signal. This section doesn't cover every available control; rather, it provides a few examples in this area to show you how to get started. These example include:

- Controlling a line's mute state
- Changing a line's volume
- Selecting among various reverberation presets

Suppose that your program has accessed all of its mixers, their lines and the controls on those lines, and that it has a data structure to manage the logical associations between the controls and their corresponding user-interface elements. Then, translating the user's manipulations of those controls into the corresponding `Control` methods becomes a fairly straightforward matter.

The following subsections describe some of the methods that must be invoked to affect the changes to specific controls.

Controlling a Line's Mute State

Controlling the mute state of any line is simply a matter of calling the following `BooleanControl` method:

```
void setValue(boolean value)
```

(Presumably, the program knows, by referring to its control-management data structures, that the mute is an instance of a `BooleanControl`.) To mute the signal that's passing through the line, the program invokes the method above, specifying `true` as the value. To turn muting off, permitting the signal to

flow through the line, the program invokes the method with the parameter set to `false`.

Changing a Line's Volume

Let's assume your program associates a particular graphic slider with a particular line's volume control. The value of a volume control (i.e., `FloatControl.Type.VOLUME`) is set using the following `FloatControl` method:

```
void setValue(float newValue)
```

Detecting that the user moved the slider, the program gets the slider's current value and passes it, as the parameter `newValue`, to the method above. This changes the volume of the signal flowing though the line that "owns" the control.

Selecting among Various Reverberation Presets

Let's suppose that our program has a mixer with a line that has a control of type `EnumControl.Type.REVERB`. Calling the `EnumControl` method:

```
java.lang.Objects[] getValues()
```

on that control produces an array of `ReverbType` objects. If desired, the particular parameter settings of each of these objects can be accessed using the following `ReverbType` methods:

```
int getDecayTime()
int getEarlyReflectionDelay()
float getEarlyReflectionIntensity()
int getLateReflectionDelay()
float getLateReflectionIntensity()
```

For example, if a program only wants a single reverb setting that sounds like a cavern, it can iterate over the `ReverbType` objects until it finds one for which `getDecayTime` returns a value greater than 2000. For a thorough explanation of these methods, including a table of representative return values, see the API reference documentation for `javax.sound.sampled.ReverbType`.

Typically, though, a program will create a user-interface element, for example, a radio button, for each of the `ReverbType` objects within the array returned by the `getValues` method. When the user clicks on one of these radio buttons, the program invokes the `EnumControl` method

```
void setValue(java.lang.Object value)
```

where `value` is set to the `ReverbType` that corresponds to the newly engaged button. The audio signal sent through the line that "owns" this `EnumControl` will then be reverberated according to the parameter settings that constitute the control's current `ReverbType` (i.e., the particular `ReverbType` specified in the `value` argument of the `setValue` method).

So, from our application program's perspective, enabling a user to move from one reverberation preset (i.e., ReverbType) to another is simply a matter of connecting each element of the array returned by `getValues` to a distinct radio button.

Manipulating the Audio Data Directly

The `Control` API allows an implementation of the Java Sound API, or a third-party provider of a mixer, to supply arbitrary sorts of signal processing through controls. But what if no mixer offers the kind of signal processing you need? It will take more work, but you might be able to implement the signal processing in your program. Because the Java Sound API gives you access to the audio data as an array of bytes, you can alter these bytes in any way you choose.

If you're processing incoming sound, you can read the bytes from a `TargetDataLine` and then manipulate them. An algorithmically trivial example that can yield sonically intriguing results is to play a sound backwards by arranging its frames in reverse order. This trivial example may not be of much use for your program, but there are numerous sophisticated digital signal processing (DSP) techniques that might be more appropriate. Some examples are equalization, dynamic-range compression, peak limiting, and time stretching or compression, as well as special effects such as delay, chorus, flanging, distortion, and so on.

To play back processed sound, you can place your manipulated array of bytes into a `SourceDataLine` or `Clip`. Of course, the array of bytes need not be derived from an existing sound. You can synthesize sounds from scratch, although this requires some knowledge of acoustics or else access to sound-synthesis functions. For either processing or synthesis, you may want to consult an audio DSP textbook for the algorithms in which you're interested, or else import a third-party library of signal-processing functions into your program. For playback of synthesized sound, consider whether the `Synthesizer` API in the `javax.sound.midi` package meets your needs instead. (See Chapter 12, "Synthesizing Sound.")

Chapter 7: Using Files and Format Converters

Most application programs that deal with sound need to read sound files or audio streams. This is common functionality, regardless of what the program may subsequently do with the data it reads (such as play, mix, or process it). Similarly, many programs need to write sound files (or streams). In some cases, the data that has been read (or that will be written) needs to be converted to a different format.

As was briefly mentioned in Chapter 3, "Accessing Audio System Resources," the Java™ Sound API provides application developers with various facilities for file input/output and format translations. Application programs can read, write, and translate between a variety of sound file formats and audio data formats.

Chapter 2, "Overview of the Sampled Package," introduced the main classes related to sound files and audio data formats. As a review:

- A stream of audio data, as might be read from or written to a file, is represented by an `AudioInputStream` object. (`AudioInputStream` inherits from `java.io.InputStream`.)
- The format of this audio data is represented by an `AudioFormat` object.

  This format specifies how the audio samples themselves are arranged, but not the structure of a file that they might be stored in. In other words, an `AudioFormat` describes "raw" audio data, such as the system might hand your program after capturing it from a microphone input or after parsing it from a sound file. An `AudioFormat` includes such information as the encoding, the byte order, the number of channels, the sampling rate, and the number of bits per sample.
- There are several well-known, standard formats for sound files, such as WAV, AIFF, or AU. The different types of sound file have different structures for storing audio data as well as for storing descriptive information about the audio data. A sound file format is represented in the Java Sound API by an `AudioFileFormat` object. The `AudioFileFormat` includes an `AudioFormat` object to describe the format of the audio data stored in the file, and also includes information about the file type and the length of the data in the file.
- The `AudioSystem` class provides methods for (1) storing a stream of audio data from an `AudioInputStream` into an audio file of a particular type (in other words, writing a file), (2) extracting a stream of audio bytes (an `AudioInputStream`) from an audio file (in other words, reading a file), and (3) converting audio data from one data format to another. The present chapter, which is divided into three sections, explains these three kinds of activity.

Note that an implementation of the Java Sound API does not necessarily provide comprehensive facilities

for reading, writing, and converting audio in different data and file formats. It might support only the most common data and file formats. However, service providers can develop and distribute conversion services that extend this set, as outlined in Chapter 14, "Providing Sampled-Audio Services." The `AudioSystem` class supplies methods that allow application programs to learn what conversions are available, as described later in this chapter under "Converting File and Data Formats."

Reading Sound Files

The `AudioSystem` class provides two types of file-reading services:

- Information about the format of the audio data stored in the sound file
- A stream of formatted audio data that can be read from the sound file

The first of these is given by three variants of the `getAudioFileFormat` method:

```
static AudioFileFormat getAudioFileFormat (java.io.File file)
static AudioFileFormat getAudioFileFormat(java.io.InputStream
stream)
static AudioFileFormat getAudioFileFormat (java.net.URL url)
```

As mentioned above, the returned `AudioFileFormat` object tells you the file type, the length of the data in the file, encoding, the byte order, the number of channels, the sampling rate, and the number of bits per sample.

The second type of file-reading functionality is given by these `AudioSystem` methods

```
static AudioInputStream getAudioInputStream (java.io.File
file)
static AudioInputStream getAudioInputStream (java.net.URL
url)
static AudioInputStream getAudioInputStream
(java.io.InputStream stream)
```

These methods give you an object (an `AudioInputStream`) that lets you read the file's audio data, using one of the read methods of `AudioInputStream`. We'll see an example momentarily.

Suppose you're writing a sound-editing application that allows the user to load sound data from a file, display a corresponding waveform or spectrogram, edit the sound, play back the edited data, and save the result in a new file. Or perhaps your program will read the data stored in a file, apply some kind of signal processing (such as an algorithm that slows the sound down without changing its pitch), and then play the processed audio. In either case, you need to get access to the data contained in the audio file. Assuming that your program provides some means for the user to select or specify an input sound file, reading that file's audio data involves three steps:

1. Get an `AudioInputStream` object from the file.
2. Create a byte array in which you'll store successive chunks of data from the file.
3. Repeatedly read bytes from the audio input stream into the array. On each iteration, do something useful with the bytes in the array (for example, you might play them, filter them, analyze them, display them, or write them to another file).

The following code example outlines these steps.

```
int totalFramesRead = 0;
File fileIn = new File(somePathName);
// somePathName is a pre-existing string whose value was
// based on a user selection.
try {
  AudioInputStream audioInputStream =
    AudioSystem.getAudioInputStream(fileIn);
  int bytesPerFrame =
    audioInputStream.getFormat().getFrameSize();
  // Set an arbitrary buffer size of 1024 frames.
  int numBytes = 1024 * bytesPerFrame;
  byte[] audioBytes = new byte[numBytes];
  try {
    int numBytesRead = 0;
    int numFramesRead = 0;
    // Try to read numBytes bytes from the file.
    while ((numBytesRead =
      audioInputStream.read(audioBytes)) != -1) {
      // Calculate the number of frames actually read.
      numFramesRead = numBytesRead / bytesPerFrame;
      totalFramesRead += numFramesRead;
      // Here, do something useful with the audio data that's
      // now in the audioBytes array...
    }
  } catch (Exception ex) {
    // Handle the error...
  }
} catch (Exception e) {
  // Handle the error...
}
```

Let's take a look at what's happening in the above code sample. First, the outer try clause instantiates an `AudioInputStream` object through the call to the `AudioSystem.getAudioInputStream(File)` method. This method transparently performs all of the testing required to determine whether the specified file is actually a sound file of a type that is supported by the Java Sound API. If the file being inspected (`fileIn` in this example) is not a sound file, or is a sound file of some unsupported type, an `UnsupportedAudioFileException` exception is thrown. This behavior is convenient, in that the application programmer need not be bothered with testing file attributes, nor with adhering to any file-naming conventions. Instead, the `getAudioInputStream` method takes care of all the low-level parsing and verification that is required to validate the input file.

The outer `try` clause then creates a byte array, `audioBytes`, of an arbitrary fixed length. We make sure that its length in bytes equals an integral number of frames, so that we won't end up reading only

part of a frame or, even worse, only part of a sample. This byte array will serve as a buffer to temporarily hold a chunk of audio data as it's read from the stream. If we knew we would be reading nothing but very short sound files, we could make this array the same length as the data in the file, by deriving the length in bytes from the length in frames, as returned by `AudioInputStream's getFrameLength` method. (Actually, we'd probably just use a `Clip` object instead.) But to avoid running out of memory in the general case, we instead read the file in chunks, one buffer at a time.

The inner `try` clause contains a `while` loop, which is where we read the audio data from the `AudioInputStream` into the byte array. You should add code in this loop to handle the audio data in this array in whatever way is appropriate for your program's needs. If you're applying some kind of signal processing to the data, you'll probably need to query the `AudioInputStream's AudioFormat` further, to learn the number of bits per sample and so on.

Note that the method `AudioInputStream.read(byte[])` returns the number of *bytes* read—not the number of samples or frames. This method returns -1 when there's no more data to read. Upon detecting this condition, we break from the `while` loop.

Writing Sound Files

The previous section described the basics of reading a sound file, using specific methods of the `AudioSystem` and `AudioInputStream` classes. This section describes how to write audio data out to a new file.

The following `AudioSystem` method creates a disk file of a specified file type. The file will contain the audio data that's in the specified `AudioInputStream`:

```
static int write(AudioInputStream in,
  AudioFileFormat.Type fileType, File out)
```

Note that the second argument must be one of the file types supported by the system (for example, AU, AIFF, or WAV), otherwise the `write` method will throw an `IllegalArgumentException`. To avoid this, you can test whether or not a particular `AudioInputStream` may be written to a particular type of file, by invoking this `AudioSystem` method:

```
static boolean isFileTypeSupported
  (AudioFileFormat.Type fileType, AudioInputStream stream)
```

which will return `true` only if the particular combination is supported.

More generally, you can learn what types of file the system can write by invoking one of these `AudioSystem` methods:

```
static AudioFileFormat.Type[] getAudioFileTypes()
static AudioFileFormat.Type[]
  getAudioFileTypes(AudioInputStream stream)
```

The first of these returns all the types of file that the system can write, and the second returns only those that the system can write from the given audio input stream.

The following excerpt demonstrates one technique for creating an output file from an `AudioInputStream` using the `write` method mentioned above.

```
File fileOut = new File(someNewPathName);
AudioFileFormat.Type fileType = fileFormat.getType();
if (AudioSystem.isFileTypeSupported(fileType,
    audioInputStream)) {
  AudioSystem.write(audioInputStream, fileType, fileOut);
}
```

The first statement above, creates a new `File` object, `fileOut`, with a user- or program-specified pathname. The second statement gets a file type from a pre-existing `AudioFileFormat` object called `fileFormat`, which might have been obtained from another sound file, such as the one that was read in the "Reading Sound Files" section of this chapter. (You could instead supply whatever supported file type you want, instead of getting the file type from elsewhere. For example, you might delete the second statement and replace the other two occurrences of `fileType` in the code above with `AudioFileFormat.Type.WAVE`.)

The third statement tests whether a file of the designated type can be written from a desired `AudioInputStream`. Like the file format, this stream might have been derived from the sound file previously read. (If so, presumably you've processed or altered its data in some way, because otherwise there are easier ways to simply copy a file.) Or perhaps the stream contains bytes that have been freshly captured from the microphone input.

Finally, the stream, file type, and output file are passed to the `AudioSystem.write` method, to accomplish the goal of writing the file.

Converting File and Data Formats

Recall from the section "What Is Formatted Audio Data?" in Chapter 2, "Overview of the Sampled Package," that the Java Sound API distinguishes between audio *file* formats and audio *data* formats. The two are more or less independent. Roughly speaking, the data format refers to the way in which the computer represents each raw data point (sample), while the file format refers to the organization of a sound file as stored on a disk. Each sound file format has a particular structure that defines, for example, the information stored in the file's header. In some cases, the file format also includes structures that contain some form of meta-data, in addition to the actual "raw" audio samples. The remainder of this chapter examines methods of the Java Sound API that enable a variety of file-format and data-format conversions.

Converting from One File Format to Another

This section covers the fundamentals of converting audio file types in the Java Sound API. Once again we pose a hypothetical program whose purpose, this time, is to read audio data from an arbitrary input file and write it into a file whose type is AIFF. Of course, the input file must be of a type that the system is capable of reading, and the output file must be of a type that the system is capable of writing. (In this example, we assume that the system is capable of writing AIFF files.) The example program doesn't do any data format conversion. If the input file's data format can't be represented as an AIFF file, the

program simply notifies the user of that problem. On the other hand, if the input sound file is an already an AIFF file, the program notifies the user that there is no need to convert it.

The following function implements the logic just described:

```
public void ConvertFileToAIFF(String inputPath,
  String outputPath) {
  AudioFileFormat inFileFormat;
  File inFile;
  File outFile;
  try {
    inFile = new File(inputPath);
    outFile = new File(outputPath);
  } catch (NullPointerException ex) {
    System.out.println("Error: one of the
      ConvertFileToAIFF" +" parameters is null!");
    return;
  }
  try {
    // query file type
    inFileFormat = AudioSystem.getAudioFileFormat(inFile);
    if (inFileFormat.getType() != AudioFileFormat.Type.AIFF)
    {
      // inFile is not AIFF, so let's try to convert it.
      AudioInputStream inFileAIS =
        AudioSystem.getAudioInputStream(inFile);
      inFileAIS.reset(); // rewind
      if (AudioSystem.isFileTypeSupported(
              AudioFileFormat.Type.AIFF, inFileAIS)) {
         // inFileAIS can be converted to AIFF.
         // so write the AudioInputStream to the
         // output file.
         AudioSystem.write(inFileAIS,
           AudioFileFormat.Type.AIFF, outFile);
         System.out.println("Successfully made AIFF file, "
           + outFile.getPath() + ", from "
           + inFileFormat.getType() + " file, " +
           inFile.getPath() + ".");
         inFileAIS.close();
         return; // All done now
       } else
         System.out.println("Warning: AIFF conversion of "
           + inFile.getPath()
           + " is not currently supported by AudioSystem.");
    } else
      System.out.println("Input file " + inFile.getPath() +
```

```
            " is AIFF." + " Conversion is unnecessary.");
      } catch (UnsupportedAudioFileException e) {
        System.out.println("Error: " + inFile.getPath()
            + " is not a supported audio file type!");
        return;
      } catch (IOException e) {
        System.out.println("Error: failure attempting to read "
          + inFile.getPath() + "!");
        return;
      }
    }
```

As mentioned, the purpose of this example function, `ConvertFileToAIFF`, is to query an input file to determine whether it's an AIFF sound file, and if it isn't, to try to convert it to one, producing a new copy whose pathname is specified by the second argument. (As an exercise, you might try making this function more general, so that instead of always converting to AIFF, the function converts to the file type specified by a new function argument.) Note that the audio data format of the copy—that is, the new file-mimics the audio data format of original input file.

Most of this function is self-explanatory and is not specific to the Java Sound API. There are, however, a few Java Sound API methods used by the routine that are crucial for sound file-type conversions. These method invocations are all found in the second `try` clause, above, and include the following:

- `AudioSystem.getAudioFileFormat`: used here to determine whether the input file is already an AIFF type. If so, the function quickly returns; otherwise the conversion attempt proceeds.
- `AudioSystem.isFileTypeSupported`: Indicates whether the system can write a file of the specified type that contains audio data from the specified `AudioInputStream.` In our example, this method returns `true` if the specified audio input file can be converted to AIFF audio file format. If `AudioFileFormat.Type.AIFF` isn't supported, `ConvertFileToAIFF` issues a warning that the input file can't be converted, then returns.
- `AudioSystem.write`: used here to write the audio data from the AudioInputStream `inFileAIS` to the output file `outFile`.

