

22

Java Media Framework e Java Sound (no CD)

Objetivos

- Entender para que serve o Java Media Framework (JMF).
- Entender para que serve a API Java Sound.
- Ser capaz de reproduzir mídia de áudio e vídeo com JMF.
- Ser capaz de transmitir fluxos de mídia através de uma rede.
- Ser capaz de capturar, formatar e salvar mídia.
- Ser capaz de reproduzir sons com a API Java Sound.
- Ser capaz de reproduzir, gravar e sintetizar MIDI com a API Java Sound.

A tevê fornece a todos uma imagem, mas o rádio dá luz a um milhão de imagens em um milhão de mentes.
Peggy Noonan

O barulho não prova nada. A galinha que bota um ovo grita como se tivesse botado um asteróide.
Mark Twain, *Following the Equator*

A tela grande só faz o filme ficar duas vezes pior.
Samuel Goldwyn

A vida não é uma série de imagens que mudam à medida que se repetem.
Andy Warhol



Sumário do capítulo

- 22.1** Introdução
- 22.2** Reproduzindo mídia
- 22.3** Formatando e salvando mídia capturada
- 22.4** *Streaming* com RTP
- 22.5** Java Sound
- 22.6** Reproduzindo amostras de áudio
- 22.7** Musical Instrument Digital Interface (MIDI)
 - 22.7.1** Reprodução de MIDI
 - 22.7.2** Gravação de MIDI
 - 22.7.3** Síntese de MIDI
 - 22.7.4** A classe `MidiDemo`
- 22.8** Recursos na Internet e na World Wide Web
- 22.9** (Estudo de caso opcional) Pensando em objetos: animação e som na visão

Resumo • Terminologia • Exercícios de auto-revisão • Respostas aos exercícios de auto-revisão • Exercícios • Seção especial: construindo seu próprio compilador

22.1 Introdução

Este capítulo dá continuidade às nossas discussões sobre multimídia do Capítulo 18, apresentando algumas das APIs de Java para multimídia que permitem aos programadores melhorar os aplicativos com recursos de vídeo e áudio. Em anos recentes, o segmento de multimídia digital do setor de computadores experimentou um crescimento enorme, como demonstra a enorme quantidade de conteúdo de multimídia disponível na Internet. Os *sites* da Web foram transformados de páginas HTML baseadas em texto em experiências intensas em multimídia. Os avanços nas tecnologias de *hardware* e de *software* permitiram que os desenvolvedores integrassem multimídia aos aplicativos mais simples. Na faixa superior dos aplicativos de multimídia, o setor de vídeo games usou a programação com multimídia para tirar proveito das mais recentes tecnologias de *hardware*, como placas de vídeo 3D que criam experiências de realidade virtual para os usuários.

Reconhecendo que os aplicativos Java deveriam suportar recursos de vídeo e áudio digitais, a Sun Microsystems, a Intel e a Silicon Graphics trabalharam juntas para produzir uma API para multimídia que é conhecida como a *Java Multimedia Framework* (JMF). A API JMF é uma das várias APIs para multimídia em Java. Usando a API JMF, os programadores podem criar aplicativos Java que reproduzem, editam e capturam muitos tipos populares de mídia. A primeira metade deste capítulo discute a API JMF.

A IBM e a Sun desenvolveram a mais recente especificação de JMF – versão 2.0. A Sun oferece uma implementação de referência – JMF 2.1.1 – da especificação de JMF que suporta tipos de arquivo de mídia como arquivos *Microsoft Audio/Video Interleave* (.avi), *Macromedia Flash 2 movies* (.swf), *Future Splash* (.spl), *MPEG Layer 3 Audio* (.mp3), *Musical Instrument Digital Interface* (MIDI; .mid), vídeos *MPEG-1* (.mpeg, .mpg), *QuickTime* (.mov), *Sun Audio* (.au), *áudio Wave* (.wav), *AIFF* (.aiff) e *GSM* (.gsm).

Além dos exemplos de clipes de mídia fornecidos com os exemplos deste capítulo no CD, muitos *sites* oferecem um suprimento grande de clipes de áudio e vídeo que podem ser baixados gratuitamente. Você pode baixar clipes de mídia destes *sites* (e muitos outros na Internet) e usá-los para testar os exemplos deste capítulo. Apresentamos uma lista de *sites* aqui para que você possa começar. O *Free Audio Clips* (www.freeaudioclips.com) é um excelente *site* para diversos tipos de arquivos de áudio. O *site 13 Even* (www.13-even.com.media.html) fornece clipes de áudio e vídeo em muitos formatos para seu uso pessoal. Se você estiver procurando arquivos de áudio MIDI para uso na Seção 22.7, dê uma olhada nos clipes MIDI gratuitos no endereço www.freestuffgalore.commidi.asp. O *Funny Video Clips* (www.video-clips.co.uk) oferece material de entretenimen-

to. O *site de downloads* da Microsoft (msdn.microsoft.com/downloads) contém uma seção de multimídia que fornece clipes de áudio e outros tipos de mídia.

Atualmente, o JMF está disponível como pacote de extensão separado do Java 2 Software Development Kit. O CD que acompanha este livro contém o JMF 2.1.1. A implementação mais recente de JMF pode ser baixada do *site* oficial de JMF:

java.sun.com/products/java-media/jmf

O *site* da Web para o JMF fornece versões de JMF que tiram proveito das características de desempenho da plataforma na qual o JMF está sendo executado. Por exemplo, o *Windows Performance Pack* de JMF fornece amplo suporte a mídias e dispositivos para programas Java que estão executados em plataformas Microsoft Windows (Windows 95/98/NT 4.0/2000). O *site* oficial da Web para o JMF também fornece suporte, informações e recursos atualizados continuamente para os programadores de JMF.



Dica de portabilidade 22.1

Escrever programas com o Windows Performance Pack de JMF reduz a portabilidade daqueles programas para outros sistemas operacionais.

O restante deste capítulo discute a *API Java Sound* e seus extensos recursos para processamento de som. Internamente, o JMF usa Java Sound para suas funções de áudio. Nas Seções 22.5 a 22.7, demonstraremos a reprodução de amostras de áudio e as funcionalidades para MIDI com o Java Sound, uma extensão-padrão do Java 2 Software Development Kit.

22.2 Reproduzindo mídia

O JMF é normalmente usado para reproduzir clipes de mídia em aplicativos Java. Muitos aplicativos, como os gerenciadores financeiros, as enciclopédias e os jogos, usam multimídia para ilustrar os recursos do aplicativo, apresentar conteúdo educacional e divertir os usuários.

O JMF oferece diversos mecanismos para reproduzir mídia, e o mais simples deles é através de objetos que implementam a interface **Player**. A interface **Player** (pacote **javax.media**) estende **Controller**, que é um tratador para mídia suportada por JMF.

As seguintes etapas são necessárias para reproduzir um clipe de mídia:

1. Especificar a fonte da mídia;
2. Criar um **Player** para a mídia;
3. Obter a mídia de saída e os controles de **Player**;
4. Exibir a mídia e os controles.

A classe **SimplePlayer** (Fig. 22.1) é um programa simples em Java que reproduz mídia que demonstra diversos recursos comuns de reprodutores de mídia populares. A demonstração **SimplePlayer** pode reproduzir a maioria dos arquivos de mídia suportados por JMF, possivelmente com exceção das versões mais recentes dos formatos. Este aplicativo permite acessar arquivos no computador local que contêm tipos de mídia suportados, clicando-se no botão **Open File**. Clicar no botão **Open Location** e especificar um URL de mídia permite acessar mídia de uma fonte de mídia, como um *dispositivo de captura*, um servidor da Web ou uma *fonte de streaming*. Um dispositivo de captura (discutido na Seção 22.3) lê mídia a partir de dispositivos de áudio e vídeo como microfones, reprodutores de CD e câmeras. O *stream Real-Time Transport Protocol* (RTP) é uma *stream* de bytes enviados através de uma rede a partir de um servidor de *streaming*. O aplicativo guarda no *buffer* e reproduz a mídia de *streaming* no computador cliente.

```
1 // Fig. 22.1: SimplePlayer.java
2 // Abre e reproduz um arquivo de mídia a partir de um
3 // computador local, um URL público ou uma sessão RTP
4
```