The second of these methods, `isFileTypeSupported`, helps to determine, in advance of the write, whether a particular input sound file can be converted to a particular output sound file type. In the next section we will see how, with a few modifications to this `ConvertFileToAIFF` sample routine, we can convert the audio data format, as well as the sound file type.

Converting Audio between Different Data Formats

The previous section showed how to use the Java Sound API to convert a file from one *file* format (that is, one type of sound file) to another. This section explores some of the methods that enable audio *data* format conversions.

In the previous section, we read data from a file of an arbitrary type, and saved it in an AIFF file. Note that although we changed the type of file used to store the data, we didn't change the format of the audio data itself. (Most common audio file types, including AIFF, can contain audio data of various formats.)

So if the original file contained CD-quality audio data (16-bit sample size, 44.1-kHz sample rate, and two channels), so would our output AIFF file.

Now suppose that we want to specify the *data* format of the output file, as well as the file type. For example, perhaps we are saving many long files for use on the Internet, and are concerned about the amount of disk space and download time required by our files. We might choose to create smaller AIFF files that contain lower-resolution data-for example, data that has an 8-bit sample size, an 8-kHz sample rate, and a single channel.

Without going into as much coding detail as before, let's explore some of the methods used for data format conversion, and consider the modifications that we would need to make to the `ConvertFileToAIFF` function to accomplish the new goal.

The principal method for audio data conversion is, once again, found in the `AudioSystem` class. This method is a variant of `getAudioInputStream`:

```
AudioInputStream getAudioInputStream(AudioFormat
    format, AudioInputStream stream)
```

This function returns an `AudioInputStream` that is the result of converting the `AudioInputStream`, `stream`, using the indicated `AudioFormat`, `format`. If the conversion isn't supported by `AudioSystem`, this function throws an `IllegalArgumentException`.

To avoid that, we can first check whether the system can perform the required conversion by invoking this `AudioSystem` method:

```
boolean isConversionSupported(AudioFormat targetFormat,
    AudioFormat sourceFormat)
```

In this case, we'd pass `stream.getFormat()` as the second argument.

To create a specific `AudioFormat` object, we use one of the two `AudioFormat` constructors shown below, either

```
AudioFormat(float sampleRate, int sampleSizeInBits,
    int channels, boolean signed, boolean bigEndian)
```

which constructs an `AudioFormat` with a linear PCM encoding and the given parameters, or

```
AudioFormat(AudioFormat.Encoding encoding,
    float sampleRate, int sampleSizeInBits, int channels,
    int frameSize, float frameRate, boolean bigEndian)
```

which also constructs an `AudioFormat`, but lets you specify the encoding, frame size, and frame rate, in addition to the other parameters.

Now, armed with the methods above, let's see how we might extend our `ConvertFileToAIFF` function to perform the desired "low-res" audio data format conversion. First, we would construct an `AudioFormat` object describing the desired output audio data format. The following statement would suffice and could be inserted near the top of the function:

```
AudioFormat outDataFormat = new AudioFormat((float) 8000.0,
(int) 8, (int) 1, true, false);
```

Since the `AudioFormat` constructor above is describing a format with 8-bit samples, the last parameter

to the constructor, which specifies whether the samples are big or little endian, is irrelevant. (Big versus little endian is only an issue if the sample size is greater than a single byte.)

The following example shows how we would use this new `AudioFormat` to convert the `AudioInputStream`, `inFileAIS`, that we created from the input file:

```
AudioInputStream lowResAIS;
  if (AudioSystem.isConversionSupported(outDataFormat,
    inFileAIS.getFormat())) {
    lowResAIS = AudioSystem.getAudioInputStream
      (outDataFormat, inFileAIS);
  }
```

It wouldn't matter too much where we inserted this code, as long as it was after the construction of `inFileAIS`. Without the `isConversionSupported` test, the call would fail and throw an `IllegalArgumentException` if the particular conversion being requested was unsupported. (In this case, control would transfer to the appropriate `catch` clause in our function.)

So by this point in the process, we would have produced a new `AudioInputStream`, resulting from the conversion of the original input file (in its `AudioInputStream` form) to the desired low-resolution audio data format as defined by `outDataFormat`.

The final step to produce the desired low-resolution, AIFF sound file would be to replace the `AudioInputStream` parameter in the call to `AudioSystem.write` (that is, the first parameter) with our converted stream, `lowResAIS`, as follows:

```
AudioSystem.write(lowResAIS, AudioFileFormat.Type.AIFF,
  outFile);
```

These few modifications to our earlier function produce something that converts both the audio data and the file format of any specified input file, assuming of course that the system supports the conversion.

Learning What Conversions Are Available

Several `AudioSystem` methods test their parameters to determine whether the system supports a particular data format conversion or file-writing operation. (Typically, each method is paired with another that performs the data conversion or writes the file.) One of these query methods, `AudioSystem.isFileTypeSupported`, was used in our example function, `ConvertFileToAIFF`, to determine whether the system was capable of writing the audio data to an AIFF file. A related `AudioSystem` method, `getAudioFileTypes(AudioInputStream)`, returns the complete list of supported file types for the given stream, as an array of `AudioFileFormat.Type` instances. The method:

```
boolean isConversionSupported(AudioFormat.Encoding encoding,
                              AudioFormat format)
```

is used to determine whether an audio input stream of the specified encoding can be obtained from an audio input stream that has the specified audio format. Similarly, the method:

```
boolean isConversionSupported(AudioFormat newFormat,
                              AudioFormat oldFormat)
```

tells us whether an `AudioInputStream` with the specified audio format, `newFormat`, can be obtained through the conversion of an `AudioInputStream` that has the audio format `oldFormat`. (This method was invoked in the previous section's code excerpt that created a low-resolution audio input stream, `lowResAIS`.)

These format-related queries help prevent errors when attempting to perform format conversions with the Java Sound API.

Chapter 8: Overview of the MIDI Package

Chapter 1,"Introduction to Java Sound," gave a glimpse into the MIDI capabilities of the Java™ Sound API. The present chapter assumes you've read Chapter 1. The discussion here provides a more detailed introduction to the Java Sound API's MIDI architecture, which is accessed through the `javax.sound.midi` package. Some basic features of MIDI itself are explained, as a refresher or introduction, to place the Java Sound API's MIDI features in context. The chapter then goes on to discuss the Java Sound API's approach to MIDI, as a preparation for the programming tasks that are explained in subsequent chapters. This chapter's discussion of the MIDI API is divided into two main areas: data and devices.

A MIDI Refresher: Wires and Files

The Musical Instrument Digital Interface (MIDI) standard defines a communication protocol for electronic music devices, such as electronic keyboard instruments and personal computers. MIDI data can be transmitted over special cables during a live performance, and can also be stored in a standard type of file for later playback or editing.

This section reviews some MIDI basics, without reference to the Java Sound API. The discussion is intended as a refresher for readers acquainted with MIDI, and as a brief introduction for those who are not, to provide background for the subsequent discussion of the Java Sound API's MIDI package. If you have a thorough understanding of MIDI, you can safely skip this section. Before writing substantial MIDI applications, programmers who are unfamiliar with MIDI will probably need a fuller description of MIDI than can be included in this programmer's guide. See the Complete MIDI 1.0 Detailed Specification, which is available only in hard copy from http://www.midi.org (although you might find paraphrased or summarized versions on the Web).

MIDI is both a hardware specification and a software specification. To understand MIDI's design, it helps to understand its history. MIDI was originally designed for passing musical events, such as key depressions, between electronic keyboard instruments such as synthesizers. (As mentioned in Chapter 1, MIDI data consists primarily of control events that communicate a musician's gestures. MIDI data doesn't contain the audio that results from these events.) Hardware devices known as sequencers stored sequences of notes that could control a synthesizer, allowing musical performances to be recorded and subsequently played back. Later, hardware interfaces were developed that connected MIDI instruments to a computer's serial port, allowing sequencers to be implemented in software. More recently, computer sound cards have incorporated hardware for MIDI I/O and for synthesizing musical sound. Today, many users of MIDI deal only with sound cards, never connecting to external MIDI devices. CPUs have become fast enough that synthesizers, too, can be implemented in software. A sound card is needed only for audio I/O and, in some applications, for communicating with external MIDI devices.

The brief hardware portion of the MIDI specification prescribes the pinouts for MIDI cables and the jacks into which these cables are plugged. This portion need not concern us. Because devices that originally required hardware, such as sequencers and synthesizers, are now implementable in software, perhaps the only reason for most programmers to know anything about MIDI hardware devices is simply to understand the metaphors in MIDI. However, external MIDI hardware devices are still essential for some important music applications, and so the Java Sound API supports input and output of MIDI data.

The software portion of the MIDI specification is extensive. This portion concerns the structure of MIDI data and how devices such as synthesizers should respond to that data. It is important to understand that MIDI data can be *streamed* or *sequenced*. This duality reflects two different parts of the Complete MIDI 1.0 Detailed Specification:

- MIDI 1.0
- Standard MIDI Files

We'll explain what's meant by streaming and sequencing by examining the purpose of each of these two parts of the MIDI specification.

Streaming Data in the MIDI Wire Protocol

The first of these two parts of the MIDI specification describes what is known informally as "MIDI wire protocol." MIDI wire protocol, which is the original MIDI protocol, is based on the assumption that the MIDI data is being sent over a MIDI cable (the "wire"). The cable transmits digital data from one MIDI device to another. Each of the MIDI devices might be a musical instrument or a similar device, or it might be a general-purpose computer equipped with a MIDI-capable sound card or a MIDI-to-serial-port interface.

MIDI data, as defined by MIDI wire protocol, is organized into messages. The different kinds of message are distinguished by the first byte in the message, known as the *status byte*. (Status bytes are the only bytes that have the highest-order bit set to 1.) The bytes that follow the status byte in a message are known as *data bytes*. Certain MIDI messages, known as *channel* messages, have a status byte that contains four bits to specify the kind of channel message and another four bits to specify the channel number. There are therefore 16 MIDI channels; devices that receive MIDI messages can be set to respond to channel messages on all or only one of these virtual channels. Often each MIDI channel (which shouldn't be confused with a channel of audio) is used to send the notes for a different instrument. As an example, two common channel messages are Note On and Note Off, which start a note sounding and then stop it, respectively. These two messages each take two data bytes: the first specifies the note's pitch and the second its "velocity" (how fast the key is depressed or released, assuming a keyboard instrument is playing the note).

MIDI wire protocol defines a streaming model for MIDI data. A central feature of this protocol is that the bytes of MIDI data are delivered in real time—in other words, they are streamed. The data itself contains no timing information; each event is processed as it's received, and it's assumed that it arrives at the correct time. That model is fine if the notes are being generated by a live musician, but it's insufficient if you want to store the notes for later playback, or if you want to compose them out of real time. This limitation is understandable when you realize that MIDI was originally designed for musical performance, as a way for a keyboard musician to control more than one synthesizer, back in the days before many musicians used computers. (The first version of the specification was released in 1984.)

Sequenced Data in Standard MIDI Files

The Standard MIDI Files part of the MIDI specification addresses the timing limitation in MIDI wire protocol. A standard MIDI file is a digital file that contains MIDI *events*. An event is simply a MIDI message, as defined in the MIDI wire protocol, but with an additional piece of information that specifies the event's timing. (There are also some events that don't correspond to MIDI wire protocol messages, as we'll see in the next section.) The additional timing information is a series of bytes that indicates when to perform the operation described by the message. In other words, a standard MIDI file specifies not just which notes to play, but exactly when to play each of them. It's a bit like a musical score.

The information in a standard MIDI file is referred to as a *sequence*. A standard MIDI file contains one or more *tracks*. Each track typically contains the notes that a single instrument would play if the music were performed by live musicians. A sequencer is a software or hardware device that can read a sequence and deliver the MIDI messages contained in it at the right time. A sequencer is a bit like an orchestra conductor: it has the information for all the notes, including their timings, and it tells some other entity when to perform the notes.

The Java Sound API's Representation of MIDI Data

Now that we've sketched the MIDI specification's approach to streamed and sequenced musical data, let's examine how the Java Sound API represents that data.

MIDI Messages

`MidiMessage` is an abstract class that represents a "raw" MIDI message. A "raw" MIDI message is usually a message defined by the MIDI wire protocol. It can also be one of the events defined by the Standard MIDI Files specification, but without the event's timing information. There are three categories of raw MIDI message, represented in the Java Sound API by these three respective `MidiMessage` subclasses:

- `ShortMessages` are the most common messages and have at most two data bytes following the status byte. The channel messages, such as Note On and Note Off, are all short messages, as are some other messages.
- `SysexMessages` contain *system-exclusive* MIDI messages. They may have many bytes, and generally contain manufacturer-specific instructions.
- `MetaMessages` occur in MIDI files, but not in MIDI wire protocol. Meta messages contain data, such as lyrics or tempo settings, that might be useful to sequencers but that are usually meaningless for synthesizers.

MIDI Events

As we've seen, standard MIDI files contain events that are wrappers for "raw" MIDI messages along with timing information. An instance of the Java Sound API's `MidiEvent` class represents an event such as might be stored in a standard MIDI file.

The API for `MidiEvent` includes methods to set and get the event's timing value. There's also a method to retrieve its embedded raw MIDI message, which is an instance of a subclass of `MidiMessage`,

discussed next. (The embedded raw MIDI message can be set only when constructing the `MidiEvent`.)

Sequences and Tracks

As mentioned earlier, a standard MIDI file stores events that are arranged into tracks. Usually the file represents one musical composition, and usually each track represents a part such as might have been played by a single instrumentalist. Each note that the instrumentalist plays is represented by at least two events: a Note On that starts the note, and a Note Off that ends it. The track may also contain events that don't correspond to notes, such as meta-events (which were mentioned above).

The Java Sound API organizes MIDI data in a three-part hierarchy:

- `Sequence`
- `Track`
- `MidiEvent`

A `Track` is a collection of `MidiEvents`, and a `Sequence` is a collection of `Tracks`. This hierarchy reflects the files, tracks, and events of the Standard MIDI Files specification. (Note: this is a hierarchy in terms of containment and ownership; it's *not* a class hierarchy in terms of inheritance. Each of these three classes inherits directly from `java.lang.Object`.)

`Sequences` can be read from MIDI files, or created from scratch and edited by adding `Tracks` to the `Sequence` (or removing them). Similarly, `MidiEvents` can be added to or removed from the tracks in the sequence.

The Java Sound API's Representation of MIDI Devices

The previous section explained how MIDI messages are represented in the Java Sound API. However, MIDI messages don't exist in a vacuum. They're typically sent from one device to another. A program that uses the Java Sound API can generate MIDI messages from scratch, but more often the messages are instead created by a software device, such as a sequencer, or received from outside the computer through a MIDI input port. Such a device usually sends these messages to another device, such as a synthesizer or a MIDI output port.

The MidiDevice Interface

In the world of external MIDI hardware devices, many devices can transmit MIDI messages to other devices and also receive messages from other devices. Similarly, in the Java Sound API, software objects that implement the `MidiDevice` interface can transmit and receive messages. Such an object can be implemented purely in software, or it can serve as an interface to hardware such as a sound card's MIDI capabilities. The base `MidiDevice` interface provides all the functionality generally required by a MIDI input or output port. Synthesizers and sequencers, however, further implement one of the subinterfaces of `MidiDevice: Synthesizer` or `Sequencer`, respectively.

The `MidiDevice` interface includes API for opening and closing a device. It also includes an inner class called `MidiDevice.Info` that provides textual descriptions of the device, including its name, vendor, and version. If you've read the sampled-audio portion of this programmer's guide, this API will probably sound familiar, because its design is similar to that of the `javax.sampled.Mixer`

interface, which represents an audio device and which has an analogous inner class, `Mixer.Info`.

Transmitters and Receivers

Most MIDI devices are capable of sending `MidiMessages`, receiving them, or both. The way a device sends data is via one or more transmitter objects that it "owns." Similarly, the way a device receives data is via one or more of its receiver objects. The transmitter objects implement the `Transmitter` interface, and the receivers implement the `Receiver` interface.

Each transmitter can be connected to only one receiver at a time, and vice versa. A device that sends its MIDI messages to multiple other devices simultaneously does so by having multiple transmitters, each connected to a receiver of a different device. Similarly, a device that can receive MIDI messages from more than one source at a time must do so via multiple receivers.

Sequencers

A sequencer is a device for capturing and playing back sequences of MIDI events. It has transmitters, because it typically sends the MIDI messages stored in the sequence to another device, such as a synthesizer or MIDI output port. It also has receivers, because it can capture MIDI messages and store them in a sequence. To its superinterface, `MidiDevice`, `Sequencer` adds methods for basic MIDI sequencing operations. A sequencer can load a sequence from a MIDI file, query and set the sequence's tempo, and synchronize other devices to it. An application program can register an object to be notified when the sequencer processes certain kinds of events.

Synthesizers

A `Synthesizer` is a device for generating sound. It's the only object in the `javax.sound.midi` package that produces audio data. A synthesizer device controls a set of MIDI channel objects—typically 16 of them, since the MIDI specification calls for 16 MIDI channels. These MIDI channel objects are instances of a class that implements the `MidiChannel` interface, whose methods represent the MIDI specification's "channel voice messages" and "channel mode messages."

An application program can generate sound by directly invoking methods of a synthesizer's MIDI channel objects. More commonly, though, a synthesizer generates sound in response to messages sent to one or more of its receivers. These messages might be sent by a sequencer or MIDI input port, for example. The synthesizer parses each message that its receivers get, and usually dispatches a corresponding command (such as `noteOn` or `controlChange`) to one of its `MidiChannel` objects, according to the MIDI channel number specified in the event.

The `MidiChannel` uses the note information in these messages to synthesize music. For example, a `noteOn` message specifies the note's pitch and "velocity" (volume). However, the note information is insufficient; the synthesizer also requires precise instructions on how to create the audio signal for each note. These instructions are represented by an `Instrument`. Each `Instrument` typically emulates a different real-world musical instrument or sound effect. The `Instruments` might come as presets with the synthesizer, or they might be loaded from soundbank files. In the synthesizer, the `Instruments` are arranged by bank number (these can be thought of as rows) and program number (columns).

This chapter has provided a background for understanding MIDI data, and it has introduced some of the

important interfaces and classes related to MIDI in the Java Sound API. Subsequent chapters show how you can access and use these objects in your application program.

Chapter 9: Accessing MIDI System Resources

The Java™ Sound API offers a flexible model for MIDI system configuration, just as it does for configuration of the sampled-audio system. An implementation of the Java Sound API can itself provide different sorts of MIDI devices, and additional ones can be supplied by service providers and installed by users. You can write your program in such a way that it makes few assumptions about which specific MIDI devices are installed on the computer. Instead, the program can take advantage of the MIDI system's defaults, or it can allow the user to select from whatever devices happen to be available.

This section shows how your program can learn what MIDI resources have been installed, and how to get access to the desired ones. After you've accessed and opened the devices, you can connect them to each other, as discussed in the next chapter, "Transmitting and Receiving MIDI Messages."

The MidiSystem Class

The role of the `MidiSystem` class in the Java Sound API's MIDI package is directly analogous to the role of `AudioSystem` in the sampled-audio package. `MidiSystem` acts as a clearinghouse for accessing the installed MIDI resources.

You can query the `MidiSystem` to learn what sorts of devices are installed, and then you can iterate over the available devices and obtain access to the desired ones. For example, an application program might start out by asking the `MidiSystem` what synthesizers are available, and then display a list of them, from which the user can select one. A simpler application program might just use the system's default synthesizer.

The `MidiSystem` class also provides methods for translating between MIDI files and `Sequences`. It can report the file format of a MIDI file and can write files of different types.

An application program can obtain the following resources from the `MidiSystem`:

- Sequencers
- Synthesizers
- Transmitters (such as those associated with MIDI input ports)
- Receivers (such as those associated with MIDI output ports)
- Data from standard MIDI files
- Data from soundbank files

This chapter focuses on the first four of these types of resource. The `MidiSystem` class's file-handling

facilities are discussed in Chapter 11, "Playing, Recording, and Editing MIDI Sequences," and Chapter 12, "Synthesizing Sound." To understand how the MIDI system itself gets access to all these resources, see Part III of this guide, "Service Provider Interfaces."

Obtaining Default Devices

A typical MIDI application program that uses the Java Sound API begins by obtaining the devices it needs, which can consist of one or more sequencers, synthesizers, input ports, or output ports.

There is a default synthesizer device, a default sequencer device, a default transmitting device, and a default receiving device. The latter two devices normally represent the MIDI input and output ports, respectively, if there are any available on the system. (It's easy to get confused about the directionality here. Think of the ports' transmission or reception in relation to the software, not in relation to any external physical devices connected to the physical ports. A MIDI input port *transmits* data from an external device to a Java Sound API `Receiver`, and likewise a MIDI output port *receives* data from a software object and relays the data to an external device.)

A simple application program might just use the default instead of exploring all the installed devices. The `MidiSystem` class includes the following methods for retrieving default resources:

```
static Sequencer getSequencer()
static Synthesizer getSynthesizer()
static Receiver getReceiver()
static Transmitter getTransmitter()
```

The first two of these methods obtain the system's default sequencing and synthesis resources, which either represent physical devices or are implemented wholly in software. The `getReceiver` method obtains a `Receiver` object that takes MIDI messages sent to it and relays them to the default receiving device. Similarly, the getTransmitter method obtains a Transmitter object that can send MIDI messages to some receiver on behalf of the default transmitting device.

Learning What Devices Are Installed

Instead of using the default devices, a more thorough approach is to select the desired devices from the full set of devices that are installed on the system. An application program can select the desired devices programmatically, or it can display a list of available devices and let the user select which ones to use. The `MidiSystem` class provides a method for learning which devices are installed, and a corresponding method to obtain a device of a given type.

Here is the method for learning about the installed devices:

```
static MidiDevice.Info[] getMidiDeviceInfo()
```

As you can see, it returns an array of information objects. Each of these returned `MidiDevice.Info` objects identifies one type of sequencer, synthesizer, port, or other device that is installed. (Usually a system has at most one instance of a given type. For example, a given model of synthesizer from a certain vendor will be installed only once.) The `MidiDevice.Info` includes the following strings to describe the device:

- Name
- Version number
- Vendor (the company that created the device)
- A description of the device

You can display these strings in your user interface to let the user select from the list of devices.

However, to use the strings programmatically to select a device (as opposed to displaying the strings to the user), you need to know in advance what they might be. The company that provides each device should include this information in its documentation. An application program that requires or prefers a particular device can use this information to locate that device. This approach has the drawback of limiting the program to device implementations that it knows about in advance.

Another, more general, approach is to go ahead and iterate over the `MidiDevice.Info` objects, obtaining each corresponding device, and determining programmatically whether it's suitable to use (or at least suitable to include in a list from which the user can choose). The next section describes how to do this.

Obtaining a Desired Device

Once an appropriate device's info object is found, the application program invokes the following `MidiSystem` method to obtain the corresponding device itself:

```
static MidiDevice getMidiDevice(MidiDevice.Info info)
```

You can use this method if you've already found the info object describing the device you need. However, if you can't interpret the info objects returned by `getMidiDeviceInfo` to determine which device you need, and if you don't want to display information about all the devices to the user, you might be able to do the following instead: Iterate over all the `MidiDevice.Info` objects returned by `getMidiDeviceInfo`, get the corresponding devices using the method above, and test each device to see whether it's suitable. In other words, you can query each device for its class and its capabilities before including it in the list that you display to the user, or as a way to decide upon a device programmatically without involving the user. For example, if your program needs a synthesizer, you can obtain each of the installed devices, see which are instances of classes that implement the `Synthesizer` interface, and then display them in a list from which the user can choose one, as follows:

```
// Obtain information about all the installed synthesizers.
Vector synthInfos;
MidiDevice device;
MidiDevice.Info[] infos = MidiSystem.getMidiDeviceInfo();
for (int i = 0; i < infos.length; i++) {
    try {
        device = MidiSystem.getMidiDevice(infos[i]);
    } catch (MidiUnavailableException e) {
          // Handle or throw exception...
    }
    if (device instanceof Synthesizer) {
```

```
            synthInfos.add(infos[i]);
        }
    }
    // Now, display strings from synthInfos list in GUI.
```

As another example, you might choose a device programmatically, without involving the user. Let's suppose you want to obtain the synthesizer that can play the most notes simultaneously. You iterate over all the MidiDevice.Info objects, as above, but after determining that a device is a synthesizer, you query its capabilities by invoking the `getMaxPolyphony` method of `Synthesizer`. You reserve the synthesizer that has the greatest polyphony, as described in the next section. Even though you're not asking the user to choose a synthesizer, you might still display strings from the chosen `MidiDevice.Info` object, just for the user's information.

Opening Devices

The previous section showed how to get an installed device. However, a device might be installed but unavailable. For example, another application program might have exclusive use of it. To actually reserve a device for your program, you need to use the `MidiDevice` method `open`:

```
if (!(device.isOpen())) {
    try {
      device.open();
  } catch (MidiUnavailableException e) {
          // Handle or throw exception...
  }
}
```

Once you've accessed a device and reserved it by opening it, you'll probably want to connect it to one or more other devices to let MIDI data flow between them. This procedure is described in Chapter 10, "Transmitting and Receiving MIDI Messages."

When done with a device, you release it for other programs' use by invoking the `close` method of `MidiDevice`.

Chapter 10: Transmitting and Receiving MIDI Messages

Understanding Devices, Transmitters, and Receivers

The Java™ Sound API specifies a message-routing architecture for MIDI data that's flexible and easy to use, once you understand how it works. The system is based on a module-connection design: distinct modules, each of which performs a specific task, can be interconnected (networked), enabling data to flow from one module to another.

The base module in the Java Sound API's messaging system is `MidiDevice` (a Java language interface). `MidiDevices` include sequencers (which record, play, load, and edit sequences of time-stamped MIDI messages), synthesizers (which generate sounds when triggered by MIDI messages), and MIDI input and output ports, through which data comes from and goes to external MIDI devices. The functionality typically required of MIDI ports is described by the base `MidiDevice` interface. The `Sequencer` and `Synthesizer` interfaces extend the `MidiDevice` interface to describe the additional functionality characteristic of MIDI sequencers and synthesizers, respectively. Concrete classes that function as sequencers or synthesizers should implement these interfaces.

A `MidiDevice` typically owns one or more ancillary objects that implement the `Receiver` or `Transmitter` interfaces. These interfaces represent the "plugs" or "portals" that connect devices together, permitting data to flow into and out of them. By connecting a `Transmitter` of one `MidiDevice` to a `Receiver` of another, you can create a network of modules in which data flows from one to another.

The `MidiDevice` interface includes methods for determining how many transmitter and receiver objects the device can support concurrently, and other methods for accessing those objects. A MIDI output port normally has at least one `Receiver` through which the outgoing messages may be received; similarly, a synthesizer normally responds to messages sent to its `Receiver` or `Receivers`. A MIDI input port normally has at least one `Transmitter`, which propagates the incoming messages. A full-featured sequencer supports both `Receivers`, which receive messages during recording, and `Transmitters`, which send messages during playback.

The `Transmitter` interface includes methods for setting and querying the receivers to which the transmitter sends its `MidiMessages`. Setting the receiver establishes the connection between the two. The `Receiver` interface contains a method that sends a `MidiMessage` to the receiver. Typically, this method is invoked by a `Transmitter`. Both the `Transmitter` and `Receiver` interfaces include a `close` method that frees up a previously connected transmitter or receiver, making it available for a different connection.

We'll now examine how to use transmitters and receivers. Before getting to the typical case of connecting two devices (such as hooking a sequencer to a synthesizer), we'll examine the simpler case where you send a MIDI message directly from your application program to a device. Studying this simple scenario should make it easier to understand how the Java Sound API arranges for sending MIDI messages between two devices.

Sending a Message to a Receiver without Using a Transmitter

Let's say you want to create a MIDI message from scratch and then send it to some receiver. You can create a new, blank `ShortMessage` and then fill it with MIDI data using the following `ShortMessage` method:

```
void setMessage(int command, int channel, int data1,
        int data2)
```

Once you have a message ready to send, you can send it to a `Receiver` object, using this `Receiver` method:

```
void send(MidiMessage message, long timeStamp)
```

The time-stamp argument will be explained momentarily. For now, we'll just mention that its value can be set to -1 if you don't care about specifying a precise time. In this case, the device receiving the message will try to respond to the message as soon as possible.

An application program can obtain a receiver for a `MidiDevice` by invoking the device's `getReceiver` method. If the device can't provide a receiver to the program (typically because all the device's receivers are already in use), a `MidiUnavailableException` is thrown. Otherwise, the receiver returned from this method is available for immediate use by the program. When the program has finished using the receiver, it should call the receiver's `close` method. If the program attempts to invoke methods on a receiver after calling `close`, an `IllegalStateException` may be thrown.

As a concrete simple example of sending a message without using a transmitter, let's send a Note On message to the default receiver, which is typically associated with a device such as the MIDI output port or a synthesizer. We do this by creating a suitable `ShortMessage` and passing it as an argument to `Receiver's send` method:

```
ShortMessage myMsg = new ShortMessage();
// Start playing the note Middle C (60),
// moderately loud (velocity = 93).
myMsg.setMessage(ShortMessage.NOTE_ON, 0, 60, 93);
long timeStamp = -1;
Receiver       rcvr = MidiSystem.getReceiver();
rcvr.send(myMsg, timeStamp);
```

This code uses a static integer field of `ShortMessage`, namely, `NOTE_ON`, for use as the MIDI

message's status byte. The other parts of the MIDI message are given explicit numeric values as arguments to the `setMessage` method. The zero indicates that the note is to be played using MIDI channel number 1; the 60 indicates the note Middle C; and the 93 is an arbitrary key-down velocity value, which typically indicates that the synthesizer that eventually plays the note should play it somewhat loudly. (The MIDI specification leaves the exact interpretation of velocity up to the synthesizer's implementation of its current instrument.) This MIDI message is then sent to the receiver with a time stamp of -1. We now need to examine exactly what the time stamp parameter means, which is the subject of the next section.

Understanding Time Stamps

Chapter 8, "Overview of the MIDI Package," explained that the MIDI specification has different parts. One part describes MIDI "wire" protocol (messages sent between devices in real time), and another part describes Standard MIDI Files (messages stored as events in "sequences"). In the latter part of the specification, each event stored in a standard MIDI file is tagged with a timing value that indicates when that event should be played. By contrast, messages in MIDI wire protocol are always supposed to be processed immediately, as soon as they're received by a device, so they have no accompanying timing values.

The Java Sound API adds an additional twist. It comes as no surprise that timing values are present in the `MidiEvent` objects that are stored in sequences (as might be read from a MIDI file), just as in the Standard MIDI Files specification. But in the Java Sound API, even the messages sent between devices—in other words, the messages that correspond to MIDI wire protocol—can be given timing values, known as *time stamps*. It is these time stamps that concern us here. (The timing values in `MidiEvent` objects are discussed in detail in Chapter 11, "Playing, Recording, and Editing MIDI Sequences.")

Time Stamps on Messages Sent to Devices

The time stamp that can optionally accompany messages sent between devices in the Java Sound API is quite different from the timing values in a standard MIDI file. The timing values in a MIDI file are often based on musical concepts such as beats and tempo, and each event's timing measures the time elapsed since the previous event. In contrast, the time stamp on a message sent to a device's `Receiver` object always measures absolute time in microseconds. Specifically, it measures the number of microseconds elapsed since the device that owns the receiver was opened.

This kind of time stamp is designed to help compensate for latencies introduced by the operating system or by the application program. It's important to realize that these time stamps are used for minor adjustments to timing, not to implement complex queues that can schedule events at completely arbitrary times (as `MidiEvent` timing values do).

The time stamp on a message sent to a device (through a `Receiver`) can provide precise timing information to the device. The device might use this information when it processes the message. For example, it might adjust the event's timing by a few milliseconds to match the information in the time stamp. On the other hand, not all devices support time stamps, so the device might completely ignore the message's time stamp.

Even if a device supports time stamps, it might not schedule the event for exactly the time that you requested. You can't expect to send a message whose time stamp is very far in the future and have the device handle it as you intended, and you certainly can't expect a device to correctly schedule a message whose time stamp is in the past! It's up to the device to decide how to handle time stamps that are too far off in the future or are in the past. The sender doesn't know what the device considers to be too far off, or whether the device had any problem with the time stamp. This ignorance mimics the behavior of external MIDI hardware devices, which send messages without ever knowing whether they were received correctly. (MIDI wire protocol is unidirectional.)

Some devices send time-stamped messages (via a `Transmitter`). For example, the messages sent by a MIDI input port might be stamped with the time the incoming message arrived at the port. On some systems, the event-handling mechanisms cause a certain amount of timing precision to be lost during subsequent processing of the message. The message's time stamp allows the original timing information to be preserved.

To learn whether a device supports time stamps, invoke the following method of `MidiDevice`:

```
long getMicrosecondPosition()
```

This method returns -1 if the device ignores time stamps. Otherwise, it returns the device's current notion of time, which you as the sender can use as an offset when determining the time stamps for messages you subsequently send. For example, if you want to send a message with a time stamp for five milliseconds in the future, you can get the device's current position in microseconds, add 5000 microseconds, and use that as the time stamp. Keep in mind that the `MidiDevice's` notion of time always places time zero at the time the device was opened.

Now, with all that explanation of time stamps as a background, let's return to the `send` method of `Receiver`:

```
void send(MidiMessage message, long timeStamp)
```

The `timeStamp` argument is expressed in microseconds, according to the receiving device's notion of time. If the device doesn't support time stamps, it simply ignores the `timeStamp` argument. You aren't required to time-stamp the messages you send to a receiver. You can use -1 for the `timeStamp` argument to indicate that you don't care about adjusting the exact timing; you're just leaving it up to the receiving device to process the message as soon as it can. However, it's not advisable to send -1 with some messages and explicit time stamps with other messages sent to the same receiver. Doing so is likely to cause irregularities in the resultant timing.

Connecting Transmitters to Receivers

We've seen how you can send a MIDI message directly to a receiver, without using a transmitter. Now let's look at the more common case, where you aren't creating MIDI messages from scratch, but are simply connecting devices together so that one of them can send MIDI messages to the other.

Connecting to a Single Device

The specific case we'll take as our first example is connecting a sequencer to a synthesizer. After this connection is made, starting the sequencer running will cause the synthesizer to generate audio from the

events in the sequencer's current sequence. For now, we'll ignore the process of loading a sequence from a MIDI file into the sequencer. Also, we won't go into the mechanism of playing the sequence. Loading and playing sequences is discussed in detail in Chapter 11, "Playing, Recording, and Editing MIDI Sequences." Loading instruments into the synthesizer is discussed in Chapter 12, "Synthesizing Sound." For now, all we're interested in is how to make the connection between the sequencer and the synthesizer. This will serve as an illustration of the more general process of connecting one device's transmitter to another device's receiver.

For simplicity, we'll use the default sequencer and the default synthesizer. (See Chapter 9, "Accessing MIDI System Resources," for more about default devices and how to access non-default devices.)

```
Sequencer           seq;
Transmitter         seqTrans;
Synthesizer         synth;
Receiver         synthRcvr;
try {
      seq     = MidiSystem.getSequencer();
      seqTrans = seq.getTransmitter();
      synth   = MidiSystem.getSynthesizer();
      synthRcvr = synth.getReceiver();
      seqTrans.setReceiver(synthRcvr);
} catch (MidiUnavailableException e) {
      // handle or throw exception
}
```

An implementation might actually have a single object that serves as both the default sequencer and the default synthesizer. In other words, the implementation might use a class that implements both the `Sequencer` interface and the `Synthesizer` interface. In that case, it probably wouldn't be necessary to make the explicit connection that we did in the code above. For portability, though, it's safer not to assume such a configuration. If desired, you can test for this condition, of course:

```
if (seq instanceof Synthesizer)
```

although the explicit connection above should work in any case.