Fig. 22.1 Reproduzindo mídia com a interface **Player** (parte 1 de 8).

```

5  // Pacotes do núcleo de Java
6  import java.awt.*;
7  import java.awt.event.*;
8  import java.io.*;
9  import java.net.*;
10
11 // Pacotes de extensão de Java
12 import javax.swing.*;
13 import javax.media.*;
14
15 public class SimplePlayer extends JFrame {
16
17     // reprodutor de mídia em Java
18     private Player player;
19
20     // componente para conteúdo visual
21     private Component visualMedia;
22
23     // componentes de controle para a mídia
24     private Component mediaControl;
25
26     // contêiner principal
27     private Container container;
28
29     // endereços do arquivo de mídia e da mídia
30     private File mediaFile;
31     private URL fileURL;
32
33     // construtor para SimplePlayer
34     public SimplePlayer()
35     {
36         super( "Simple Java Media Player" );
37
38         container = getContentPane();
39
40         // painel que contém botões
41         JPanel buttonPanel = new JPanel();
42         container.add( buttonPanel, BorderLayout.NORTH );
43
44         // abrindo um arquivo a partir do botão de diretório
45         JButton openFile = new JButton( "Open File" );
46         buttonPanel.add( openFile );
47
48         // registra um ActionListener para eventos de openFile
49         openFile.addActionListener(
50
51             // classe interna anônima para tratar eventos de openFile
52             new ActionListener() {
53
54                 // abre e cria "player" para o arquivo
55                 public void actionPerformed( ActionEvent event )
56                 {
57                     mediaFile = getFile();
58
59                     if ( mediaFile != null ) {
60
61                         // obtém URL a partir do arquivo
62                         try {
63                             fileURL = mediaFile.toURL();
64

```

Fig. 22.1 Reproduzindo mídia com a interface Player (parte 2 de 8).

```

65
66         // caminho para o arquivo não pode ser encontrado
67         catch ( MalformedURLException badURL ) {
68             badURL.printStackTrace();
69             showErrorMessage( "Bad URL" );
70         }
71
72         makePlayer( fileURL.toString() );
73
74     }
75
76     } // fim do método actionPerformed
77
78     } // fim do método ActionListener
79
80 ); // fim da chamada para o método addActionListener
81
82 // botão de abertura do URL
83 JButton openURL = new JButton( "Open Locator" );
84 buttonPanel.add( openURL );
85
86 // registra um ActionListener para eventos de openURL
87 openURL.addActionListener(
88
89     // classe interna anônima para tratar de eventos de openURL
90     new ActionListener() {
91
92         // abre e cria um "player" para o localizador de mídia
93         public void actionPerformed((ActionEvent event) )
94         {
95             String addressName = getMediaLocation();
96
97             if ( addressName != null )
98                 makePlayer( addressName );
99         }
100
101     } // fim do método ActionListener
102
103 ); // fim da chamada para o método addActionListener
104
105 // liga a geração leve nos players para permitir
106 // melhor compatibilidade com componentes GUI de peso leve
107 Manager.setHint( Manager.LIGHTWEIGHT_RENDERER,
108     Boolean.TRUE );
109
110 } // fim do construtor SimplePlayer
111
112 // método utilitário para mensagens de erro "pop-up"
113 public void showErrorMessage( String error )
114 {
115     JOptionPane.showMessageDialog( this, error, "Error",
116     JOptionPane.ERROR_MESSAGE );
117 }
118
119 // obtém arquivo do computador
120 public File getFile()
121 {
122     JFileChooser fileChooser = new JFileChooser();
123
124     fileChooser.setFileSelectionMode(

```

Fig. 22.1 Reproduzindo mídia com a interface Player (parte 3 de 8).

```

125         JFileChooser.FILES_ONLY );
126
127         int result = fileChooser.showOpenDialog( this );
128
129         if ( result == JFileChooser.CANCEL_OPTION )
130             return null;
131
132         else
133             return fileChooser.getSelectedFile();
134     }
135
136     // obtém endereço da mídia digitado pelo usuário
137     public String getMediaLocation()
138     {
139         String input = JOptionPane.showInputDialog(
140             this, "Enter URL" );
141
142         // se o usuário pressionar OK sem digitar dados
143         if ( input != null && input.length() == 0 )
144             return null;
145
146         return input;
147     }
148
149     // cria player com o endereço da mídia
150     public void makePlayer( String mediaLocation )
151     {
152         // restaura o player e a janela se houver player anterior
153         if ( player != null )
154             removePlayerComponents();
155
156         // endereço da origem da mídia
157         MediaLocator mediaLocator =
158             new MediaLocator( mediaLocation );
159
160         if ( mediaLocator == null ) {
161             showErrorMessage( "Error opening file" );
162             return;
163         }
164
165         // cria um player a partir de MediaLocator
166         try {
167             player = Manager.createPlayer( mediaLocator );
168
169             // registra ControllerListener para tratar de eventos do Player
170             player.addControllerListener(
171                 new PlayerEventHandler() );
172
173             // chama realize para permitir a geração da mídia do player
174             player.realize();
175         }
176
177         // não existe nenhum player ou o formato não é suportado
178         catch ( NoPlayerException noPlayerException ) {
179             noPlayerException.printStackTrace();
180         }
181
182         // erro na leitura do arquivo
183         catch ( IOException ioException ) {
184             ioException.printStackTrace();

```

Fig. 22.1 Reproduzindo mídia com a interface Player (parte 4 de 8).

```

185     }
186
187 } // fim do método makePlayer
188
189 // devolve o player para os recursos do sistema
190 // e restaura a mídia e os controles
191 public void removePlayerComponents()
192 {
193     // remove componente de vídeo anterior, se existe um
194     if ( visualMedia != null )
195         container.remove( visualMedia );
196
197     // remove controle de mídia anterior, se existe um
198     if ( mediaControl != null )
199         container.remove( mediaControl );
200
201     // faz parar o player e devolve os recursos alocados
202     player.close();
203 }
204
205 // obtém controles visuais para mídia e player
206 public void getMediaComponents()
207 {
208     // obtém componente visual do player
209     visualMedia = player.getVisualComponent();
210
211     // adiciona componente visual, se estiver presente
212     if ( visualMedia != null )
213         container.add( visualMedia, BorderLayout.CENTER );
214
215     // obtém a GUI de controle do player
216     mediaControl = player.getControlPanelComponent();
217
218     // adiciona componente de controles, se estiver presente
219     if ( mediaControl != null )
220         container.add( mediaControl, BorderLayout.SOUTH );
221
222 } // fim do método getMediaComponents
223
224 // tratador para os eventos ControllerEvents do player
225 private class PlayerEventHandler extends ControllerAdapter {
226
227     // carrega antecipadamente a mídia assim que o player é realizado
228     public void realizeComplete(
229         RealizeCompleteEvent realizeDoneEvent )
230     {
231         player.prefetch();
232     }
233
234     // player pode começar a mostrar a mídia após a carga antecipada
235     public void prefetchComplete(
236         PrefetchCompleteEvent prefetchDoneEvent )
237     {
238         getMediaComponents();
239
240         // assegura um leiaute válido para a frame
241         validate();
242
243         // começa a reproduzir a mídia

```

Fig. 22.1 Reproduzindo mídia com a interface `Player` (parte 5 de 8).

```

244         player.start();
245     } // fim do método prefetchComplete
246
247     // se fim da mídia, restaura para o início e pára de reproduzir
248     public void endOfMedia( EndOfMediaEvent mediaEndEvent )
249     {
250         player.setMediaTime( new Time( 0 ) );
251         player.stop();
252     }
253
254 } // fim da classe interna PlayerEventHandler
255
256 // executa o aplicativo
257 public static void main( String args[] )
258 {
259     SimplePlayer testPlayer = new SimplePlayer();
260
261     testPlayer.setSize( 300, 300 );
262     testPlayer.setLocation( 300, 300 );
263     testPlayer.setDefaultCloseOperation( EXIT_ON_CLOSE );
264     testPlayer.setVisible( true );
265 }
266
267 } // fim da classe SimplePlayer

```

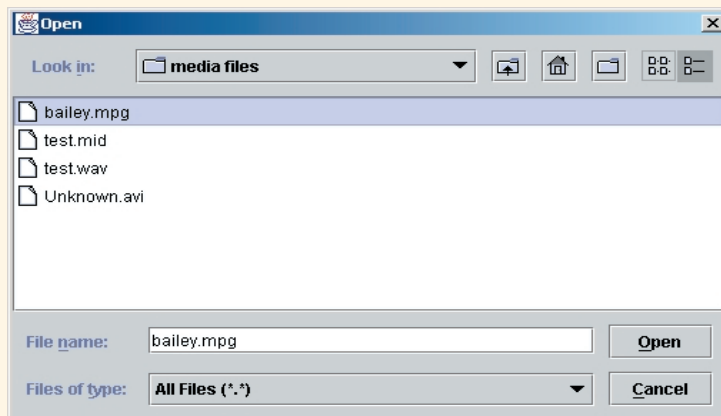
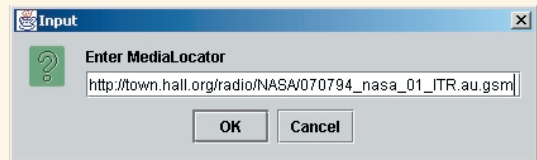
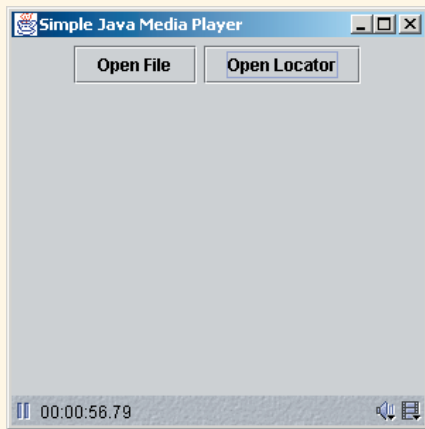


Fig. 22.1 Reproduzindo mídia com a interface `Player` (parte 6 de 8).