Connecting to More than One Device

The previous code example illustrated a one-to-one connection between a transmitter and a receiver. But, what if you need to send the same MIDI message to multiple receivers? For example, suppose you want to capture MIDI data from an external device to drive the internal synthesizer while simultaneously recording the data to a sequence. This form of connection, sometimes referred to as "fan out" or as a "splitter," is straightforward. The following statements show how to create a fan-out connection, through which the MIDI messages arriving at the MIDI input port are sent to both a `Synthesizer` object and a `Sequencer` object. We assume you've already obtained and opened the three devices: the input port, sequencer, and synthesizer. (To obtain the input port, you'll need to iterate over all the items returned by `MidiSystem.getMidiDeviceInfo`.)

```
Synthesizer  synth;
Sequencer    seq;
MidiDevice   inputPort;
// [obtain and open the three devices...]
Transmitter   inPortTrans1, inPortTrans2;
Receiver            synthRcvr;
Receiver            seqRcvr;
try {
      inPortTrans1 = inputPort.getTransmitter();
      synthRcvr = synth.getReceiver();
      inPortTrans1.setReceiver(synthRcvr);
      inPortTrans2 = inputPort.getTransmitter();
      seqRcvr = seq.getReceiver();
      inPortTrans2.setReceiver(seqRcvr);
} catch (MidiUnavailableException e) {
      // handle or throw exception
}
```

This code introduces a dual invocation of the `MidiDevice.getTransmitter` method, assigning the results to `inPortTrans1` and `inPortTrans2`. As mentioned earlier, a device can own multiple transmitters and receivers. Each time `MidiDevice.getTransmitter()` is invoked for a given device, another transmitter is returned, until no more are available, at which time an exception will be thrown.

To learn how many transmitters and receivers a device supports, you can use the following `MidiDevice` method:

```
int getMaxTransmitters()
int getMaxReceivers()
```

These methods return the total number owned by the device, not the number currently available.

A transmitter can transmit MIDI messages to only one receiver at a time. (Every time you call `Transmitter's setReceiver` method, the existing `Receiver`, if any, is replaced by the newly specified one. You can tell whether the transmitter currently has a receiver by invoking `Transmitter.getReceiver`.) However, if a device has multiple transmitters, it can send data to more than one device at a time, by connecting each transmitter to a different receiver, as we saw in the case of the input port above.

Similarly, a device can use its multiple receivers to receive from more than one device at a time. The multiple-receiver code that's required is straightforward, being directly analogous to the multiple-transmitter code above. It's also possible for a single receiver to receive messages from more than one transmitter at a time.

Closing Connections

Once you're done with a connection, you can free up its resources by invoking the `close` method for each transmitter and receiver that you've obtained. The `Transmitter` and `Receiver` interfaces each have a `close` method. Note that invoking `Transmitter.setReceiver` doesn't close the transmitter's current receiver. The receiver is left open, and it can still receive messages from any other transmitter that's connected to it.

If you're also done with the devices, you can similarly make them available to other application programs by invoking `MidiDevice.close()`. Closing a device automatically closes all its transmitters and receivers.

Chapter 11: Playing, Recording and Editing MIDI Sequences

In the world of MIDI, a *sequencer* is any hardware or software device that can precisely play or record a *sequence* of time-stamped MIDI messages. Similarly, in the Java™ Sound API, the `Sequencer` abstract interface defines the properties of an object that can play and record `Sequences` of `MidiEvent` objects. A `Sequencer` typically loads these `MidiEvent` sequences from a standard MIDI file or saves them to such a file. Sequences can also be edited. This chapter explains how to use `Sequencer` objects, along with related classes and interfaces, to accomplish such tasks.

Introduction to Sequencers

To develop an intuitive understanding of what a `Sequencer` is, think of it by analogy with a tape recorder, which a sequencer resembles in many respects. Whereas a tape recorder plays audio, a sequencer plays MIDI data. A sequence is a multi-track, linear, time-ordered recording of MIDI musical data, which a sequencer can play at various speeds, rewind, shuttle to particular points, record into, or copy to a file for storage.

Chapter 10, "Transmitting and Receiving MIDI Messages," explained that devices typically have `Receiver` objects, `Transmitter` objects, or both. To *play* music, a device generally receives `MidiMessages` through a `Receiver`, which in turn has usually received them from a `Transmitter` that belongs to a `Sequencer`. The device that owns this `Receiver` might be a `Synthesizer`, which will generate audio directly, or it might be a MIDI output port, which transmits MIDI data through a physical cable to some external piece of equipment. Similarly, to *record* music, a series of time-stamped `MidiMessages` are generally sent to a `Receiver` owned by a `Sequencer`, which places them in a `Sequence` object. Typically the object sending the messages is a `Transmitter` associated with a hardware input port, and the port relays MIDI data that it gets from an external instrument. However, the device responsible for sending the messages might instead be some other `Sequencer`, or any other device that has a `Transmitter`. Furthermore, as mentioned in Chapter 10, a program can send messages without using any `Transmitter` at all.

A `Sequencer` itself has both `Receivers` and `Transmitters`. When it's recording, it actually obtains `MidiMessages` via its `Receivers`. During playback, it uses its `Transmitters` to send `MidiMessages` that are stored in the `Sequence` that it has recorded (or loaded from a file).

One way to think of the role of a `Sequencer` in the Java Sound API is as an aggregator and "de-aggregator" of `MidiMessages`. A series of separate `MidiMessages`, each of which is independent, is sent to the `Sequencer` along with its own time stamp that marks the timing of a musical event. These `MidiMessages` are encapsulated in `MidiEvent` objects and collected in

`Sequence` objects through the action of the `Sequencer.record` method. A `Sequence` is a data structure containing aggregates of `MidiEvents`, and it usually represents a series of musical notes, often an entire song or composition. On playback, the `Sequencer` again extracts the `MidiMessages` from the `MidiEvent` objects in the `Sequence` and then transmits them to one or more devices that will either render them into sound, save them, modify them, or pass them on to some other device.

Some sequencers might have neither transmitters nor receivers. For example, they might create `MidiEvents` from scratch as a result of keyboard or mouse events, instead of receiving `MidiMessages` through `Receivers`. Similarly, they might play music by communicating directly with an internal synthesizer (which could actually be the same object as the sequencer) instead of sending `MidiMessages` to a `Receiver` associated with a separate object. However, the rest of this chapter assumes the normal case of a sequencer that uses `Receivers` and `Transmitters`.

When to Use a Sequencer

It's possible for an application program to send MIDI messages directly to a device, without using a sequencer, as was described in Chapter 10, "Transmitting and Receiving MIDI Messages." The program simply invokes the `Receiver.send` method each time it wants to send a message. This is a straightforward approach that's useful when the program itself creates the messages in real time. For example, consider a program that lets the user play notes by clicking on an onscreen piano keyboard. When the program gets a mouse-down event, it immediately sends the appropriate Note On message to the synthesizer.

As mentioned in Chapter 10, the program can include a time stamp with each MIDI message it sends to the device's receiver. However, such time stamps are used only for fine-tuning the timing, to correct for processing latency. The caller can't generally set arbitrary time stamps; the time value passed to `Receiver.send` must be close to the present time, or the receiving device might not be able to schedule the message correctly. This means that if an application program wanted to create a queue of MIDI messages for an entire piece of music ahead of time (instead of creating each message in response to a real-time event), it would have to be very careful to schedule each invocation of `Receiver.send` for nearly the right time.

Fortunately, most application programs don't have to be concerned with such scheduling. Instead of invoking `Receiver.send` itself, a program can use a `Sequencer` object to manage the queue of MIDI messages for it. The sequencer takes care of scheduling and sending the messages—in other words, playing the music with the correct timing. Generally, it's advantageous to use a sequencer whenever you need to convert a non-real-time series of MIDI messages to a real-time series (as in playback), or vice versa (as in recording). Sequencers are most commonly used for playing data from MIDI files and for recording data from a MIDI input port.

Understanding Sequence Data

Before examining the `Sequencer` API, it helps to understand the kind of data that's stored in a sequence.

Sequences and Tracks

In the Java Sound API, sequencers closely follow the Standard MIDI Files specification in the way that they organize recorded MIDI data. As mentioned above, a `Sequence` is an aggregation of `MidiEvents`, organized in time. But there is more structure to a `Sequence` than just a linear series of `MidiEvents`: a `Sequence` actually contains global timing information plus a collection of `Tracks`, and it is the `Tracks` themselves that hold the `MidiEvent` data. So the data played by a sequencer consists of a three-level hierarchy of objects: `Sequencer`, `Track`, and `MidiEvent`.

In the conventional use of these objects, the `Sequence` represents a complete musical composition or section of a composition, with each `Track` corresponding to a voice or player in the ensemble. In this model, all the data on a particular `Track` would also therefore be encoded into a particular MIDI channel reserved for that voice or player.

This way of organizing data is convenient for purposes of editing sequences, but note that this is just a conventional way to use `Tracks`. There is nothing in the definition of the `Track` class per se that keeps it from containing a mix of `MidiEvents` on different MIDI channels. For example, an entire multi-channel MIDI composition can be mixed and recorded onto one `Track`. Also, standard MIDI files of Type 0 (as opposed to Type 1 and Type 2) contain by definition only one track; so a `Sequence` that's read from such a file will necessarily have a single `Track` object.

MidiEvents and Ticks

As discussed in Chapter 8, "Overview of the MIDI Package," the Java Sound API includes `MidiMessage` objects that correspond to the raw two- or three-byte sequences that make up most standard MIDI messages. A `MidiEvent` is simply a packaging of a `MidiMessage` along with an accompanying timing value that specifies when the event occurs. (We might then say that a sequence really consists of a four- or five-level hierarchy of data, rather than three-level, because the ostensible lowest level, `MidiEvent`, actually contains a lower-level `MidiMessage`, and likewise the `MidiMessage` object contains an array of bytes that comprises a standard MIDI message.)

In the Java Sound API, there are two different ways in which `MidiMessages` can be associated with timing values. One is the way mentioned above under "When to Use a Sequencer." This technique is described in detail under "Sending a Message to a Receiver without Using a Transmitter" and "Understanding Time Stamps" in Chapter 10, "Transmitting and Receiving MIDI Messages." There, we saw that the `send` method of `Receiver` takes a `MidiMessage` argument and a time-stamp argument. That kind of time stamp can only be expressed in microseconds.

The other way in which a `MidiMessage` can have its timing specified is by being encapsulated in a `MidiEvent`. In this case, the timing is expressed in slightly more abstract units called *ticks*.

What is the duration of a tick? It can vary between sequences (but not within a sequence), and its value is stored in the header of a standard MIDI file. The size of a tick is given in one of two types of units:

- Pulses (ticks) per quarter note, abbreviated as PPQ
- Ticks per frame, also known as SMPTE time code (a standard adopted by the Society of Motion Picture and Television Engineers)

If the unit is PPQ, the size of a tick is expressed as a fraction of a quarter note, which is a relative, not absolute, time value. A quarter note is a musical duration value that often corresponds to one beat of the

music (a quarter of a measure in 4/4 time). The duration of a quarter note is dependent on the tempo, which can vary during the course of the music if the sequence contains tempo-change events. So if the sequence's timing increments (ticks) occur, say 96 times per quarter note, each event's timing value measures that event's position in musical terms, not as an absolute time value.

On the other hand, in the case of SMPTE, the units measure absolute time, and the notion of tempo is inapplicable. There are actually four different SMPTE conventions available, which refer to the number of motion-picture frames per second. The number of frames per second can be 24, 25, 29.97, or 30. With SMPTE time code, the size of a tick is expressed as a fraction of a frame.

In the Java Sound API, you can invoke `Sequence.getDivisionType` to learn which type of unit—namely, PPQ or one of the SMPTE units—is used in a particular sequence. You can then calculate the size of a tick after invoking `Sequence.getResolution`. The latter method returns the number of ticks per quarter note if the division type is PPQ, or per SMPTE frame if the division type is one of the SMPTE conventions. You can get the size of a tick using this formula in the case of PPQ:

```
ticksPerSecond =
    resolution * (currentTempoInBeatsPerMinute / 60.0);
tickSize = 1.0 / ticksPerSecond;
```

and this formula in the case of SMPTE:

```
framesPerSecond =
  (divisionType == Sequence.SMPTE_24 ? 24
    : (divisionType == Sequence.SMPTE_25 ? 25
      : (divisionType == Sequence.SMPTE_30 ? 30
        : (divisionType == Sequence.SMPTE_30DROP ?

            29.97))));
ticksPerSecond = resolution * framesPerSecond;
tickSize = 1.0 / ticksPerSecond;
```

The Java Sound API's definition of timing in a sequence mirrors that of the Standard MIDI Files specification. However, there's one important difference. The tick values contained in `MidiEvents` measure *cumulative* time, rather than *delta* time. In a standard MIDI file, each event's timing information measures the amount of time elapsed since the onset of the previous event in the sequence. This is called delta time. But in the Java Sound API, the ticks aren't delta values; they're the previous event's time value *plus* the delta value. In other words, in the Java Sound API the timing value for each event is always greater than that of the previous event in the sequence (or equal, if the events are supposed to be simultaneous). Each event's timing value measures the time elapsed since the beginning of the sequence.

To summarize, the Java Sound API expresses timing information in either MIDI ticks or microseconds. `MidiEvents` store timing information in terms of MIDI ticks. The duration of a tick can be calculated from the `Sequence's` global timing information and, if the sequence uses tempo-based timing, the current musical tempo. The time stamp associated with a `MidiMessage` sent to a `Receiver`, on the other hand, is always expressed in microseconds.

One goal of this design is to avoid conflicting notions of time. It's the job of a `Sequencer` to interpret the time units in its `MidiEvents`, which might have PPQ units, and translate these into absolute time in microseconds, taking the current tempo into account. The sequencer must also express the microseconds relative to the time when the device receiving the message was opened. Note that a sequencer can have multiple transmitters, each delivering messages to a different receiver that might be associated with a completely different device. You can see, then, that the sequencer has to be able to perform multiple translations at the same time, making sure that each device receives time stamps appropriate for its notion of time.

To make matters more complicated, different devices might update their notions of time based on different sources (such as the operating system's clock, or a clock maintained by a sound card). This means that their timings can drift relative to the sequencer's. To keep in synchronization with the sequencer, some devices permit themselves to be "slaves" to the sequencer's notion of time. Setting masters and slaves is discussed later under "Using Advanced Sequencer Features."

Overview of Sequencer Methods

The `Sequencer` interface provides methods in several categories:

- Methods to load sequence data from a MIDI file or a `Sequence` object, and to save the currently loaded sequence data to a MIDI file.
- Methods analogous to the transport functions of a tape recorder, for stopping and starting playback and recording, enabling and disabling recording on specific tracks, and shuttling the current playback or recording position in a `Sequence`.
- Advanced methods for querying and setting the synchronization and timing parameters of the object. A `Sequencer` may play at different tempos, with some `Tracks` muted, and in various synchronization states with other objects.
- Advanced methods for registering "listener" objects that are notified when the `Sequencer` processes certain kinds of MIDI events.

Regardless of which `Sequencer` methods you'll invoke, the first step is to obtain a `Sequencer` device from the system and reserve it for your program's use.

Obtaining a Sequencer

An application program doesn't instantiate a `Sequencer`; after all, `Sequencer` is just an interface. Instead, like all devices in the Java Sound API's MIDI package, a `Sequencer` is accessed through the static `MidiSystem` object. As mentioned in Chapter 9, "Accessing MIDI System Resources," the following `MidiSystem` method can be used to obtain the default `Sequencer`:

```
static Sequencer getSequencer()
```

The following code fragment obtains the default `Sequencer`, acquires any system resources it needs, and makes it operational:

```
Sequencer sequencer;
// Get default sequencer.
```

```
sequencer = MidiSystem.getSequencer();
if (sequencer == null) {
    // Error -- sequencer device is not supported.
    // Inform user and return...
} else {
     // Acquire resources and make operational.
    sequencer.open();
}
```

The invocation of `open` reserves the sequencer device for your program's use. It doesn't make much sense to imagine sharing a sequencer, because it can play only one sequence at a time. When you're done using the sequencer, you can make it available to other programs by invoking `close`.

Non-default sequencers can be obtained as described in Chapter 9, "Accessing MIDI System Resources."

Loading a Sequence

Having obtained a sequencer from the system and reserved it, you then need load the data that the sequencer should play. There are three typical ways of accomplishing this:

- Reading the sequence data from a MIDI file
- Recording it in real time by receiving MIDI messages from another device, such as a MIDI input port
- Building it programmatically "from scratch" by adding tracks to an empty sequence and adding `MidiEvent` objects to those tracks

We'll now look at the first of these ways of getting sequence data. (The other two ways are described later under "Recording and Saving Sequences" and "Editing a Sequence," respectively.) This first way actually encompasses two slightly different approaches. One approach is to feed MIDI file data to an `InputStream` that you then read directly to the sequencer by means of `Sequencer.setSequence(InputStream)`. With this approach, you don't explicitly create a `Sequence` object. In fact, the `Sequencer` implementation might not even create a `Sequence` behind the scenes, because some sequencers have a built-in mechanism for handling data directly from a file.

The other approach is to create a `Sequence` explicitly. You'll need to use this approach if you're going to edit the sequence data before playing it. With this approach, you invoke `MidiSystem's` overloaded method `getSequence`. The method is able to get the sequence from an `InputStream`, a `File`, or a `URL`. The method returns a `Sequence` object that can then be loaded into a `Sequencer` for playback. Expanding on the previous code excerpt, here's an example of obtaining a `Sequence` object from a `File` and loading it into our `sequencer`:

```
try {
    File myMidiFile = new File("seq1.mid");
    // Construct a Sequence object, and
    // load it into my sequencer.
     Sequence mySeq = MidiSystem.getSequence(myMidiFile);
    sequencer.setSequence(mySeq);
```

```
} catch (Exception e) {
    // Handle error and/or return
}
```

Like `MidiSystem's getSequence` method, `setSequence` may throw an `InvalidMidiDataException`—and, in the case of the `InputStream` variant, an `IOException`—if it runs into any trouble.

Playing a Sequence

Starting and stopping a `Sequencer` is accomplished using the following methods:

```
void start()
```

and

```
void stop()
```

The `Sequencer.start` method begins playback of the sequence. Note that playback starts at the current position in a sequence. Loading an existing sequence using the `setSequence` method, described above, initializes the sequencer's current position to the very beginning of the sequence. The `stop` method stops the sequencer, but it does not automatically rewind the current `Sequence`. Starting a stopped `Sequence` without resetting the position simply resumes playback of the sequence from the current position. In this case, the `stop` method has served as a pause operation. However, there are various `Sequencer` methods for setting the current sequence position to an arbitrary value before playback is started. (We'll discuss these methods below.)

As mentioned earlier, a `Sequencer` typically has one or more `Transmitter` objects, through which it sends `MidiMessages` to a `Receiver`. It is through these `Transmitters` that a `Sequencer` plays the `Sequence`, by emitting appropriately timed `MidiMessages` that correspond to the `MidiEvents` contained in the current `Sequence`. Therefore, part of the setup procedure for playing back a `Sequence` is to invoke the `setReceiver` method on the `Sequencer's Transmitter` object, in effect wiring its output to the device that will make use of the played-back data. For more details on `Transmitters` and `Receivers`, see Chapter 10, "Transmitting and Receiving MIDI Messages."

Recording and Saving Sequences

To capture MIDI data to a `Sequence`, and subsequently to a file, you need to perform some additional steps beyond those described above. The following outline shows the steps necessary to set up for recording to a `Track` in a `Sequence`:

1. Use `MidiSystem.getSequencer` to get a new sequencer to use for recording, as above.
2. Set up the "wiring" of the MIDI connections. The object that is transmitting the MIDI data to be recorded should be configured, through its `setReceiver` method, to send data to a `Receiver` associated with the recording `Sequencer`.
3. Create a new `Sequence` object, which will store the recorded data. When you create the

`Sequence` object, you must specify the global timing information for the sequence. For example:

```
Sequence mySeq;
try{
    mySeq = new Sequence(Sequence.PPQ, 10);
} catch (Exception ex) {
    ex.printStackTrace();
}
```

The constructor for `Sequence` takes as arguments a `divisionType` and a timing resolution. The `divisionType` argument specifies the units of the resolution argument. In this case, we've specified that the timing resolution of the `Sequence` we're creating will be 10 pulses per quarter note. An additional optional argument to the `Sequence` constructor is a number of tracks argument, which would cause the initial sequence to begin with the specified number of (initially empty) `Tracks`. Otherwise the `Sequence` will be created with no initial `Tracks`; they can be added later as needed.

4. Create an empty `Track` in the `Sequence`, with `Sequence.createTrack`. This step is unnecessary if the `Sequence` was created with initial `Tracks`.
5. Using `Sequencer.setSequence`, select our new `Sequence` to receive the recording. The `setSequence` method ties together an existing `Sequence` with the `Sequencer`, which is somewhat analogous to loading a tape onto a tape recorder.
6. Invoke `Sequencer.recordEnable` for each `Track` to be recorded. If necessary, get a reference to the available `Tracks` in the `Sequence` by invoking `Sequence.getTracks`.
7. Invoke `startRecording` on the `Sequencer`.
8. When done recording, invoke `Sequencer.stop` or `Sequencer.stopRecording`.
9. Save the recorded `Sequence` to a MIDI file with `MidiSystem.write`. The `write` method of `MidiSystem` takes a `Sequence` as one of its arguments, and will write that `Sequence` to a stream or file.

Editing a Sequence

Many application programs allow a sequence to be created by loading it from a file, and quite a few also allow a sequence to be created by capturing it from live MIDI input (that is, recording). Some programs, however, will need to create MIDI sequences from scratch, whether programmatically or in response to user input. Full-featured sequencer programs permit the user to manually construct new sequences, as well as to edit existing ones.

These data-editing operations are achieved in the Java Sound API not by `Sequencer` methods, but by methods of the data objects themselves: `Sequence`, `Track`, and `MidiEvent`. You can create an empty sequence using one of the `Sequence` constructors, and then add tracks to it by invoking the following `Sequence` method:

```
Track createTrack()
```

If your program allows the user to edit sequences, you'll need this `Sequence` method to remove tracks:

```
boolean deleteTrack(Track track)
```

Once the sequence contains tracks, you can modify the contents of the tracks by invoking methods of the `Track` class. The `MidiEvents` contained in the `Track` are stored as a `java.util.Vector` in the `Track` object, and `Track` provides a set of methods for accessing, adding, and removing the events in the list. The methods `add` and `remove` are fairly self-explanatory, adding or removing a specified `MidiEvent` from a `Track`. A `get` method is provided, which takes an index into the `Track's` event list and returns the `MidiEvent` stored there. In addition, there are `size` and `tick` methods, which respectively return the number of `MidiEvents` in the track, and the track's duration, expressed as a total number of `Ticks`.

To create a new event before adding it to the track, you'll of course use the `MidiEvent` constructor. To specify or modify the MIDI message embedded in the event, you can invoke the `setMessage` method of the appropriate `MidiMessage` subclass (`ShortMessage`, `SysexMessage`, or `MetaMessage`). To modify the time that the event should occur, invoke `MidiEvent.setTick`.

In combination, these low-level methods provide the basis for the editing functionality needed by a full-featured sequencer program.

Using Advanced Sequencer Features

So far, this chapter has focused on simple playback and recording of MIDI data. This section will briefly describe some of the more advanced features available through methods of the `Sequencer` interface and the `Sequence` class.

Moving to an Arbitrary Position in the Sequence

There are two `Sequencer` methods that obtain the sequencer's current position in the sequence. The first of these:

```
long getTickPosition()
```

returns the position measured in MIDI ticks from the beginning of the sequence. The second method:

```
long getMicrosecondPosition()
```

returns the current position in microseconds. This method assumes that the sequence is being played at the default rate as stored in the MIDI file or in the `Sequence`. It does *not* return a different value if you've changed the playback speed as described below.

You can similarly set the sequencer's current position according to one unit or the other:

```
void setTickPosition(long tick)
```

or

```
void setMicrosecondPosition(long microsecond)
```

Changing the Playback Speed

As indicated earlier, a sequence's speed is indicated by its tempo, which can vary over the course of the sequence. A sequence can contain events that encapsulate standard MIDI tempo-change messages. When

the sequencer processes such an event, it changes the speed of playback to reflect the indicated tempo. In addition, you can programmatically change the tempo by invoking any of these `Sequencer` methods:

```
public void setTempoInBPM(float bpm)
public void setTempoInMPQ(float mpq)
public void setTempoFactor(float factor)
```

The first two of these methods set the tempo in beats per minute or microseconds per quarter note, respectively. The tempo will stay at the specified value until one of these methods is invoked again, or until a tempo-change event is encountered in the sequence, at which point the current tempo is overridden by the newly specified one.

The third method, `setTempoFactor`, is different in nature. It scales whatever tempo is set for the sequencer (whether by tempo-change events or by one of the first two methods above). The default scalar is 1.0 (no change). Although this method causes the playback or recording to be faster or slower than the nominal tempo (unless the factor is 1.0), it doesn't alter the nominal tempo. In other words, the tempo values returned by `getTempoInBPM` and `getTempoInMPQ` are unaffected by the tempo factor, even though the tempo factor does affect the actual rate of playback or recording. Also, if the tempo is changed by a tempo-change event or by one of the first two methods, it still gets scaled by whatever tempo factor was last set. If you load a new sequence, however, the tempo factor is reset to 1.0.

Note that all these tempo-change directives are ineffectual when the sequence's division type is one of the SMPTE types, instead of PPQ.

Muting or Soloing Individual Tracks in the Sequence

It's often convenient for users of sequencers to be able to turn off certain tracks, to hear more clearly exactly what is happening in the music. A full-featured sequencer program lets the user choose which tracks should sound during playback. (Speaking more precisely, since sequencers don't actually create sound themselves, the user chooses which tracks will contribute to the stream of MIDI messages that the sequencer produces.) Typically, there are two types of graphical controls on each track: a *mute* button and a *solo* button. If the mute button is activated, that track will not sound under any circumstances, until the mute button is deactivated. Soloing is a less well-known feature. It's roughly the opposite of muting. If the solo button on any track is activated, only tracks whose solo buttons are activated will sound. This feature lets the user quickly audition a small number of tracks without having to mute all the other tracks. The mute button typically takes priority over the solo button: if both are activated, the track doesn't sound.

Using `Sequencer` methods, muting or soloing tracks (as well as querying a track's current mute or solo state) is easily accomplished. Let's assume we have obtained the default `Sequencer` and that we've loaded sequence data into it. Muting the fifth track in the sequence would be accomplished as follows:

```
sequencer.setTrackMute(4, true);
boolean muted = sequencer.getTrackMute(4);
if (!muted) {
    return;         // muting failed
}
```

There are a couple of things to note about the above code snippet. First, tracks of a sequence are numbered starting with 0 and ending with the total number of tracks minus 1. Also, the second argument

to `setTrackMute` is a boolean. If it's true, the request is to mute the track; otherwise the request is to unmute the specified track. Lastly, in order to test that the muting took effect, we invoke the `Sequencer getTrackMute` method, passing it the track number we're querying. If it returns `true`, as we'd expect in this case, then the mute request worked. If it returns `false`, then it failed.

Mute requests may fail for various reasons. For example, the track number specified in the `setTrackMute` call might exceed the total number of tracks, or the sequencer might not support muting. By calling `getTrackMute`, we can determine if our request succeeded or failed.

As an aside, the boolean that's returned by `getTrackMute` can, indeed, tell us if a failure occurred, but it can't tell us why it occurred. We could test to see if a failure was caused by passing an invalid track number to the `setTrackMute` method. To do this, we would call the `getTracks` method of `Sequence`, which returns an array containing all of the tracks in the sequence. If the track number specified in the `setTrackMute` call exceeds the length of this array, then we know we specified an invalid track number.

If the mute request succeeded, then in our example, the fifth track will not sound when the sequence is playing, nor will any other tracks that are currently muted.

The method and techniques for soloing a track are very similar to those for muting. To solo a track, invoke the `setTrackSolo` method of `Sequence:`

```
void setTrackSolo(int track, boolean bSolo)
```

As in `setTrackMute`, the first argument specifies the zero-based track number, and the second argument, if `true`, specifies that the track should be in solo mode; otherwise the track should not be soloed.

By default, a track is neither muted nor soloed.

Synchronizing with Other MIDI Devices

`Sequencer` has an inner class called `Sequencer.SyncMode`. A `SyncMode` object represents one of the ways in which a MIDI sequencer's notion of time can be synchronized with a master or slave device. If the sequencer is being synchronized to a master, the sequencer revises its current time in response to certain MIDI messages from the master. If the sequencer has a slave, the sequencer similarly sends MIDI messages to control the slave's timing.

There are three predefined modes that specify possible masters for a sequencer: `INTERNAL_CLOCK`, `MIDI_SYNC`, and `MIDI_TIME_CODE`. The latter two work if the sequencer receives MIDI messages from another device. In these two modes, the sequencer's time gets reset based on system real-time timing clock messages or MIDI time code (MTC) messages, respectively. (See the MIDI specification for more information about these types of message.) These two modes can also be used as slave modes, in which case the sequencer sends the corresponding types of MIDI messages to its receiver. A fourth mode, `NO_SYNC`, is used to indicate that the sequencer should not send timing information to its receivers.

By calling the `setMasterSyncMode` method with a supported `SyncMode` object as the argument, you can specify how the sequencer's timing is controlled. Likewise, the `setSlaveSyncMode` method

determines what timing information the sequencer will send to its receivers. This information controls the timing of devices that use the sequencer as a master timing source.

Specifying Special Event Listeners

Each track of a sequence can contain many different kinds of `MidiEvents`. Such events include Note On and Note Off messages, program changes, control changes, and meta events. The Java Sound API specifies "listener" interfaces for the last two of these event types (control change events and meta events). You can use these interfaces to receive notifications when such events occur during playback of a sequence.

Objects that support the `ControllerEventListener` interface can receive notification when a `Sequencer` processes particular control-change messages. A control-change message is a standard type of MIDI message that represents a change in the value of a MIDI controller, such as a pitch-bend wheel or a data slider. (See the MIDI specification for the complete list of control-change messages.) When such a message is processed during playback of a sequence, the message instructs any device (probably a synthesizer) that's receiving the data from the sequencer to update the value of some parameter. The parameter usually controls some aspect of sound synthesis, such as the pitch of the currently sounding notes if the controller was the pitch-bend wheel. When a sequence is being recorded, the control-change message means that a controller on the external physical device that created the message has been moved, or that such a move has been simulated in software.

Here's how the `ControllerEventListener` interface is used. Let's assume that you've developed a class that implements the `ControllerEventListener` interface, meaning that your class contains the following method:

```
void controlChange(ShortMessage msg)
```

Let's also assume that you've created an instance of your class and assigned it to a variable called `myListener`. If you include the following statements somewhere within your program:

```
int[] controllersOfInterest = { 1, 2, 4 };
sequencer.addControllerEventListener(myListener,
    controllersOfInterest);
```

then your class's `controlChange` method will be invoked every time the sequencer processes a control-change message for MIDI controller numbers 1, 2, or 4. In other words, when the `Sequencer` processes a request to set the value of any of the registered controllers, the `Sequencer` will invoke your class's `controlChange` method. (Note that the assignments of MIDI controller numbers to specific control devices is detailed in the MIDI 1.0 Specification.)

The `controlChange` method is passed a `ShortMessage` containing the controller number affected, and the new value to which the controller was set. You can obtain the controller number using the `ShortMessage.getData1` method, and the new setting of the controller's value using the `ShortMessage.getData2` method.

The other kind of special event listener is defined by the `MetaEventListener` interface. Meta messages, according to the Standard MIDI Files 1.0 specification, are messages that are not present in MIDI wire protocol but that can be embedded in a MIDI file. They are not meaningful to a synthesizer,

but can be interpreted by a sequencer. Meta messages include instructions (such as tempo change commands), lyrics or other text, and other indicators (such as end-of-track).

The `MetaEventListener` mechanism is analogous to `ControllerEventListener`. Implement the `MetaEventListener` interface in any class whose instances need to be notified when a `MetaMessage` is processed by the sequencer. This involves adding the following method to the class:

```
void meta(MetaMessage msg)
```

You register an instance of this class by passing it as the argument to the `Sequencer addMetaEventListener` method:

```
boolean b = sequencer.addMetaEventListener
        (myMetaListener);
```

This is slightly different from the approach taken by the `ControllerEventListener` interface, because you have to register to receive all `MetaMessages,` not just selected ones of interest. If the sequencer encounters a `MetaMessage` in its sequence, it will invoke `myMetaListener.meta`, passing it the `MetaMessage` encountered. The `meta` method can invoke `getType` on its `MetaMessage` argument to obtain an integer from 0 to 127 that indicates the message type, as defined by the Standard MIDI Files 1.0 specification.

Chapter 12: Synthesizing Sound

Most programs that avail themselves of the Java™ Sound API's MIDI package do so to synthesize sound. The entire apparatus of MIDI files, events, sequences, and sequencers, which was discussed in other chapters, nearly always has the goal of eventually sending musical data to a synthesizer to convert into audio. (Possible exceptions include programs that convert MIDI into musical notation that can be read by a musician, and programs that send messages to external MIDI-controlled devices such as mixing consoles.)

The `Synthesizer` interface is therefore fundamental to the MIDI package. This chapter shows how to manipulate a synthesizer to play sound. Many programs will simply use a sequencer to send MIDI file data to the synthesizer, and won't need to invoke many `Synthesizer` methods directly. However, it's possible to control a synthesizer directly, without using sequencers or even `MidiMessage` objects, as explained near the end of this chapter.

The synthesis architecture might seem complex for readers who are unfamiliar with MIDI. Its API includes three interfaces:

- `Synthesizer`
- `MidiChannel`
- `Soundbank`

and four classes:

- `Instrument`
- `Patch`
- `SoundbankResource`
- `VoiceStatus`

As orientation for all this API, the next section explains some of the basics of MIDI synthesis and how they're reflected in the Java Sound API. (Also see the brief section "Synthesizers" under "The Java Sound API's Representation of MIDI Devices" in Chapter 8, "Overview of the MIDI Package.") Subsequent sections give a more detailed look at the API.

Understanding MIDI Synthesis

How does a synthesizer generate sound? Depending on its implementation, it may use one or more sound-synthesis techniques. For example, many synthesizers use wavetable synthesis. A wavetable synthesizer reads stored snippets of audio from memory, playing them at different sample rates and looping them to create notes of different pitches and durations. For example, to synthesize the sound of a saxophone playing the note C#4 (MIDI note number 61), the synthesizer might access a very short

snippet from a recording of a saxophone playing the note Middle C (MIDI note number 60), and then cycle repeatedly through this snippet at a slightly faster sample rate than it was recorded at, which creates a long note of a slightly higher pitch. Other synthesizers use techniques such as frequency modulation (FM), additive synthesis, or physical modeling, which don't make use of stored audio but instead generate audio from scratch using different algorithms.