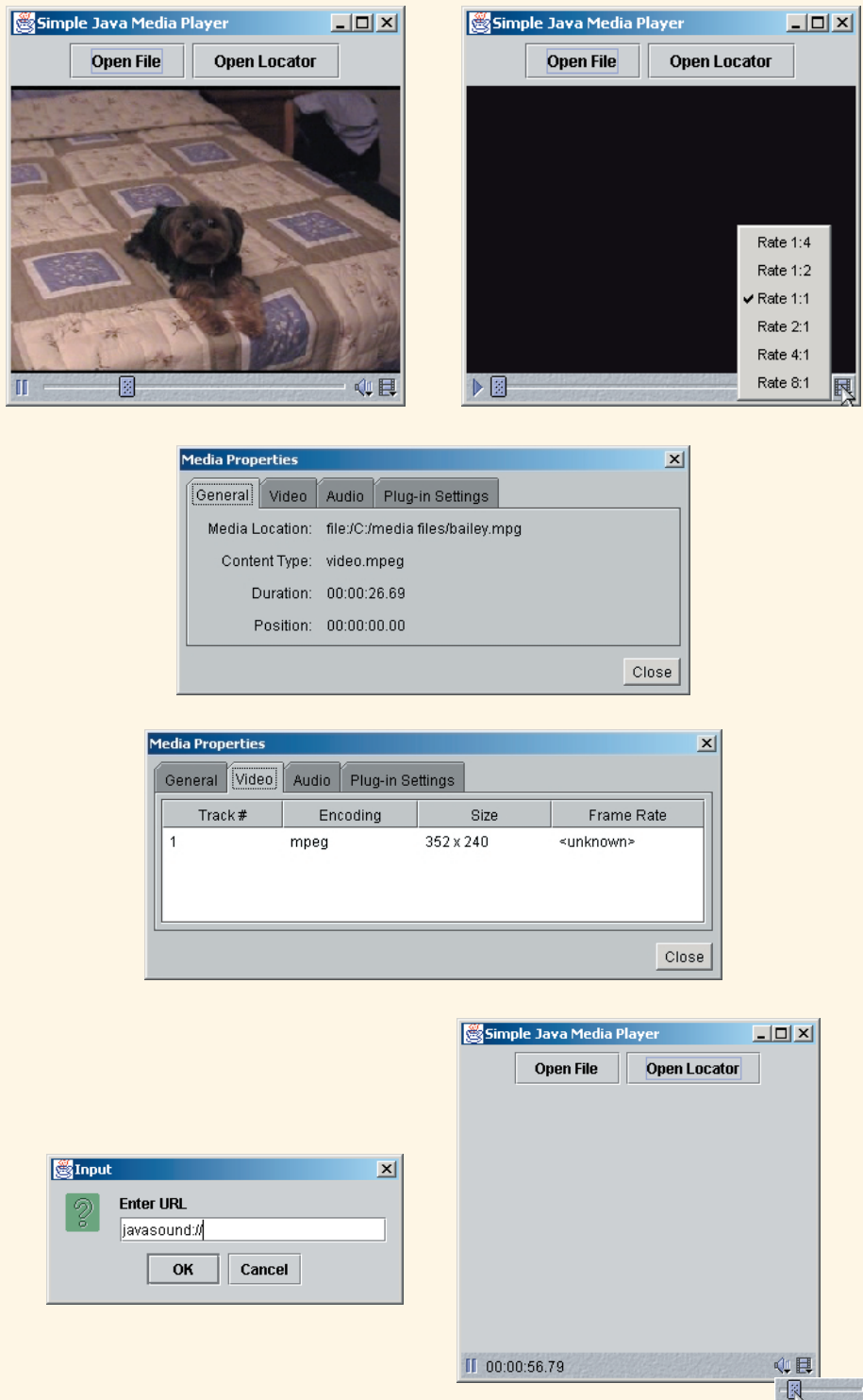


Fig. 22.1 Reproduzindo mídia com a interface `Player` (parte 7 de 8).

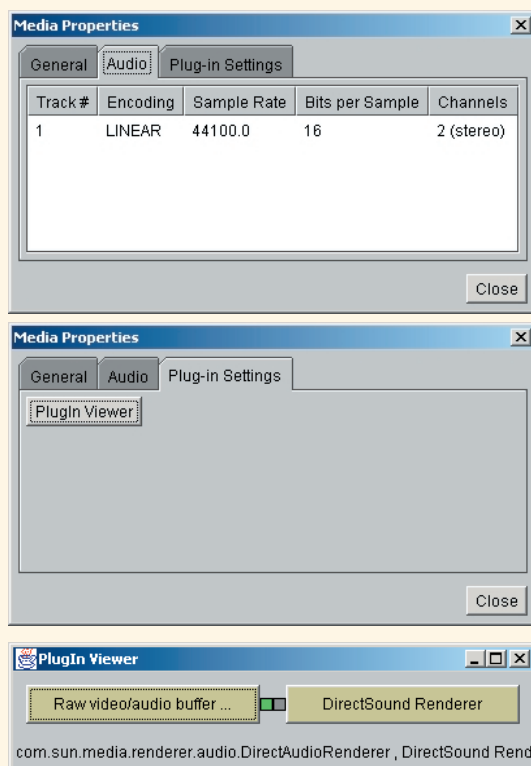


Fig. 22.1 Reproduzindo mídia com a interface **Player** (parte 8 de 8).

O clipe de mídia precisa ser processado antes de ser reproduzido. Para processar um clipe de mídia, o programa precisa acessar uma fonte de mídia, criar um **Controller** para aquela fonte e enviar a mídia para a saída. Antes de enviar para a saída, os usuários podem fazer formatação opcional, como mudar um vídeo AVI para um vídeo QuickTime. Embora o JMF oculte do programador o processamento de mídia de baixo nível (por exemplo, verificar se o arquivo é compatível), programadores e usuários podem configurar como um **Player** apresenta a mídia. A Seção 22.3 e a Seção 22.4 revelam que capturar e fazer *streaming* de mídia seguem as mesmas diretrizes. A Seção 22.8 lista diversos *sites* da Web que têm conteúdo de mídia suportado por JMF.

A Fig. 22.1 apresenta alguns objetos fundamentais para reproduzir mídia. O pacote de extensão de JMF **javax.media** – importado na linha 13 – contém a interface **Player** e outras classes e interfaces necessárias para eventos. A linha 18 declara um objeto **Player** para reproduzir cliques de mídia. As linhas 30 a 31 declaram os endereços destes cliques como referências **File** e **URL**.

As linhas 21 e 24 declaram objetos **Component** para a exibição do vídeo e para manter os controles. O **Component mediaControl** permite reproduzir, fazer pausa e parar o clipe de mídia. O **Component visualMedia** exibe a parte de vídeo de um clipe de mídia (se o clipe de mídia for um vídeo). O JMF fornece geradores de vídeo peso-leve que são compatíveis com os componentes peso-leve do Swing (ver Capítulo 13). As linhas 107 e 108 no construtor de **SimplePlayer** especificam que o **Player** deve desenhar seus componentes GUI e sua parte de vídeo (se existir uma) usando geradores peso-leve, de modo que o reprodutor de mídia terá aparência semelhante à de outros componentes GUI com componentes Swing.

Antes de reproduzir a mídia, **SimplePlayer** exibe uma GUI inicial que consiste em dois botões, **Open File** e **Open Locator**, que permitem especificar o endereço da mídia. Os tratadores de eventos para estes dois botões (linhas 52 a 78 e linhas 90 a 101) executam funções semelhantes. Cada botão pede aos usuários que informem um recurso de mídia, como um clipe de áudio ou de vídeo, depois cria um **Player** para a mídia especificada. Quando

o usuário clica em **Open File**, a linha 57 chama o método **getFile** (linhas 120 a 134) para pedir aos usuários que selecionem um arquivo de mídia do computador local. A linha 63 chama o método **toURL** de **File** para obter uma representação como **URL** do nome e endereço do arquivo selecionado. A linha 72 chama o método **makePlayer** de **SimplePlayer** (linhas 150 a 187) para criar um **Player** para a mídia selecionada pelo usuário. Quando os usuários clicam em **Open Locator**, a linha 95 invoca o método **getMediaLocation** (linhas 137 a 147), pedindo aos usuários que digitem um *string* que especifica o endereço da mídia. A linha 98 chama o método **makePlayer** de **SimplePlayer** para criar um **Player** para a mídia que está no endereço especificado.