Instruments

What all synthesis techniques have in common is the ability to create many sorts of sounds. Different algorithms, or different settings of parameters within the same algorithm, create different-sounding results. An *instrument* is a specification for synthesizing a certain type of sound. That sound may emulate a traditional musical instrument, such as a piano or violin; it may emulate some other kind of sound source, for instance, a telephone or helicopter; or it may emulate no "real-world" sound at all. A specification called General MIDI defines a standard list of 128 instruments, but most synthesizers allow other instruments as well. Many synthesizers provide a collection of built-in instruments that are always available for use; some synthesizers also support mechanisms for loading additional instruments.

An instrument may be vendor-specific—in other words, applicable to only one synthesizer or several models from the same vendor. This incompatibility results when two different synthesizers use different sound-synthesis techniques, or different internal algorithms and parameters even if the fundamental technique is the same. Because the details of the synthesis technique are often proprietary, incompatibility is common. The Java Sound API includes ways to detect whether a given synthesizer supports a given instrument.

An instrument can usually be considered a preset; you don't have to know anything about the details of the synthesis technique that produces its sound. However, you can still vary aspects of its sound. Each Note On message specifies the pitch and volume of an individual note. You can also alter the sound through other MIDI commands such as controller messages or system-exclusive messages.

Channels

Many synthesizers are *multimbral* (sometimes called *polytimbral*), meaning that they can play the notes of different instruments simultaneously. (*Timbre* is the characteristic sound quality that enables a listener to distinguish one kind of musical instrument from other kinds.) Multimbral synthesizers can emulate an entire ensemble of real-world instruments, instead of only one instrument at a time. MIDI synthesizers normally implement this feature by taking advantage of the different MIDI channels on which the MIDI specification allows data to be transmitted. In this case, the synthesizer is actually a collection of sound-generating units, each emulating a different instrument and responding independently to messages that are received on a different MIDI channel. Since the MIDI specification provides only 16 channels, a typical MIDI synthesizer can play up to 16 different instruments at once. The synthesizer receives a stream of MIDI commands, many of which are channel commands. (Channel commands are targeted to a particular MIDI channel; for more information, see the MIDI specification.) If the synthesizer is multitimbral, it routes each channel command to the correct sound-generating unit, according to the channel number indicated in the command.

In the Java Sound API, these sound-generating units are instances of classes that implement the `MidiChannel` interface. A `synthesizer` object has at least one `MidiChannel` object. If the

synthesizer is multimbral, it has more than one, normally 16. Each `MidiChannel` represents an independent sound-generating unit.

Because a synthesizer's `MidiChannel` objects are more or less independent, the assignment of instruments to channels doesn't have to be unique. For example, all 16 channels could be playing a piano timbre, as though there were an ensemble of 16 pianos. Any grouping is possible—for instance, channels 1, 5, and 8 could be playing guitar sounds, while channels 2 and 3 play percussion and channel 12 has a bass timbre. The instrument being played on a given MIDI channel can be changed dynamically; this is known as a *program change*.

Even though most synthesizers allow only 16 or fewer instruments to be active at a given time, these instruments can generally be chosen from a much larger selection and assigned to particular channels as required.

Soundbanks and Patches

Instruments are organized hierarchically in a synthesizer. As was mentioned in Chapter 8, "Overview of the MIDI Package," the instruments are arranged by bank number and program number. Banks and programs can be thought of as rows and columns in a two-dimensional table of instruments. A bank is a collection of programs. The MIDI specification allows up to 128 programs in a bank, and up to 128 banks. However, a particular synthesizer might support only one bank, or a few banks, and might support fewer than 128 programs per bank.

In the Java Sound API, there's a higher level to the hierarchy: a soundbank. Soundbanks can contain up to 128 banks, each containing up to 128 instruments. Some synthesizers can load an entire soundbank into memory.

To select an instrument from the current soundbank, you specify a bank number and a program number. The MIDI specification accomplishes this with two MIDI commands: bank select and program change. In the Java Sound API, the combination of a bank number and program number is encapsulated in a `Patch` object. You change a MIDI channel's current instrument by specifying a new patch. The patch can be considered the two-dimensional index of the instruments in the current soundbank.

You might be wondering if soundbanks, too, are indexed numerically. The answer is no; the MIDI specification does not provide for this. In the Java Sound API, a `Soundbank` object can be obtained by reading a soundbank file. If the soundbank is supported by the synthesizer, its instruments can be loaded into the synthesizer individually as desired, or all at once. Many synthesizers have a built-in or default soundbank; the instruments contained in this soundbank are always available to the synthesizer.

Voices

It's important to distinguish between the number of *timbres* a synthesizer can play simultaneously and the number of *notes* it can play simultaneously. The former was described above under "Channels." The ability to play multiple notes at once is referred to as *polyphony*. Even a synthesizer that isn't multitimbral can generally play more than one note at a time (all having the same timbre, but different pitches). For example, playing any chord, such as a G major triad or a B minor seventh chord, requires polyphony. Any synthesizer that generates sound in real time has a limitation on the number of notes it can synthesize at once. In the Java Sound API, the synthesizer reports this limitation through the

`getMaxPolyphony` method.

A *voice* is a succession of single notes, such as a melody that a person can sing. Polyphony consists of multiple voices, such as the parts sung by a choir. A 32-voice synthesizer, for example, can play 32 notes simultaneously. (However, some MIDI literature uses the word "voice" in a different sense, similar to the meaning of "instrument" or "timbre.")

The process of assigning incoming MIDI notes to specific voices is known as *voice allocation*. A synthesizer maintains a list of voices, keeping track of which ones are active (meaning that they currently have notes sounding). When a note stops sounding, the voice becomes inactive, meaning that it's now free to accept the next note-on request that the synthesizer receives. An incoming stream of MIDI commands can easily request more simultaneous notes than the synthesizer is capable of generating. When all the synthesizer's voices are active, how should the next Note On request be handled? Synthesizers can implement different strategies: the most recently requested note can be ignored; or it can be played by discontinuing another note, such as the least recently started one.

Although the MIDI specification does not require it, a synthesizer can make public the contents of each of its voices. The Java Sound API includes a `VoiceStatus` class for this purpose.

A `VoiceStatus` reports on the voice's current active or inactive status, MIDI channel, bank and program number, MIDI note number, and MIDI volume.

With this background, let's examine the specifics of the Java Sound API for synthesis.

Managing Instruments and Soundbanks

In many cases, a program can make use of a `Synthesizer` object without explicitly invoking almost any of the synthesis API. For example, suppose you're playing a standard MIDI file. You load it into a `Sequence` object, which you play by having a sequencer send the data to the default synthesizer. The data in the sequence controls the synthesizer as intended, playing all the right notes at the right times.

However, there are cases when this simple scenario is inadequate. The sequence contains the right music, but the instruments sound all wrong! This unfortunate situation probably arose because the creator of the MIDI file had different instruments in mind than the ones that are currently loaded into the synthesizer.

The MIDI 1.0 Specification provides for bank-select and program-change commands, which affect which instrument is currently playing on each MIDI channel. However, the specification does not define what instrument should reside in each patch location (bank and program number). The more recent General MIDI specification addresses this problem by defining a bank containing 128 programs that correspond to specific instrument sounds. A General MIDI synthesizer uses 128 instruments that match this specified set. Different General MIDI synthesizers can sound quite different, even when playing what's supposed to be the same instrument. However, a MIDI file should for the most part sound similar (even if not identical), no matter which General MIDI synthesizer is playing it.

Nonetheless, not all creators of MIDI files want to be limited to the set of 128 timbres defined by General MIDI. This section shows how to change the instruments from the default set that the synthesizer comes with. (If there is no default, meaning that no instruments are loaded when you access the synthesizer, you'll have to use this API to start with in any case.)

Learning What Instruments Are Loaded

To learn whether the instruments currently loaded into the synthesizer are the ones you want, you can invoke this `Synthesizer` method:

```
Instrument[] getLoadedInstruments()
```

and iterate over the returned array to see exactly which instruments are currently loaded. Most likely, you would display the instruments' names in the user interface (using the `getName` method of `Instrument`), and let the user decide whether to use those instruments or load others. The `Instrument` API includes a method that reports which soundbank the instrument belongs to. The soundbank's name might help your program or the user ascertain exactly what the instrument is.

This `Synthesizer` method:

```
Soundbank getDefaultSoundbank()
```

gives you the default soundbank. The `Soundbank` API includes methods to retrieve the soundbank's name, vendor, and version number, by which the program or the user can verify the bank's identity. However, you can't assume when you first get a synthesizer that the instruments from the default soundbank have been loaded into the synthesizer. For example, a synthesizer might have a large assortment of built-in instruments available for use, but because of its limited memory it might not load them automatically.

Loading Different Instruments

The user might decide to load instruments that are different from the current ones (or you might make that decision programmatically). The following method tells you which instruments come with the synthesizer (versus having to be loaded from soundbank files):

```
Instrument[] getAvailableInstruments()
```

You can load any of these instruments by invoking:

```
boolean loadInstrument(Instrument instrument)
```

The instrument gets loaded into the synthesizer in the location specified by the instrument's `Patch` object (which can be retrieved using the `getPatch` method of `Instrument`).

To load instruments from other soundbanks, first invoke `Synthesizer's isSupportedSoundbank` method to make sure that the soundbank is compatible with this synthesizer (if it isn't, you can iterate over the system's synthesizers to try to find one that supports the soundbank). You can then invoke one of these methods to load instruments from the soundbank:

```
boolean loadAllInstruments(Soundbank soundbank)
boolean loadInstruments(Soundbank soundbank,
  Patch[] patchList)
```

As the names suggest, the first of these loads the entire set of instruments from a given soundbank, and the second loads selected instruments from the soundbank. You could also use `Soundbank's getInstruments` method to access all the instruments, then iterate over them and load selected instruments one at a time using `loadInstrument`.

It's not necessary for all the instruments you load to be from the same soundbank. You could use `loadInstrument` or `loadInstruments` to load certain instruments from one soundbank, another set from a different soundbank, and so on.

Each instrument has its own `Patch` object that specifies the location on the synthesizer where the instrument should be loaded. The location is defined by a bank number and a program number. There's no API to change the location by changing the patch's bank or program number.

However, it is possible to load an instrument into a location other than the one specified by its patch, using the following method of `Synthesizer`:

```
boolean remapInstrument(Instrument from, Instrument to)
```

This method unloads its first argument from the synthesizer, and places its second argument in whatever synthesizer patch location had been occupied by the first argument.

Unloading Instruments

Loading an instrument into a program location automatically unloads whatever instrument was already at that location, if any. You can also explicitly unload instruments without necessarily replacing them with new ones. `Synthesizer` includes three unloading methods that correspond to the three loading methods. If the synthesizer receives a program-change message that selects a program location where no instrument is currently loaded, there won't be any sound from the MIDI channel on which the program-change message was sent.

Accessing Soundbank Resources

Some synthesizers store other information besides instruments in their soundbanks. For example, a wavetable synthesizer stores audio samples that one or more instruments can access. Because the samples might be shared by multiple instruments, they're stored in the soundbank independently of any instrument. Both the `Soundbank` interface and the `Instrument` class provide a method call `getSoundbankResources`, which returns a list of `SoundbankResource` objects. The details of these objects are specific to the synthesizer for which the soundbank is designed. In the case of wavetable synthesis, a resource might be an object that encapsulates a series of audio samples, taken from one snippet of a sound recording. Synthesizers that use other synthesis techniques might store other kinds of objects in the synthesizer's `SoundbankResources` array.

Querying the Synthesizer's Capabilities and Current State

The `Synthesizer` interface includes methods that return information about the synthesizer's capabilities:

```
public long getLatency()
public int getMaxPolyphony()
```

The latency measures the worst-case delay between the time a MIDI message is delivered to the synthesizer and the time that the synthesizer actually produces the corresponding result. For example, it might take a synthesizer a few milliseconds to begin generating audio after receiving a note-on event.

The `getMaxPolyphony` method indicates how many notes the synthesizer can sound simultaneously,

as discussed under "Voices" in the section "Understanding MIDI Synthesis" earlier in this chapter. As mentioned in the same discussion, a synthesizer can provide information about its voices. This is accomplished through the following method:

```
public VoiceStatus[] getVoiceStatus()
```

Each `VoiceStatus` in the returned array reports the voice's current active or inactive status, MIDI channel, bank and program number, MIDI note number, and MIDI volume. The array's length should normally be the same number returned by `getMaxPolyphony`. If the synthesizer isn't playing, all its `VoiceStatus` objects have their active field set to `false`.

You can learn additional information about the current status of a synthesizer by retrieving its `MidiChannel` objects and querying their state. This is discussed more in the next section.

Using Channels

Sometimes it's useful or necessary to access a synthesizer's `MidiChannel` objects directly. This section discusses such situations.

Controlling the Synthesizer without Using a Sequencer

When using a sequence, such as one read from a MIDI file, you don't need to send MIDI commands to the synthesizer yourself. Instead, you just load the sequence into a sequencer, connect the sequencer to the synthesizer, and let it run. The sequencer takes care of scheduling the events, and the result is a predictable musical performance. This scenario works fine when the desired music is known in advance, which is true when it's read from a file.

In some situations, however, the music is generated on the fly as it's playing. For example, the user interface might display a musical keyboard or a guitar fretboard and let the user play notes at will by clicking with the mouse. As another example, an application might use a synthesizer not to play music per se, but to generate sound effects in response to the user's actions. This scenario is typical of games. As a final example, the application might indeed be playing music that's read from a file, but the user interface allows the user to interact with the music, altering it dynamically. In all these cases, the application sends commands directly to the synthesizer, since the MIDI messages need to be delivered immediately, instead of being scheduled for some determinate point in the future.

There are at least two ways of sending a MIDI message to the synthesizer without using a sequencer. The first is to construct a `MidiMessage` and pass it to the synthesizer using the send method of `Receiver`. For example, to produce a Middle C (MIDI note number 60) on MIDI channel 5 (one-based) immediately, you could do the following:

```
ShortMessage myMsg = new ShortMessage();
// Play the note Middle C (60) moderately loud
// (velocity = 93)on channel 4 (zero-based).
myMsg.setMessage(ShortMessage.NOTE_ON, 4, 60, 93);
Synthesizer synth = MidiSystem.getSynthesizer();
Receiver synthRcvr = synth.getReceiver();
synthRcvr.send(myMsg, -1); // -1 means no time stamp
```

The second way is to bypass the message-passing layer (that is, the `MidiMessage` and `Receiver` API) altogether, and interact with the synthesizer's `MidiChannel` objects directly. You first need to retrieve the synthesizer's `MidiChannel` objects, using the following `Synthesizer` method:

```
public MidiChannel[] getChannels()
```

after which you can invoke the desired `MidiChannel` methods directly. This is a more immediate route than sending the corresponding `MidiMessages` to the synthesizer's `Receiver` and letting the synthesizer handle the communication with its own `MidiChannels`. For example, the code corresponding to the preceding example would be:

```
Synthesizer synth = MidiSystem.getSynthesizer();
MidiChannel chan[] = synth.getChannels();
// Check for null; maybe not all 16 channels exist.
if (chan[4] != null) {
     chan[4].noteOn(60, 93);
}
```

Getting a Channel's Current State

The `MidiChannel` interface provides methods that correspond one-to-one to each of the "channel voice" or "channel mode" messages defined by the MIDI specification. We saw one case with the use of the noteOn method in the previous example. However, in addition to these canonical methods, the Java Sound API's `MidiChannel` interface adds some "get" methods to retrieve the value most recently set by corresponding voice or mode "set" methods:

```
int       getChannelPressure()
int       getController(int controller)
boolean   getMono()
boolean   getOmni()
int       getPitchBend()
int       getPolyPressure(int noteNumber)
int       getProgram()
```

These methods might be useful for displaying channel state to the user, or for deciding what values to send subsequently to the channel.

Muting and Soloing a Channel

The Java Sound API adds the notions of per-channel solo and mute, which are not required by the MIDI specification. These are similar to the solo and mute on the tracks of a MIDI sequence. (See "Muting or Soloing Individual Tracks in the Sequence" in Chapter 11, "Playing, Recording, and Editing MIDI Sequences.)"

If mute is on, this channel will not sound, but other channels are unaffected. If solo is on, this channel, and any other soloed channel, will sound (if it isn't muted), but no other channels will sound. A channel that is both soloed and muted will not sound. The `MidiChannel` API includes four methods:

```
boolean      getMute()
boolean      getSolo()
void         setMute(boolean muteState)
void         setSolo(boolean soloState)
```

Permission to Play Synthesized Sound

The audio produced by any installed MIDI synthesizer is typically routed through the sampled-audio system. If your program doesn't have permission to play audio, the synthesizer's sound won't be heard, and a security exception will be thrown. For more information on audio permissions, see "Permission to Use Audio Resources" in Chapter 3, "Accessing Audio System Resources."

Chapter 13: Introduction to the Service Provider Interfaces

What Are Services?

Services are units of sound-handling functionality that are automatically available when an application program makes use of an implementation of the Java™ Sound API. They consist of objects that do the work of reading, writing, mixing, processing, and converting audio and MIDI data. An implementation of the Java Sound API generally supplies a basic set of services, but mechanisms are also included in the API to support the development of new sound services by third-party developers (or by the vendor of the implementation itself). These new services can be "plugged into" an existing installed implementation to expand its functionality without requiring a new release. In the Java Sound API architecture, third-party services are integrated into the system in such a way that an application program's interface to them is the same as the interface to the "built-in" services. In some cases, application developers who use the `javax.sound.sampled` and `javax.sound.midi` packages might not even be aware that they are employing third-party services.

Examples of potential third-party, sampled-audio services include:

- Sound file readers and writers
- Converters that translate between different audio data formats
- New audio mixers and input/output devices, whether implemented purely in software, or in hardware with a software interface

Third-party MIDI services might consist of:

- MIDI file readers and writers
- Readers for various types of soundbank files (which are often specific to particular synthesizers)
- MIDI-controlled sound synthesizers, sequencers, and I/O ports, whether implemented purely in software, or in hardware with a software interface

How Services Work

The `javax.sound.sampled` and `javax.sound.midi` packages provide functionality to application developers who wish to include sound services in their application programs. These packages are for *consumers* of sound services, providing interfaces to get information about, control, and access audio and MIDI services. In addition, the Java Sound API also supplies two packages that define abstract classes to be used by *providers* of sound services: the `javax.sound.sampled.spi` and `javax.sound.midi.spi` packages.

Developers of new sound services implement concrete subclasses of the appropriate classes in the SPI packages. These subclasses, along with any additional classes required to support the new service, are placed in a Java™ Archive (JAR) archive file with a description of the included service or services. When this JAR file is installed in the user's `CLASSPATH`, the runtime system automatically makes the new service available, extending the functionality of the Java™ platform's runtime system.

Once the new service is installed, it can be accessed just like any previously installed service. Consumers of services

can get information about the new service, or obtain instances of the new service class itself, by invoking methods of the `AudioSystem` and `MidiSystem` classes (in the `javax.sound.sampled` and `javax.sound.midi` packages, respectively) to return information about the new services, or to return instances of new or existing service classes themselves. Application programs need not—and should not—reference the classes in the SPI packages (and their subclasses) directly to make use of the installed services.

For example, suppose a hypothetical service provider called Acme Software, Inc. is interested in supplying a package that allows application programs to read a new format of sound file (but one whose audio data is in a standard data format). The SPI class `AudioFileReader` can be subclassed into a class called, say, `AcmeAudioFileReader`. In the new subclass, Acme would supply implementations of all the methods defined in `AudioFileReader`; in this case there are only two methods (with argument variants), `getAudioFileFormat` and `getAudioInputStream`. Then when an application program attempted to read a sound file that happened to be in Acme's file format, it would invoke methods of the `AudioSystem` class in `javax.sound.sampled` to access the file and information about it. The methods `AudioSystem.getAudioInputStream` and `AudioSystem.getAudioFileFormat` provide a standard API to read audio streams; with the `AcmeAudioFileReader` class installed, this interface is extended to support the new file type transparently. Application developers don't need direct access to the newly registered SPI classes: the `AudioSystem` object methods pass the query on to the installed `AcmeAudioFileReader` class.

What's the point of having these "factory" classes? Why not permit the application developer to get access directly to newly provided services? That is a possible approach, but having all management and instantiation of services pass through gatekeeper system objects shields the application developer from having to know anything about the identity of installed services. Application developers just use services of value to them, perhaps without even realizing it. At the same time this architecture permits service providers to effectively manage the available resources in their packages.

Often the use of new sound services is transparent to the application program. For example, imagine a situation where an application developer wants to read in a stream of audio from a file. Assuming that `thePathName` identifies an audio input file, the program does this:

```
File theInFile = new File(thePathName);
AudioInputStream theInStream = AudioSystem.getAudioInputStream(theInFile);
```

Behind the scenes, the `AudioSystem` determines what installed service can read the file and asks it to supply the audio data as an `AudioInputStream` object. The developer might not know or even care that the input audio file is in some new file format (such as Acme's), supported by installed third-party services. The program's first contact with the stream is through the `AudioSystem` object, and all its subsequent access to the stream and its properties are through the methods of `AudioInputStream`. Both of these are standard objects in the `javax.sound.sampled` API; the special handling that the new file format may require is completely hidden.

How Providers Prepare New Services

Service providers supply their new services in specially formatted JAR files, which are to be installed in a directory on the user's system where the Java runtime will find them. JAR files are archive files, each containing sets of files that might be organized in hierarchical directory structures within the archive. Details about the preparation of the class files that go into these archives are discussed in Chapters 14 and 15, which describe the specifics of the audio and MIDI SPI packages; here we'll just give an overview of the process of JAR file creation.

The JAR file for a new service or services should contain a class file for each service supported in the JAR file. Following the Java platform's convention, each class file has the name of the newly defined class, which is a concrete subclass of one of the abstract service provider classes. The JAR file also must include any supporting classes required by the new service implementation. So that the new service or services can be located by the runtime system's service provider mechanism, the JAR file must also contain special files (described below) that map the SPI class names to the new subclasses being defined.

To continue from our example above, say Acme Software, Inc. is distributing a package of new sampled-audio services. Let's suppose this package consists of two new services:

- The `AcmeAudioFileReader` class, which was mentioned above, and which is a subclass of `AudioFileReader`
- A subclass of `AudioFileWriter` called `AcmeAudioFileWriter`, which will write sound files in Acme's new format

Starting from an arbitrary directory—let's call it `/devel`—where we want to do the build, we create subdirectories and put the new class files in them, organized in such a manner as to give the desired pathname by which the new classes will be referenced:

```
com/acme/AcmeAudioFileReader.class
com/acme/AcmeAudioFileWriter.class
```

In addition, for each new SPI class being subclassed, we create a mapping file in a specially named directory `META-INF/services`. The name of the file is the name of the SPI class being subclassed, and the file contains the names of the new subclasses of that SPI abstract class.

We create the file

`META-INF/services/javax.sound.sampled.spi.AudioFileReader`, which consists of

```
# Providers of sound file-reading services
# (a comment line begins with a pound sign)
com.acme.AcmeAudioFileReader
```

and also the file

`META-INF/services/javax.sound.sampled.spi.AudioFileWriter,` which consists of

```
# Providers of sound file-writing services
com.acme.AcmeAudioFileWriter
```

Now we run `jar` from any directory with the command line:

```
jar cvf acme.jar -C /devel .
```

The `-C` option causes `jar` to switch to the `/devel` directory, instead of using the directory in which the command is executed. The final period argument instructs `jar` to archive all the contents of that directory (namely, `/devel`), but not the directory itself.

This run will create the file `acme.jar` with the contents:

```
com/acme/AcmeAudioFileReader.class
com/acme/AcmeAudioFileWriter.class
META-INF/services/javax.sound.sampled.spi.AudioFileReader
META-INF/services/javax.sound.sampled.spi.AudioFileWriter
META-INF/Manifest.mf
```

The file `Manifest.mf,` which is generated by the `jar` utility itself, is a list of all the files contained in the archive.

How Users Install New Services

For end users (or system administrators) who wish to get access to a new service through their application programs, installation is simple. They place the provided JAR file in a directory in their `CLASSPATH.` Upon execution, the Java runtime will find the referenced classes when needed.

It's not an error to install more than one provider for the same service. For example, two different service providers might supply support for reading the same type of sound file. In such a case, the system arbitrarily chooses one of the providers. Users who care which provider is chosen should install only the desired one.

Chapter 14: Providing Sampled-Audio Services

As discussed in Chapter 13, "Introduction to the Service Provider Interfaces," the Java™ Sound API includes two packages, `javax.sound.sampled.spi` and `javax.sound.midi.spi`, that define abstract classes to be used by developers of sound services. By implementing and installing a subclass of one of these abstract classes, a service provider registers the new service, extending the functionality of the runtime system. The present chapter tells you how to go about using the `javax.sound.sampled.spi` package to provide new services for handling sampled audio.

This chapter can be safely skipped by application programmers who merely wish to use existing audio services in their programs. For the use of the installed audio services in an application program, see Part I, "Sampled Audio," of this Programmer's Guide. This chapter assumes that the reader is familiar with the Java™ Sound API methods that application programs invoke to access installed audio services.

Introduction

There are four abstract classes in the `javax.sound.sampled.spi` package, representing four different types of services that you can provide for the sampled-audio system:

- `AudioFileWriter` provides sound file-writing services. These services make it possible for an application program to write a stream of audio data to a file of a particular type.
- `AudioFileReader` provides file-reading services. These services enable an application program to ascertain a sound file's characteristics, and to obtain a stream from which the file's audio data can be read.
- `FormatConversionProvider` provides services for converting audio data formats. These services allow an application program to translate audio streams from one data format to another.
- `MixerProvider` provides management of a particular kind of mixer. This mechanism allows an application program to obtain information about, and access instances of, a given kind of mixer.

To recapitulate the discussion in Chapter 13, service providers can extend the functionality of the runtime system. A typical SPI class has two types of methods: ones that respond to queries about the types of services available from a particular provider, and ones that either perform the new service directly, or return instances of objects that actually provide the service. The runtime environment's service-provider mechanism provides *registration* of installed services with the audio system, and *management* of the new service provider classes.

In essence there is a double isolation of the service instances from the application developer. An application program never directly creates instances of the service objects, such as mixers or format

converters, that it needs for its audio processing tasks. Nor does the program even directly request these objects from the SPI classes that administer them. The application program makes requests to the `AudioSystem` object in the `javax.sound.sampled` package, and `AudioSystem` in turn uses the SPI objects to process these queries and service requests.

The existence of new audio services might be completely transparent to both the user and the application programmer. All application references are through standard objects of the `javax.sound.sampled` package, primarily `AudioSystem`, and the special handling that new services might be providing is often completely hidden.

In this chapter, we'll continue the previous chapter's convention of referring to new SPI subclasses by names like `AcmeMixer` and `AcmeMixerProvider`.

Providing Audio File-Writing Services

Let's start with `AudioFileWriter`, one of the simpler SPI classes.

A subclass that implements the methods of `AudioFileWriter` must provide implementations of a set of methods to handle queries about the file formats and file types supported by the class, as well as provide methods that actually write out a supplied audio data stream to a `File` or `OutputStream`.

`AudioFileWriter` includes two methods that have concrete implementations in the base class:

```
boolean isFileTypeSupported(AudioFileFormat.Type fileType)
boolean isFileTypeSupported(AudioFileFormat.Type fileType,
    AudioInputStream stream)
```

The first of these methods informs the caller whether this file writer can write sound files of the specified type. This method is a general inquiry, it will return `true` if the file writer can write that kind of file, assuming the file writer is handed appropriate audio data. However, the ability to write a file can depend on the format of the specific audio data that's handed to the file writer. A file writer might not support every audio data format, or the constraint might be imposed by the file format itself. (Not all kinds of audio data can be written to all kinds of sound files.) The second method is more specific, then, asking whether a particular `AudioInputStream` can be written to a particular type of file.

Generally, you won't need to override these two concrete methods. Each is simply a wrapper that invokes one of two other query methods and iterates over the results returned. These other two query methods are abstract and therefore need to be implemented in the subclass:

```
abstract AudioFileFormat.Type[] getAudioFileTypes()
abstract AudioFileFormat.Type[]
    getAudioFileTypes(AudioInputStream stream)
```

These methods correspond directly to the previous two. Each returns an array of all the supported file types-all that are supported in general, in the case of the first method, and all that are supported for a specific audio stream, in the case of the second method. A typical implementation of the first method might simply return an array that the file writer's constructor initializes. An implementation of the second method might test the stream's `AudioFormat` object to see whether it's a data format that the requested type of file supports.

The final two methods of `AudioFileWriter` do the actual file-writing work:

```
abstract  int write(AudioInputStream stream,
    AudioFileFormat.Type fileType, java.io.File out)
 abstract  int write(AudioInputStream stream,
    AudioFileFormat.Type fileType, java.io.OutputStream out)
```

These methods write a stream of bytes representing the audio data to the stream or file specified by the third argument. The details of how this is done depend on the structure of the specified type of file. The `write` method must write the file's header and the audio data in the manner prescribed for sound files of this format (whether it's a standard type of sound file or a new, possibly proprietary one).

Providing Audio File-Reading Services

The `AudioFileReader` class consists of six abstract methods that your subclass needs to implement-actually, two different overloaded methods, each of which can take a `File`, `URL`, or `InputStream` argument. The first of these overloaded methods accepts queries about the file format of a specified file:

```
abstract AudioFileFormat getAudioFileFormat(
    java.io.File file)
abstract AudioFileFormat getAudioFileFormat(
    java.io.InputStream stream)
abstract AudioFileFormat getAudioFileFormat(
    java.net.URL url)
```

A typical implementation of `getAudioFileFormat` method reads and parses the sound file's header to ascertain its file format. See the description of the AudioFileFormat class to see what fields need to be read from the header, and refer to the specification for the particular file type to figure out how to parse the header.

Because the caller providing a stream as an argument to this method expects the stream to be unaltered by the method, the file reader should generally start by marking the stream. After reading to the end of the header, it should reset the stream to its original position.

The other overloaded `AudioFileReader` method provides file-reading services, by returning an AudioInputStream from which the file's audio data can be read:

```
abstract AudioInputStream getAudioInputStream(
    java.io.File file)
abstract AudioInputStream getAudioInputStream(
     java.io.InputStream stream)
abstract AudioInputStream getAudioInputStream(
     java.net.URL url)
```

Typically, an implementation of `getAudioInputStream` returns an `AudioInputStream` wound to the beginning of the file's data chunk (after the header), ready for reading. It would be conceivable, though, for a file reader to return an `AudioInputStream` whose audio format represents a stream of data that is in some way decoded from what is contained in the file. The important thing is that the method return a formatted stream from which the audio data contained in the file can be read. The `AudioFormat` encapsulated in the returned `AudioInputStream` object will inform the caller about

the stream's data format, which is usually, but not necessarily, the same as the data format in the file itself.

Generally, the returned stream is an instance of `AudioInputStream`; it's unlikely you would ever need to subclass `AudioInputStream`.

Providing Format-Conversion Services

A `FormatConversionProvider` subclass transforms an `AudioInputStream` that has one audio data format into one that has another format. The former (input) stream is referred to as the *source* stream, and the latter (output) stream is referred to as the *target* stream. Recall from Chapter 2, "Overview of the Sampled Package," that an `AudioInputStream` contains an `AudioFormat`, and the `AudioFormat` in turn contains a particular type of data encoding, represented by an `AudioFormat.Encoding` object. The format and encoding in the source stream are called the source format and source encoding, and those in the target stream are likewise called the target format and target encoding.

The work of conversion is performed in the overloaded abstract method of `FormatConversionProvider` called `getAudioInputStream`. The class also has abstract query methods for learning about all the supported target and source formats and encodings. There are concrete wrapper methods for querying about a specific conversion.

The two variants of `getAudioInputStream` are:

```
abstract AudioInputStream getAudioInputStream(
    AudioFormat.Encoding targetEncoding,
    AudioInputStream sourceStream)
```

and

```
abstract AudioInputStream getAudioInputStream(
    AudioFormat targetFormat,
    AudioInputStream sourceStream)
```

These differ in the first argument, according to whether the caller is specifying a complete target format or just the format's encoding.

A typical implementation of `getAudioInputStream` works by returning a new subclass of `AudioInputStream` that wraps around the original (source) `AudioInputStream` and applies a data format conversion to its data whenever a `read` method is invoked. For example, consider the case of a new `FormatConversionProvider` subclass called `AcmeCodec`, which works with a new `AudioInputStream` subclass called `AcmeCodecStream`.

The implementation of `AcmeCodec's` second `getAudioInputStream` method might be:

```
public AudioInputStream getAudioInputStream
      (AudioFormat outputFormat, AudioInputStream stream) {
        AudioInputStream cs = null;
        AudioFormat inputFormat = stream.getFormat();
```

```
            if (inputFormat.matches(outputFormat) ) {
                cs = stream;
            } else {
                cs = (AudioInputStream)
                    (new AcmeCodecStream(stream, outputFormat));
                tempBuffer = new byte[tempBufferSize];
            }
            return cs;
        }
```

The actual format conversion takes place in new `read` methods of the returned `AcmeCodecStream`, a subclass of `AudioInputStream`. Again, application programs that access this returned `AcmeCodecStream` simply operate on it as an `AudioInputStream`, and don't need to know the details of its implementation.

The other methods of a `FormatConversionProvider` all permit queries about the input and output encodings and formats that the object supports. The following four methods, being abstract, need to be implemented:

```
abstract AudioFormat.Encoding[] getSourceEncodings()
abstract AudioFormat.Encoding[] getTargetEncodings()
abstract AudioFormat.Encoding[] getTargetEncodings(
    AudioFormat sourceFormat)
abstract  AudioFormat[] getTargetFormats(
    AudioFormat.Encoding targetEncoding,
    AudioFormat sourceFormat)
```

As in the query methods of the `AudioFileReader` class discussed above, these queries are typically handled by checking private data of the object and, for the latter two methods, comparing them against the argument(s).

The remaining four `FormatConversionProvider` methods are concrete and generally don't need to be overridden:

```
boolean isConversionSupported(
    AudioFormat.Encoding targetEncoding,
    AudioFormat sourceFormat)
boolean isConversionSupported(AudioFormat targetFormat,
    AudioFormat sourceFormat)
boolean isSourceEncodingSupported(
    AudioFormat.Encoding sourceEncoding)
boolean isTargetEncodingSupported(
    AudioFormat.Encoding targetEncoding)
```

As with `AudioFileWriter.isFileTypeSupported()`, the default implementation of each of these methods is essentially a wrapper that invokes one of the other query methods and iterates over the results returned.

Providing New Types of Mixers

As its name implies, a `MixerProvider` supplies instances of mixers. Each concrete `MixerProvider` subclass acts as a factory for the `Mixer` objects used by an application program. Of course, defining a new `MixerProvider` only makes sense if one or more new implementations of the `Mixer` interface are also defined. As in the `FormatConversionProvider` example above, where our `getAudioInputStream` method returned a subclass of `AudioInputStream` that performed the conversion, our new class `AcmeMixerProvider` has a method `getMixer` that returns an instance of another new class that implements the `Mixer` interface. We'll call the latter class `AcmeMixer`. Particularly if the mixer is implemented in hardware, the provider might support only one static instance of the requested device. If so, it should return this static instance in response to each invocation of `getMixer`.

Since `AcmeMixer` supports the `Mixer` interface, application programs don't require any additional information to access its basic functionality. However, if `AcmeMixer` supports functionality not defined in the `Mixer` interface, and the vendor wants to make this extended functionality accessible to application programs, the mixer should of course be defined as a public class with additional, well-documented public methods, so that a program that wishes to make use of this extended functionality can import `AcmeMixer` and cast the object returned by `getMixer` to this type.