O método **makePlayer** (linhas 150 a 187) faz os preparativos necessários para criar um **Player** de cliques de mídia. O argumento **String** indica o endereço da mídia. As linhas 153 e 154 invocam o método **removePlayerComponents** (linhas 191 a 203) para remover da *frame* o componente visual e os controles da GUI do **Player** anterior, antes de criar um novo **Player**. A linha 202 invoca o método **close** de **Player** para parar toda a atividade do **Player** e liberar recursos do sistema mantidos pelo **Player** anterior.

O método **makePlayer** exige um ponteiro para a fonte da qual a mídia é recuperada, o que é feito instanciando-se um novo **MediaLocator** para o valor dado pelo argumento **String** (linhas 157 e 158). O **MediaLocator** especifica o endereço de uma fonte de mídia, de forma muito parecida como um **URL** geralmente especifica o endereço de uma página da Web. O **MediaLocator** pode acessar mídia a partir de dispositivos de captura e sessões RTP e a partir de endereços de arquivos. O construtor de **MediaLocator** exige o endereço da mídia como um **String**, de modo que todos os **URLs** devem ser convertidos para **Strings** como na linha 72.

O método **makePlayer** instancia um novo **Player** com uma chamada ao método **createPlayer** de **Manager**. A classe **Manager** fornece métodos **static** que permitem acessar a maioria dos recursos do JMF. O método **createPlayer** abre a fonte de mídia especificada e determina o **Player** apropriado para a fonte de mídia. O método **createPlayer** dispara uma **NoPlayerException** se não puder ser encontrado um **Player** apropriado para o clipe de mídia. Se ocorrerem problemas na conexão com a fonte de mídia, uma **IOException** é disparada.

Os **ControllerListeners** esperam os **ControllerEvents** que os **Players** geram, para monitorar o progresso de um **Player** no processo de tratamento da mídia. As linhas 170 e 171 registram uma instância da classe interna **PlayerEventHandler** (linhas 225 a 255) para esperar certos eventos que **player** gera. A classe **PlayerEventHandler** estende a classe **ControllerAdapter**, que fornece implementações vazias dos métodos da interface **ControllerListener**. A classe **ControllerAdapter** facilita a implementação de **ControllerListener** para as classes que precisam tratar de apenas alguns tipos de **ControllerEvent**.

Os **Players** confirmam seu progresso durante o processamento da mídia com base em suas transições de estado. A linha 174 invoca o método **realize** de **Player** no estado *Realizing* para indicar que ele está se conectando à sua fonte de mídia e interagindo com ela. Quando um **Player** completa a realização, ele gera um **RealizeCompleteEvent** – um tipo de **ControllerEvent** que ocorre quando o **Player** completa sua transição para o estado *Realized*. Este estado indica que o **Player** completou todos os preparativos necessários para começar a processar a mídia. O programa invoca o método **realizeComplete** (linhas 228 a 232) quando o **Player** gera um **RealizeCompleteEvent**.

A maioria dos reprodutores de mídia têm um recurso de uso de *buffer*, que armazena localmente um pedaço da mídia baixada, de modo que os usuários não precisem esperar que um clipe inteiro seja baixado antes de reproduzi-lo, já que a leitura de dados da mídia pode levar muito tempo. Invocando o método **prefetch** de **Player**, a linha 231 faz a transição de **player** para o estado *Prefetching*. Quando o **Player** carrega antecipadamente um clipe de mídia, o **Player** obtém controle exclusivo sobre certos recursos do sistema necessários para reproduzir o clipe. O **Player** também começa a colocar dados da mídia no *buffer* para reduzir o atraso antes da reprodução do clipe de mídia.

Quando o **Player** completa a carga antecipada, ele passa para o estado *Prefetched* e está pronto para reproduzir a mídia. Durante esta transição, o **Player** gera um **ControllerEvent** do tipo **PrefetchCompleteEvent**, para indicar que está pronto para exibir a mídia. O **Player** invoca o método **prefetchComplete** de **PlayerEventHandler** (linhas 235 a 246), que exibe a GUI do **Player** na *frame*. Após obter os recursos de *hardware*, o programa pode obter os componentes de mídia que ele exige. A linha 238 invoca o método **getMediaComponents** (linhas 206 a 222) para obter os controles da GUI e o componente visual da mídia (se a mídia for um clipe de vídeo) e os anexa ao painel de conteúdo da janela do aplicativo. O método **getVisualComponent** de **Player** (linha 209) obtém o componente visual do clipe de vídeo. Similarmente, a linha 216 invoca o método **getControlPanelComponent** para devolver os controles da GUI. A GUI (Fig. 22.1) normalmente fornece os seguintes controles:

1. Um controle deslizante de posicionamento, para pular para certos pontos no clipe de mídia;
2. Um botão de pausa;
3. Um botão de volume que fornece os controles de volume quando se clicar nele com o botão direito do mouse e uma função de emudecer quando se clicar nele com o botão esquerdo;
4. Um botão de propriedades da mídia que fornece informações detalhadas sobre a mídia quando se clicar nele com o botão direito do mouse e a taxa de exibição de quadros quando se clicar nele com o botão esquerdo.



Observação de aparência e comportamento 22.1

Invocar o método `getVisualComponent` de **Player** devolve `null` para arquivos de áudio, porque não há componente visual a exibir.



Observação de aparência e comportamento 22.2

Invocar o método `getControlPanelComponent` de **Player** devolve conjuntos diferentes de controles de GUI, dependendo do tipo de mídia. Por exemplo, o conteúdo de mídia com stream diretamente de uma conferência ao vivo não tem uma barra de andamento porque o comprimento da mídia não é determinado.

Depois de validar o novo leiaute da *frame* (linha 241), a linha 244 invoca o método **start** de **Player** (linha 239) para começar a reproduzir o clipe de mídia.



Observação de engenharia de software 22.1

Se o **Player** não fez a carga antecipada nem realizou a mídia, invocar o método **start** faz a carga antecipada e realiza a mídia.



Dica de desempenho 22.1

Iniciar o **Player** consome menos tempo se o **Player** já tiver carregado a mídia antecipadamente antes de **start** ser chamado.

Quando o clipe de mídia terminar, o **Player** gera um **ControllerEvent** do tipo **EndOfMediaEvent**. A maioria dos reprodutores de mídia “reenrolam” o clipe de mídia depois de chegar ao fim, de modo que os usuários possam ver ou ouvir novamente a partir do início. O método **endOfMedia** (linhas 249 a 253) trata o **EndOfMediaEvent** e restaura o clipe de mídia para sua posição inicial invocando o método **setMediaTime** de **Player** com um novo **Time** (pacote **javax.media**) de 0 (linha 251). O método **setMediaTime** ajusta a posição da mídia para uma posição específica de tempo e é útil para “pular” para uma parte diferente da mídia. A linha 252 invoca o método **stop** de **Player**, que termina o processamento da mídia e coloca o **Player** no estado *Stopped*. Invocar o método **start** para um **Player Stopped** que não tenha sido fechado retoma a reprodução da mídia.

Freqüentemente, é desejável configurar a mídia antes da apresentação. Na próxima seção, discutimos a interface **Processor**, que tem mais capacidades de configuração do que a interface **Player.Processor** permite que um programa formate a mídia e a salve em um arquivo.

22.3 Formatando e salvando mídia capturada

O *Java Media Framework* consegue reproduzir e salvar a mídia a partir de dispositivos de captura como microfones e câmeras de vídeo. Este tipo de mídia é conhecido como *mídia capturada*. Os dispositivos de captura convertem mídia analógica em mídia digitalizada. Por exemplo, o programa que captura uma voz analógica a partir de um microfone ligado ao computador pode criar um arquivo digitalizado a partir daquela gravação.

O JMF pode acessar dispositivos de captura de vídeo que usam *drivers Video for Windows*. Além disso, o JMF suporta dispositivos de captura de áudio que usam a *Direct Sound Interface* do Windows ou a *Java Sound Interface*. O *driver Video for Windows* fornece interfaces que permitem aos aplicativos Windows acessar e processar mídia a partir de dispositivos de vídeo. Similarmente, *Direct Sound* e *Java Sound* são interfaces que permitem acessar dispositivos de som como placas de som em *hardware*. O *Solaris Performance Pack* fornece suporte para dispositivos de captura *Java Sound* e *Sun Video* na plataforma *Solaris*. Para obter uma lista completa dos dispositivos suportados pelo JMF, visite o site oficial do JMF na Web.