The other two methods of `MixerProvider` are:

```
abstract Mixer.Info[] getMixerInfo()
```

and

```
boolean isMixerSupported(Mixer.Info info)
```

These methods allow the audio system to determine whether this particular provider class can produce a device that an application program needs. In other words, the `AudioSystem` object can iterate over all the installed `MixerProviders` to see which ones, if any, can supply the device that the application program has requested of the `AudioSystem`. (See the discussion under "Getting a Mixer" in Chapter 3, "Accessing Audio System Resources.") The `getMixerInfo` method returns an array of objects containing information about the kinds of mixer available from this provider object. The system can pass these information objects, along with those from other providers, to an application program.

A single `MixerProvider` can provide more than one kind of mixer. When the system invokes the `MixerProvider's getMixerInfo` method, it gets a list of information objects identifying the different kinds of mixer that this provider supports. The system can then invoke `MixerProvider.getMixer(Mixer.Info)` to obtain each mixer of interest.

Your subclass needs to implement `getMixerInfo`, as it's abstract. The `isMixerSupported` method is concrete and doesn't generally need to be overridden. The default implementation simply compares the provided `Mixer.Info` to each one in the array returned by `getMixerInfo`.

Chapter 15: Providing MIDI Services

Chapter 13, "Introduction to the Service Provider Interfaces," explained that the `javax.sound.sampled.spi` and `javax.sound.midi.spi` packages define abstract classes to be used by developers of sound services. By implementing a subclass of one of these abstract classes, a service provider can create a new service that extends the functionality of the runtime system. Chapter 14 covered the use of the `javax.sound.sampled.spi` package. The present chapter discusses how to use the `javax.sound.midi.spi` package to provide new services for handling MIDI devices and files.

Applications programmers who only use existing MIDI services in their programs can safely skip this chapter. For an overview of MIDI and the use of the installed MIDI services in an application program, see Part II, "MIDI," of this Programmer's Guide. This chapter assumes that the reader is familiar with the Java™ Sound API methods that application programs invoke to access installed MIDI services.

Introduction

There are four abstract classes in the `javax.sound.midi.spi` package, which represent four different types of services that you can provide for the MIDI system:

- `MidiFileWriter` provides MIDI file-writing services. These services make it possible for an application program to save, to a MIDI file, a MIDI `Sequence` that it has generated or processed.
- `MidiFileReader` provides file-reading services that return a MIDI `Sequence` from a MIDI file for use in an application program.
- `MidiDeviceProvider` supplies instances of one or more specific types of MIDI device, possibly including hardware devices.
- `SoundbankReader` supplies soundbank file-reading services. Concrete subclasses of `SoundbankReader` parse a given soundbank file, producing a `Soundbank` object that can be loaded into a `Synthesizer`.

An application program will not directly create an instance of a service object—whether a provider object, such as a `MidiDeviceProvider`, or an object, such as a `Synthesizer`, that is supplied by the provider object. Nor will the program directly refer to the SPI classes. Instead, the application program makes requests to the `MidiSystem` object in the `javax.sound.midi` package, and `MidiSystem` in turn uses concrete subclasses of the `javax.sound.midi.spi` classes to process these requests.

Providing MIDI File-Writing Services

There are three standard MIDI file formats, all of which an implementation of the Java Sound API can support: Type 0, Type 1, and Type 2. These file formats differ in their internal representation of the MIDI sequence data in the file, and are appropriate for different kinds of sequences. If an implementation doesn't itself support all three types, a service provider can supply the support for the unimplemented ones. There are also variants of the standard MIDI file formats, some of them proprietary, which similarly could be supported by a third-party vendor.

The ability to write MIDI files is provided by concrete subclasses of `MidiFileWriter`. This abstract class is directly analogous to `javax.sampled.spi.AudioFileWriter`. Again, the methods are grouped into query methods for learning what types of files can be written, and methods for actually writing a file. As with `AudioFileWriter`, two of the query methods are concrete:

```
boolean isFileTypeSupported(int fileType)
boolean isFileTypeSupported(int fileType, Sequence sequence)
```

The first of these provides general information about whether the file writer can ever write the specified type of MIDI file type. The second method is more specific: it asks whether a particular Sequence can be written to the specified type of MIDI file. Generally, you don't need to override either of these two concrete methods. In the default implementation, each invokes one of two other corresponding query methods and iterates over the results returned. Being abstract, these other two query methods need to be implemented in the subclass:

```
abstract int[] getMidiFileTypes()
abstract int[] getMidiFileTypes(Sequence sequence)
```

The first of these returns an array of all the file types that are supported in general. A typical implementation might initialize the array in the file writer's constructor and return the array from this method. From that set of file types, the second method finds the subset to which the file writer can write the given Sequence. In accordance with the MIDI specification, not all types of sequences can be written to all types of MIDI files.

The `write` methods of a `MidiFileWriter` subclass perform the encoding of the data in a given `Sequence` into the correct data format for the requested type of MIDI file, writing the coded stream to either a file or an output stream:

```
abstract int write(Sequence in, int fileType,
                   java.io.File out)
abstract int write(Sequence in, int fileType,
                   java.io.OutputStream out)
```

To do this, the `write` method must parse the `Sequence` by iterating over its tracks, construct an appropriate file header, and write the header and tracks to the output. The MIDI file's header format is, of course, defined by the MIDI specification. It includes such information as a "magic number" identifying this as a MIDI file, the header's length, the number of tracks, and the sequence's timing information (division type and resolution). The rest of the MIDI file consists of the track data, in the format defined by the MIDI specification.

Let's briefly look at how the application program, MIDI system, and service provider cooperate in writing a MIDI file. In a typical situation, an application program has a particular MIDI `Sequence` to

save to a file. The program queries the `MidiSystem` object to see what MIDI file formats, if any, are supported for the particular `Sequence` at hand, before attempting to write the file. The `MidiSystem.getMidiFileTypes(Sequence)` method returns an array of all the MIDI file types to which the system can write a particular sequence. It does this by invoking the corresponding `getMidiFileTypes` method for each of the installed `MidiFileWriter` services, and collecting and returning the results in an array of integers that can be thought of as a master list of all file types compatible with the given `Sequence`. When it comes to writing the `Sequence` to a file, the call to `MidiSystem.write` is passed an integer representing a file type, along with the `Sequence` to be written and the output file; `MidiSystem` uses the supplied type to decide which installed `MidiFileWriter` should handle the write request, and dispatches a corresponding `write` to the appropriate `MidiFileWriter`.

Providing MIDI File-Reading Services

The `MidiFileReader` abstract class is directly analogous to `javax.sampled.spi.AudioFileReader` class. Both consist of two overloaded methods, each of which can take a `File`, `URL`, or `InputStream` argument. The first of the overloaded methods returns the file format of a specified file. In the case of `MidiFileReader`, the API is:

```
abstract MidiFileFormat getMidiFileFormat(java.io.File file)
abstract MidiFileFormat getMidiFileFormat(
    java.io.InputStream stream)
abstract MidiFileFormat getMidiFileFormat(java.net.URL url)
```

Concrete subclasses must implement these methods to return a filled-out `MidiFileFormat` object describing the format of the specified MIDI file (or stream or URL), assuming that the file is of a type supported by the file reader and that it contains valid header information. Otherwise, an `InvalidMidiDataException` should be thrown.

The other overloaded method returns a MIDI `Sequence` from a given file, stream, or URL :

```
abstract Sequence getSequence(java.io.File file)
abstract Sequence getSequence(java.io.InputStream stream)
abstract Sequence getSequence(java.net.URL url)
```

The `getSequence` method performs the actual work of parsing the bytes in the MIDI input file and constructing a corresponding `Sequence` object. This is essentially the inverse of the process used by `MidiFileWriter.write`. Because there is a one-to-one correspondence between the contents of a MIDI file as defined by the MIDI specification and a `Sequence` object as defined by the Java Sound API, the details of the parsing are straightforward. If the file passed to `getSequence` contains data that the file reader can't parse (for example, because the file has been corrupted or doesn't conform to the MIDI specification), an `InvalidMidiDataException` should be thrown.

Providing Particular MIDI Devices

A `MidiDeviceProvider` can be considered a factory that supplies one or more particular types of MIDI device. The class consists of a method that returns an instance of a MIDI device, as well as query methods to learn what kinds of devices this provider can supply.

As with the other javax.sound.midi.spi services, application developers get indirect access to a MidiDeviceProvider service through a call to MidiSystem methods, in this case MidiSystem.getMidiDevice and MidiSystem.getMidiDeviceInfo. The purpose of subclassing MidiDeviceProvider is to supply a new kind of device, so the service developer must also create an accompanying class for the device being returned—just as we saw with MixerProvider in the javax.sound.sampled.spi package. There, the returned device's class implemented the javax.sound.sampled.Mixer interface; here it implements the javax.sound.midi.MidiDevice interface. It might also implement a subinterface of MidiDevice, such as Synthesizer or Sequencer.

Because a single subclass of MidiDeviceProvider can provide more than one type of MidiDevice, the getDeviceInfo method of the class returns an array of MidiDevice.Info objects enumerating the different MidiDevices available:

```
abstract MidiDevice.Info[] getDeviceInfo()
```

The returned array can contain a single element, of course. A typical implementation of the provider might initialize an array in its constructor and return it here. This allows MidiSystem to iterate over all installed MidiDeviceProviders to construct a list of all installed devices. MidiSystem can then return this list (MidiDevice.Info[] array) to an application program.

MidiDeviceProvider also includes a concrete query method:

```
boolean isDeviceSupported(MidiDevice.Info info)
```

This method permits the system to query the provider about a specific kind of device. Generally, you don't need to override this convenience method. The default implementation iterates over the array returned by getDeviceInfo and compares the argument to each element.

The third and final MidiDeviceProvider method returns the requested device:

```
abstract MidiDevice getDevice(MidiDevice.Info info)
```

This method should first test the argument to make sure it describes a device that this provider can supply. If it doesn't, it should throw an IllegalArgumentException. Otherwise, it returns the device.

Providing Soundbank File-Reading Services

A SoundBank is a set of Instruments that can be loaded into a Synthesizer. An Instrument is an implementation of a sound-synthesis algorithm that produces a particular sort of sound, and includes accompanying name and information strings. A SoundBank roughly corresponds to a bank in the MIDI specification, but it's a more extensive and addressable collection; it can perhaps better be thought of as a collection of MIDI banks. (For more background information on SoundBanks and Synthesizers, see Chapter 12, "Synthesizing Sound.")

SoundbankReader consists of a single overloaded method, which the system invokes to read a Soundbank object from a soundbank file:

```
abstract Soundbank getSoundbank(java.io.File file)
abstract Soundbank getSoundbank(java.io.InputStream stream)
```

```
abstract Soundbank getSoundbank(java.net.URL url)
```

Concrete subclasses of `SoundbankReader` will work in tandem with particular provider-defined implementations of `SoundBank`, `Instrument`, and `Synthesizer` to allow the system to load a `SoundBank` from a file into an instance of a particular `Synthesizer` class. Synthesis techniques may differ wildly from one `Synthesizer` to another, and, as a consequence, the data stored in an `Instrument` or `SoundBank` providing control or specification data for the synthesis process of a `Synthesizer` can take a variety of forms. One synthesis technique may require only a few bytes of parameter data; another may be based on extensive sound samples. The resources present in a `SoundBank` will depend upon the nature of the `Synthesizer` into which they get loaded, and therefore the implementation of the `getSoundbank` method of a `SoundbankReader` subclass has access to knowledge of a particular kind of `SoundBank`. In addition, a particular subclass of `SoundbankReader` understands a particular file format for storing the `SoundBank` data. That file format may be vendor-specific and proprietary.

`SoundBank` is just an interface, with only weak constraints on the contents of a `SoundBank` object. The methods an object must support to implement this interface (`getResources`, `getInstruments`, `getVendor`, `getName`, etc.) impose loose requirements on the data that the object contains. For example, `getResources` and `getInstruments` can return empty arrays. The actual contents of a subclassed `SoundBank` object, in particular its instruments and its non-instrument resources, are defined by the service provider. Thus, the mechanism of parsing a soundbank file depends entirely on the specification of that particular kind of soundbank file.

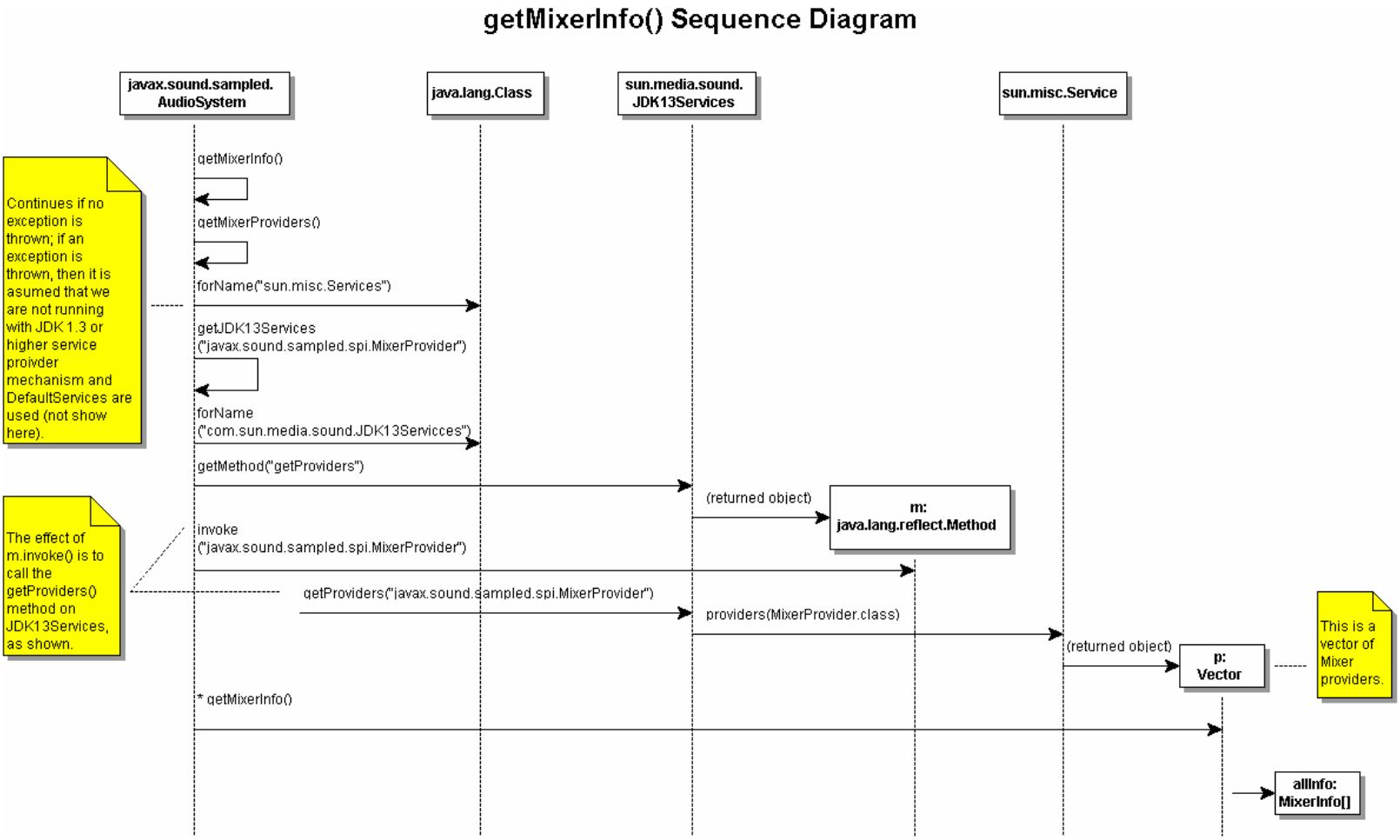
Soundbank files are created outside the Java Sound API, typically by the vendor of the synthesizer that can load that kind of soundbank. Some vendors might supply end-user tools for creating such files.

Appendix 1: Code Overview: `AudioSystem.java`

For the benefit of service providers and API implementers, a brief overview of the JavaSound source code is provided.

`javax.sound.sampled.AudioSystem` is the entry point to JavaSound for obtaining resources; i.e., mixers, lines, etc. And each method of `AudioSystem` involves getting the providers of some service—`MixerProvider[]`, `FormatConversionProvider[]`, `AudioFileReader[]`, or `AudioFileWriter[]`. Then the method goes onto to obtain some specific information or perform some task. There is much similarity in how these methods work. Let us take a look at how `getMixerInfo()` works.

The following is a sequence diagram of `getMixerInfo()` in `AudioSystem.java`:



`getMixerInfo()` in `AudioSystem` first calls `getMixerProviders()`, a `private static` method of `AudioSystem`. The first thing that `getMixerProviders()` does is attempt to load `sun.misc.Service`. If an exception is thrown, it means that a pre-1.3 JRE is in use and there is no service provider lookup mechanism present. If an exception is thrown, then `sun.media.sound.DefaultServices` is used to obtain service providers (not depicted in above diagram). If the 1.3 mechanims is present, then `getJDK13Serives()` is called (shown above) with `"javax.sound.sampled.spi.MixerProvider"` as the argument.

`getJDK13Services()` is another `private static` method of `AudioSystem`. It attempts to the load the class for `"com.sun.media.sound.JDK13Services"`, and if it succeeds it set `Class jdk13Services` equal to it. A bit of trickery is then performed by using `Class.getMethod()` to get the `getProviders()` method of `jdk13Services`, which is returned as object `Method m`. The method is then invoked, which has the effect of invoking `getProviders("javax.sound.sampled.spi.MixerProvider")` on `JDK13Services`. This in turn uses the `providers()` method of `sun.misc.Service` to return a vector of mixer providers, `MixerProvider[]`. The `getMixerInfo()` method of `MixerProvider` is then called on each element of the vector to return info (`Mixer.info`) for all mixers of all mixer providers.

Other services are handled in a similar way. For instance, `getTargetEncodings()` calls `getFormatConversionProviders()`, `getAudioFileFormat()` calls `getAudioFileReaders()`, etc., which are similarly structured to `getMixerProviders()`.