O aplicativo **SimplePlayer** apresentado na Fig. 22.1 permitiu aos usuários reproduzir mídia de um dispositivo de captura. Um *string* *localizador* especifica a localização do dispositivo de captura que a demonstração **SimplePlayer** acessa. Por exemplo, para testar as capacidades de captura de **SimplePlayer**, conecte um microfone ao *plug* de entrada de microfone de uma placa de som. Digitar o *string* localizador `javasound://` no diálogo de entrada **Open Location** especifica que a entrada da mídia deve ser feita a partir do dispositivo de captura habilitado para Java Sound. O *string* localizador inicializa o **MediaLocator** de que o **Player** necessita para o material de áudio capturado do microfone.

Embora **SimplePlayer** forneça acesso a dispositivos de captura, ele não formata a mídia nem salva os dados capturados. A Fig. 22.2 apresenta um programa que executa estas duas novas tarefas. A classe **CapturePlayer** fornece mais controle sobre as propriedades da mídia através da classe **DataSource** (pacote `javax.media.protocol`). A classe **DataSource** fornece a conexão à fonte de mídia, depois abstrai esta conexão para permitir que os usuários a manipulem. Este programa usa uma **DataSource** para formatar a mídia de entrada e de saída. A **DataSource** passa a mídia de saída formatada para um **Controller**, que irá formatá-la adicionalmente para que ela possa ser salva em um arquivo. O **Controller** que manipula a mídia é um **Processor**, que estende a interface **Player**. Finalmente, o objeto que implementa a interface **DataSink** salva a mídia capturada e formatada. O objeto **Processor** trata do fluxo de dados da **DataSource** para o objeto **DataSink**.

```

1  // Fig. 22.2: CapturePlayer.java
2  // Apresenta e salva a mídia capturada.
3
4  // Pacotes do núcleo de Java
5  import java.awt.*;
6  import java.awt.event.*;
7  import java.io.*;
8  import java.util.*;
9
10 // Pacotes de extensão de Java
11 import javax.swing.*;
12 import javax.swing.event.*;
13 import javax.media.*;
14 import javax.media.protocol.*;
15 import javax.media.format.*;
16 import javax.media.control.*;
17 import javax.media.datasink.*;
18
19 public class CapturePlayer extends JFrame {
20
21     // botões de captura e salvamento
22     private JButton captureButton;
23
24     // componente GUI para salvar os dados capturados
25     private Component saveProgress;
26
27     // formatos da mídia do dispositivo, formato escolhido pelo usuário
28     private Format formats[], selectedFormat;
29
30     // controles dos formatos de mídia do dispositivo
31     private FormatControl formatControls[];
32
33     // informações de especificação do dispositivo
34     private CaptureDeviceInfo deviceInfo;
35
36     // Vector contendo todas as informações do dispositivo
37     private Vector deviceList;

```

Fig. 22.2 Formatando e salvando mídia a partir de dispositivos de captura (parte 1 de 9).

```

38
39 // fontes de dados de entrada e saída
40 private DataSource inSource, outSource;
41
42 // gravador do arquivo para a mídia capturada
43 private DataSink dataSink;
44
45 // Processor para gerar e salvar a mídia capturada
46 private Processor processor;
47
48 // construtor para CapturePlayer
49 public CapturePlayer()
50 {
51     super( "Capture Player" );
52
53     // painel contendo botões
54     JPanel buttonPanel = new JPanel();
55     getContentPane().add( buttonPanel );
56
57     // botão para acessar e inicializar dispositivos de captura
58     captureButton = new JButton( "Capture and Save File" );
59     buttonPanel.add( captureButton, BorderLayout.CENTER );
60
61     // registra um ActionListener para eventos do captureButton
62     captureButton.addActionListener( new CaptureHandler() );
63
64     // ativa a geração leve para tornar
65     // compatível com componentes GUI peso leve
66     Manager.setHint( Manager.LIGHTWEIGHT_RENDERER,
67         Boolean.TRUE );
68
69     // registra um WindowListener para eventos da frame
70     addWindowListener(
71
72         // classe interna anônima para tratar WindowEvents
73         new WindowAdapter() {
74
75             // descarta o Processor
76             public void windowClosing(
77                 WindowEvent windowEvent )
78             {
79                 if ( processor != null )
80                     processor.close();
81             }
82
83         } // fim de WindowAdapter
84
85     ); // fim da chamada para o método addWindowListener
86
87 } // fim do construtor
88
89 // classe tratadora de ações para configurar dispositivo
90 private class CaptureHandler implements ActionListener {
91
92     // inicializa e configura dispositivo de captura
93     public void actionPerformed( ActionEvent actionEvent )
94     {
95         // coloca as informações disponíveis do dispositivo no Vector
96         deviceList =
97             CaptureDeviceManager.getDeviceList( null );

```

Fig. 22.2 Formatando e salvando mídia a partir de dispositivos de captura (parte 2 de 9).

```

98
99     // se nenhum dispositivo for encontrado, exibe mensagem de erro
100    if ( ( deviceList == null ) ||
101        ( deviceList.size() == 0 ) ) {
102
103        showMessage( "No capture devices found!" );
104
105        return;
106    }
107
108    // array de nomes de dispositivos
109    String deviceNames[] = new String[ deviceList.size() ];
110
111    // armazena todos os nomes de dispositivos em um
112    // array de strings, para fins de exibição
113    for ( int i = 0; i < deviceList.size(); i++ ){
114
115        deviceInfo =
116            ( CaptureDeviceInfo ) deviceList.elementAt( i );
117
118        deviceNames[ i ] = deviceInfo.getName();
119    }
120
121    // obtém o índice no Vector do dispositivo selecionado
122    int selectDeviceIndex =
123        getSelectedDeviceIndex( deviceNames );
124
125    if ( selectDeviceIndex == -1 )
126        return;
127
128    // obtém as informações de dispositivo do dispositivo selecionado
129    deviceInfo = ( CaptureDeviceInfo )
130        deviceList.elementAt( selectDeviceIndex );
131
132    formats = deviceInfo.getFormats();
133
134    // se o dispositivo de captura anterior estiver aberto, desconecta-o
135    if ( inSource != null )
136        inSource.disconnect();
137
138    // obtém dispositivo e configura seu formato
139    try {
140
141        // cria fonte de dados a partir do MediaLocator do dispositivo
142        inSource = Manager.createDataSource(
143            deviceInfo.getLocator() );
144
145        // obtém controles de configuração de formato para o dispositivo
146        formatControls = ( ( CaptureDevice )
147            inSource ).getFormatControls();
148
149        // obtém a configuração de formato do dispositivo desejada pelo usuário
150        selectedFormat = getSelectedFormat( formats );
151
152        if ( selectedFormat == null )
153            return;
154
155        setDeviceFormat( selectedFormat );

```

Fig. 22.2 Formatando e salvando mídia a partir de dispositivos de captura (parte 3 de 9).

```

156         captureSaveFile();
157     }
158     // fim do try
159 }
160 // não consegue encontrar DataSource a partir do MediaLocator
161 catch ( NoDataSourceException noDataException ) {
162     noDataException.printStackTrace();
163 }
164 // erro de conexão ao dispositivo
165 catch ( IOException ioException ) {
166     ioException.printStackTrace();
167 }
168 // fim do método actionPerformed
169 }
170 // fim da classe interna CaptureHandler
171
172 // configura o formato de saída da mídia capturada pelo dispositivo
173 public void setDeviceFormat( Format currentFormat )
174 {
175     // configura formato desejado em todos os controles de formato
176     for ( int i = 0; i < formatControls.length; i++ ) {
177         // assegura que o controle de formato pode ser configurado
178         if ( formatControls[ i ].isEnabled() ) {
179             formatControls[ i ].setFormat( currentFormat );
180
181             System.out.println (
182                 "Presentation output format currently set as " +
183                 formatControls[ i ].getFormat() );
184         }
185     } // fim do laço for
186 }
187
188 // obtém o índice no Vector do dispositivo selecionado
189 public int getSelectedDeviceIndex( String[] names )
190 {
191     // obtém o nome do dispositivo da caixa de diálogo de seleção de dispositivo
192     String name = ( String ) JOptionPane.showInputDialog(
193         this, "Select a device:", "Device Selection",
194         JOptionPane.QUESTION_MESSAGE,
195         null, names, names[ 0 ] );
196
197     // se o formato foi selecionado, obtém índice do nome no array names
198     if ( name != null )
199         return Arrays.binarySearch( names, name );
200
201     // senão, devolve valor indicando seleção inválida
202     else
203         return -1;
204 }
205
206 // devolve formato selecionado pelo usuário para o dispositivo
207 public Format getSelectedFormat( Format[] showFormats )
208 {

```

Fig. 22.2 Formatando e salvando mídia a partir de dispositivos de captura (parte 4 de 9).

```

215         return ( Format ) JOptionPane.showInputDialog( this,
216         "Select a format: ", "Format Selection",
217         JOptionPane.QUESTION_MESSAGE,
218         null, showFormats, null );
219     }
220
221     // exibe mensagens de erro
222     public void showErrorMessage( String error )
223     {
224         JOptionPane.showMessageDialog( this, error, "Error",
225         JOptionPane.ERROR_MESSAGE );
226     }
227
228     // obtém arquivo desejado para salvar a mídia capturada
229     public File getSaveFile()
230     {
231         JFileChooser fileChooser = new JFileChooser();
232
233         fileChooser.setFileSelectionMode(
234             JFileChooser.FILES_ONLY );
235         int result = fileChooser.showSaveDialog( this );
236
237         if ( result == JFileChooser.CANCEL_OPTION )
238             return null;
239
240         else
241             return fileChooser.getSelectedFile();
242     }
243
244     // mostra o monitor de salvamento dos dados capturados
245     public void showSaveMonitor()
246     {
247         // mostra o diálogo do monitor de salvamento
248         int result = JOptionPane.showConfirmDialog( this,
249             saveProgress, "Save capture in progress...",
250             JOptionPane.DEFAULT_OPTION,
251             JOptionPane.INFORMATION_MESSAGE );
252
253         // termina salvamento se o usuário pressiona "OK" ou fecha diálogo
254         if ( ( result == JOptionPane.OK_OPTION ) ||
255             ( result == JOptionPane.CLOSED_OPTION ) ) {
256
257             processor.stop();
258             processor.close();
259
260             System.out.println ( "Capture closed." );
261         }
262     }
263
264     // processa mídia capturada e salva no arquivo
265     public void captureSaveFile()
266     {
267         // array de formatos desejados de salvamento suportados pelas trilhas
268         Format outFormats[] = new Format[ 1 ];
269
270         outFormats[ 0 ] = selectedFormat;
271
272         // formato de saída no arquivo
273         FileTypeDescriptor outFileType =
274             new FileTypeDescriptor( FileTypeDescriptor.QUICKTIME );

```

Fig. 22.2 Formatando e salvando mídia a partir de dispositivos de captura (parte 5 de 9).

```

275
276 // configura e inicia processador e monitora a captura
277 try {
278
279     // cria processador a partir do modelo de processador
280     // da fonte de dados específica, formatos de saída nas trilhas
281     // e formato de saída no arquivo
282     processor = Manager.createRealizedProcessor(
283         new ProcessorModel( inSource, outFormats,
284             outFileType ) );
285
286     // tenta criar um gravador de dados para a saída de mídia
287     if ( !makeDataWriter() )
288         return;
289
290     // chama start do processador para iniciar alimentação dos dados capturados
291     processor.start();
292
293     // obtém controle de monitor para captura e codificação
294     MonitorControl monitorControl =
295         ( MonitorControl ) processor.getControl(
296             "javax.media.control.MonitorControl" );
297
298     // obtém componente GUI do controle de monitoração
299     saveProgress = monitorControl.getControlComponent();
300
301     showSaveMonitor();
302
303 } // fim do try
304
305 // nenhum processador pode ser encontrado
306 // para uma fonte de dados específica
307 catch ( NoProcessorException processorException ) {
308     processorException.printStackTrace();
309 }
310
311 // incapaz de realizar através do
312 // método createRealizedProcessor
313 catch ( CannotRealizeException realizeException ) {
314     realizeException.printStackTrace();
315 }
316
317 // erro de conexão ao dispositivo
318 catch ( IOException ioException ) {
319     ioException.printStackTrace();
320 }
321
322 } // fim do método captureSaveFile
323
324 // método que inicializa o gravador do arquivo de mídia
325 public boolean makeDataWriter()
326 {
327     File saveFile = getSaveFile();
328
329     if ( saveFile == null )
330         return false;
331
332     // obtém fonte de dados de saída a partir do processador
333     outSource = processor.getDataOutput();

```

Fig. 22.2 Formatando e salvando mídia a partir de dispositivos de captura (parte 6 de 9).

```

334
335     if ( outSource == null ) {
336         showErrorMessage( "No output from processor!" );
337         return false;
338     }
339
340     // inicia o processo de gravação de dados
341     try {
342
343         // cria um novo MediaLocator a partir do URL saveFile
344         MediaLocator saveLocator =
345             new MediaLocator ( saveFile.toURL() );
346
347         // cria DataSink a partir da fonte de dados de saída e do
348         // arquivo de destino de salvamento especificado pelo usuário
349         dataSink = Manager.createDataSink(
350             outSource, saveLocator );
351
352         // registra um DataSinkListener para DataSinkEvents
353         dataSink.addDataSinkListener(
354
355             // classe interna anônima para tratar DataSinkEvents
356             new DataSinkListener () {
357
358                 // se fim da mídia, fecha o gravador de dados
359                 public void dataSinkUpdate(
360                     DataSinkEvent dataEvent )
361                 {
362                     // se a captura foi parada, fecha DataSink
363                     if ( dataEvent instanceof EndOfStreamEvent )
364                         dataSink.close();
365                 }
366
367                 } // fim de DataSinkListener
368
369             ); // fim da chamada ao método addDataSinkListener
370
371         // começa a salvar
372         dataSink.open();
373         dataSink.start();
374
375     } // fim do try
376
377     // DataSink não pode ser encontrada para o arquivo de
378     // salvamento e a fonte de dados específicos
379     catch ( NoDataSinkException noDataSinkException ) {
380         noDataSinkException.printStackTrace();
381         return false;
382     }
383
384     // violação enquanto acessava
385     // destino de MediaLocator
386     catch ( SecurityException securityException ) {
387         securityException.printStackTrace();
388         return false;
389     }
390
391     // problema na abertura e inicialização de DataSink
392     catch ( IOException ioException ) {
393         ioException.printStackTrace();

```

Fig. 22.2 Formatando e salvando mídia a partir de dispositivos de captura (parte 7 de 9).

```

394     return false;
395 }
396
397     return true;
398
399 } // fim do método makeDataWriter
400
401 // método main
402 public static void main( String args[] )
403 {
404     CapturePlayer testPlayer = new CapturePlayer();
405
406     testPlayer.setSize( 200, 70 );
407     testPlayer.setLocation( 300, 300 );
408     testPlayer.setDefaultCloseOperation( EXIT_ON_CLOSE );
409     testPlayer.setVisible( true );
410 }
411
412 } // fim da classe CapturePlayer

```

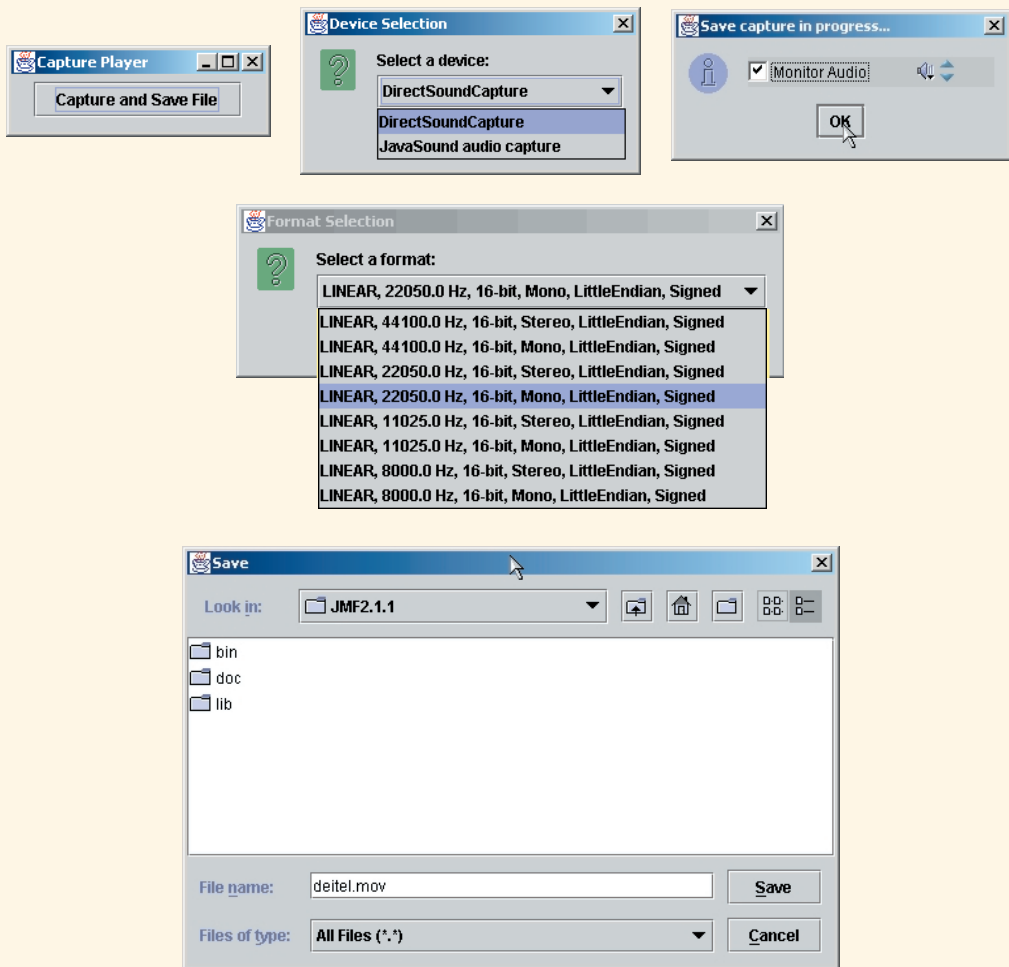


Fig. 22.2 Formatando e salvando mídia a partir de dispositivos de captura (parte 8 de 9).

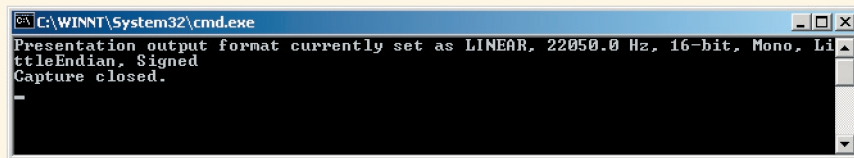


Fig. 22.2 Formatando e salvando mídia a partir de dispositivos de captura (parte 9 de 9).

JMF e Java Sound usam fontes de mídia extensivamente, de modo que os programadores entender a disposição dos dados na mídia. O *cabeçalho* em uma fonte de dados especifica o formato da mídia e outras informações essenciais necessárias para reproduzir a mídia. O conteúdo da mídia normalmente consiste em *trilhas* de dados, semelhantes às trilhas de música em um CD. As fontes de mídia podem ter uma ou mais trilhas que contêm diversos dados. Por exemplo, um clipe de cinema pode conter uma trilha para vídeo, uma trilha para áudio e uma terceira trilha para *closed captioning*, para os deficientes auditivos.

A classe **CapturePlayer** (Fig. 22.2) ilustra a captura, a configuração de formatos de mídia e o salvamento de mídia a partir de dispositivos de captura suportados pelo JMF. O teste mais simples do programa usa um microfone para entrada de áudio. Inicialmente, a GUI tem apenas um botão, no qual o usuário clica para iniciar o processo de configuração. Depois, o usuário seleciona um dispositivo de captura em um diálogo de menu escamoteável. A caixa de diálogo seguinte tem opções para o formato do dispositivo de captura e da saída no arquivo. A terceira caixa de diálogo pede aos usuários para salvar a mídia em um arquivo específico. A caixa de diálogo final oferece um controle de volume e a opção de monitorar os dados. A monitoração permite que os usuários escutem ou vejam a mídia à medida que é capturada e salvado, sem modificá-la de nenhuma maneira. Muitas tecnologias de captura de mídia oferecem recursos de monitoração. Por exemplo, muitos gravadores de vídeo têm uma tela acoplada para deixar os usuários ver o que a câmera está capturando sem olhar através do visor. Em um estúdio de gravações, os produtores podem escutar de uma outra sala a música ao vivo que está sendo gravada, através de fones de ouvido. Monitorar dados é diferente de reproduzir dados e não provoca nenhuma alteração no formato da mídia ou afeta os dados que estão sendo enviados para o **Processor**.

No programa **CapturePlayer**, as linhas 14 a 16 importam os pacotes de extensão de Java **javax.media.protocol**, **javax.media.format** e **javax.media.control**, que contêm classes e interfaces para controle de mídia e formatação de dispositivos. A linha 17 importa o pacote de JMF **javax.media.datasink**, que contêm classes para enviar mídia formatada para a saída. O programa usa as classes e interfaces fornecidas por estes pacotes para obter as informações sobre o dispositivo de captura desejado, configurar o formato do dispositivo de captura, criar um **Processor** para tratar os dados de mídia capturados, criar um **DataSink** para gravar os dados de mídia em um arquivo e monitorar o processo de salvamento.

CapturePlayer pode configurar o formato da mídia. O JMF fornece a classe **Format** para descrever os atributos de um formato de mídia, como a *taxa de amostragem* (que controla a qualidade do som) ou se a mídia deve ser em formato *estéreo* ou *mono*. Cada formato de mídia é codificado de maneira diferente e pode ser reproduzido somente com um tratador de mídia que suporte seu formato particular. A linha 28 declara um **array** com os **Formats** que o dispositivo de captura suporta e uma referência **Format** para o formato selecionado pelo usuário (**selectedFormat**).

Depois de obter os objetos **Format**, o programa precisa ter acesso aos controles de formatação do dispositivo de captura. A linha 31 declara um **array** para guardar os **FormatControls** que irão configurar o formato do dispositivo de captura. A classe **CapturePlayer** configura o **Format** desejado para a fonte de mídia através dos **FormatControls** do dispositivo (linha 31). A referência **CaptureDeviceInfo deviceInfo** (linha 34) armazena as informações do dispositivo de captura, que serão colocadas em um **Vector** que contém todas as informações do dispositivo. A classe **DataSource** conecta os programas a fontes de dados de mídia, como dispositivos de captura. O **SimplePlayer** da Fig. 22.1 acessou um objeto **DataSource** invocando o método **createPlayer** de **Manager**, passando-lhe um **MediaLocator**. Entretanto, a classe **CapturePlayer** acessa a **DataSource** diretamente. A linha 40 declara duas **DataSources** – **inSource** conecta com a mídia do dispositivo de captura e **outSource** conecta à fonte de dados de saída na qual a mídia capturada será salva.

O objeto que implementa a interface **Processor** fornece a função básica que controla e processa o fluxo de dados de mídia na classe **CapturePlayer** (linha 46). A classe também cria um objeto que implementa a interface **DataSink** para gravar os dados capturados em um arquivo (linha 43).

Clicar no botão **Capture and Save File** configura o dispositivo de captura invocando o método **actionPerformed** (linhas 93 a 171) na classe interna **private CaptureHandler** (linhas 90 a 173). Uma instância da classe interna **CaptureHandler** é registrada para esperar **ActionEvents** de **captureButton** (linha 62). O programa fornece aos usuários uma lista dos dispositivos de captura disponíveis quando as linhas 96 e 97 invocam o método **static getDeviceList** de **CaptureDeviceManager**. O método **getDeviceList** obtém todos os dispositivos disponíveis no computador que suportam o **Format** especificado. Especificar **null** como parâmetro **Format** devolve uma lista completa de dispositivos disponíveis. A classe **CaptureDeviceManager** permite que um programa acesse esta lista de dispositivos. As linhas 109 a 119 copiam os nomes de todos os dispositivos de captura para um **array** de **Strings** (**deviceNames**) para fins de exibição. As linhas 122 e 123 invocam o método **getSelectedDeviceIndex** de **CapturePlayer** (linhas 195 a 210) para mostrar um diálogo de seleção com uma lista de todos os nomes de dispositivos armazenados no **array deviceNames**. A chamada para o método **showInputDialog** (linhas 198 a 201) tem uma lista de parâmetros diferente da de exemplos anteriores. Os quatro primeiros parâmetros são o componente-pai do diálogo, a mensagem, o título e o tipo de mensagem, como usado em capítulos anteriores. Os últimos três, que são novos, especificam o ícone (neste caso, **null**), a lista de valores apresentada para o usuário (**deviceNames**) e a seleção **default** (o primeiro elemento de **deviceNames**). Quando o usuário seleciona um dispositivo, o diálogo devolve o **string**, que é usado para devolver o índice em **deviceNames** do nome selecionado. Este elemento, que é uma instância de **CaptureDeviceInfo**, cria e configura um novo dispositivo a partir do qual a mídia desejada pode ser gravada. O objeto **CaptureDeviceInfo** encapsula as informações de que o programa precisa para acessar e configurar um dispositivo de captura, como preferências de localização e formato. Chamar os métodos **getLocator** (linha 143) e **getFormats** (linha 132) acessa estas porções de informações. As linhas 129 e 130 acessam o novo **CaptureDeviceInfo** que o usuário especificou na **deviceList**. A seguir, as linhas 135 e 136 chamam o método **disconnect** de **inSource** para desconectar quaisquer dispositivos de captura abertos anteriormente antes de conectar o novo dispositivo.

As linhas 142 e 143 invocam o método **createDataSource** de **Manager** para obter o objeto **DataSource** que se conecta à fonte de mídia do dispositivo de captura, passando-lhe o objeto **MediaLocator** do dispositivo de captura como argumento. O método **getLocator** de **CaptureDeviceInfo** devolve o **MediaLocator** do dispositivo de captura. O método **createDataSource**, por sua vez, invoca o método **connect** de **DataSource**, que estabelece uma conexão com o dispositivo de captura. O método **createDataSource** dispara uma **NoDataSourceException** se não pode localizar uma **DataSource** para o dispositivo de captura. Se houver um erro durante a abertura do dispositivo, ocorre uma **IOException**.

Antes de capturar a mídia, o programa precisa formatar a **DataSource** como especificado pelo usuário no diálogo de seleção de formato. As linhas 146 e 147 usam o método **getFormatControls** de **CaptureDevice** para obter um **array** de **FormatControls** para a **DataSource inSource**. O objeto que implementa a interface **FormatControl** especifica o formato da **DataSource**. Os objetos **DataSource** podem representar outras fontes de mídia, que não dispositivos de captura, de modo que para este exemplo o operador de coerção na linha 146 manipula o objeto **inSource** como um **CaptureDevice** e acessa métodos de captura de dispositivo como **getFormatControls**. A linha 150 invoca o método **getSelectedFormat** (linhas 213 a 219) para exibir um diálogo de entrada a partir do qual os usuários podem selecionar um dos formatos disponíveis. As linhas 176 a 192 chamam o método **setDeviceFormat** para configurar o formato da mídia de saída para o dispositivo de acordo com o **Format** selecionado pelo usuário. Cada dispositivo de captura pode ter diversos **FormatControls**, de modo que **setDeviceFormat** usa o método **setFormat** de **FormatControl** para especificar o formato para cada objeto **FormatControl**.

Formatar a **DataSource** completa a configuração do dispositivo de captura. O **Processor** (objeto **inSource**) converte os dados para o formato de arquivo no qual eles serão salvos. O **Processor** funciona como conector entre a **outSource** e o método **captureSaveFile**, já que a **DataSource inSource** não reproduz nem salva a mídia: ela serve somente para configurar o dispositivo de captura. A linha 157 invoca o método **captureSaveFile** (linhas 265 a 322) para executar as etapas necessárias para salvar a mídia capturada em um formato de arquivo reconhecível.

Para criar um **Processor**, este programa cria primeiro um **ProcessorModel**, um gabarito para o **Processor**. O **ProcessorModel** determina os atributos de um **Processor** através de um conjunto de informações que inclui um **DataSource** ou **MediaLocator**, os formatos desejados para as trilhas de mídia que o **Processor** irá manipular e um **ContentDescriptor** que indica o tipo de conteúdo da saída. A linha 268 cria um novo **array** de **Format** (**outFormats**) que representa os formatos possíveis para cada trilha na mídia. A linha 270 configura o formato **default** para o primeiro elemento do **array**. Para salvar a saída capturada em um arquivo, o **Pro-**

cessor precisa primeiro converter os dados que ele recebe para um formato habilitado pelo arquivo. Um novo **FileTypeDescriptor** QuickTime (pacote `javax.media.format`) é criado para armazenar uma descrição do tipo de conteúdo da saída do **Processor** e armazena em **outFileType** (linhas 273 e 274). As linhas 282 a 284 usam a **DataSource inSource**, o **array outFormats** e o tipo de arquivo **outFileType** para instanciar um novo **ProcessorModel** (linhas 283 e 284).

Em geral, os **Processors** precisam ser configurados antes que eles possam processar mídia, mas o deste aplicativo não precisa, já que as linhas 282 a 284 invocam o método **createRealizedProcessor** de **Manager**.

Este método cria um **Processor** realizado e configurado, com base no objeto **ProcessorModel** que lhe foi passado como argumento. O método **createRealizedProcessor** dispara uma **NoProcessorException** se o programa não conseguir localizar um **Processor** para a mídia ou se o JMF não suportar o tipo de mídia. O método dispara uma **CannotRealizeException** se o **Processor** não puder ser realizado. Isto pode ocorrer se um outro programa já estiver usando a mídia, bloqueando, portanto, a comunicação com a fonte de mídia.



Erro comum de programação 22.1

Tenha cuidado ao especificar formatos de trilhas. Formatos incompatíveis para tipos de arquivos de saída específicos impedem que o programa realize o **Processor**.



Observação de engenharia de software 22.2

Lembre-se de que o **Processor** faz transições através de diversos estados antes de ser realizado. O método **createProcessor** de **Manager** permite que um programa forneça configuração mais personalizada antes de um **Processor** ser realizado.



Dica de desempenho 22.2

Quando o método **createRealizedProcessor** configura o **Processor**, o método bloqueia até que o **Processor** seja realizado. Isto pode impedir que outras partes do programa sejam executadas. Em alguns casos, usar um **ControllerListener** para atender aos **ControllerEvents** pode habilitar um programa a operar de maneira mais eficiente. Quando o **Processor** está realizado, o ouvinte é notificado, de modo que o programa possa prosseguir com o processamento da mídia.

Tendo obtido os dados de mídia do **Processor** em um formato de arquivo, o programa pode criar um “gravador de dados” para gravar a saída da mídia em um arquivo. O objeto que implementa a interface **DataSink** habilita os dados de mídia a serem enviados para a saída em um endereço específico – na maior parte das vezes um arquivo. A linha 287 invoca o método **makeDataWriter** (linhas 325 a 399) para criar um objeto **DataSink** que pode salvar o arquivo. O método **createDataSink** de **Manager** exige como argumentos a **DataSource** do **Processor** e o **MediaLocator** para o novo arquivo. Dentro de **makeDataWriter**, as linhas 229 a 242 invocam o método **getSaveFile** para pedir aos usuários que especifiquem o nome e o endereço do arquivo no qual a mídia deve ser salva. O objeto **File saveFile** armazena as informações. O método **getDataOutput** de **Processor** (linha 333) devolve a **DataSource** da qual ele recebeu a mídia. As linhas 344 e 345 criam um novo **MediaLocator** para **saveFile**. Usando este **MediaLocator** e a **DataSource**, as linhas 349 e 350 criam um objeto **DataSink** que grava a mídia de saída a partir da **DataSource** para o arquivo no qual os dados serão salvos, como especificado pelo **MediaLocator**. O método **createDataSink** dispara uma **NoDataSinkException** se ele não puder criar um **DataSink** que possa ler os dados da **DataSource** e enviá-los para a saída no endereço especificado pelo **MediaLocator**. Esta falha pode ocorrer como resultado de uma mídia inválida ou um **MediaLocator** inválido.

O programa precisa saber quando parar de enviar dados para a saída, de modo que as linhas 333 a 369 registram um **DataSinkListener** para esperar por **DataSinkEvents**. O método **dataSinkUpdate** de **DataSinkListener** (linhas 359 a 365) é chamado quando ocorre **DataSinkEvent**. Se o **DataSinkEvent** é um **EndOfStreamEvent**, indicando que o **Processor** foi fechado porque a conexão com o fluxo de captura fechou, a linha 364 fecha a **DataSink**. Chamar o método **close** de **DataSink** faz parar a transferência de dados. O **DataSink** fechado não pode ser usado novamente.



Erro comum de programação 22.2

A saída no arquivo de mídia com um **DataSink** ficará corrompida se o **DataSink** não for fechado adequadamente.



Observação de engenharia de software 22.3

A mídia capturada pode não gerar um `EndOfMediaEvent` se o ponto final da mídia não puder ser determinado.

Depois de configurar o `DataSink` e registrar o ouvinte, a linha 372 chama o método `open` de `DataSink` para conectar o `DataSink` ao destino que o `MediaLocator` especifica. O método `open` dispara uma `SecurityException` se o `DataSink` tentar gravar em um destino para o qual o programa não tem permissão de gravação, como um arquivo somente de leitura.

A linha 373 chama o método `start` de `DataSink` para iniciar a transferência de dados. Neste ponto, o programa retorna do método `makeDataWriter` de volta para o método `captureSaveFile` (linhas 265 a 322). Embora o `DataSink` prepare a si mesmo para receber a transferência e indique que está pronto chamando `start`, a transferência não começa efetivamente enquanto o método `start` do `Processor` não é chamado. A invocação do método `start` de `Processor` começa o fluxo de dados do dispositivo de captura, formata os dados e os transfere para o `DataSink`. O `DataSink` grava a mídia no arquivo, o que completa o processo executado pela classe `CapturePlayer`.

Enquanto o `Processor` codifica os dados e o `DataSink` os salva em um arquivo, `CapturePlayer` monitora o processo. O monitoramento fornece um método de supervisionar os dados enquanto o dispositivo de captura os coleta. As linhas 294 a 296 obtêm um objeto que implementa a interface `MonitorControl` (pacote `javax.media.control`) a partir do `Processor` chamando o método `getControl`. A linha 299 chama o método `getControlComponent` de `MonitorControl` para obter o componente de GUI que exibe a mídia monitorada. Os `MonitorControls` normalmente têm uma caixa de marcação para habilitar ou desabilitar a exibição da mídia. Além disso, os dispositivos de áudio têm um controle de volume e os dispositivos de vídeo têm um controle para ajustar a taxa de exibição das frames. A linha 301 invoca o método `showSaveMonitor` (linhas 245 a 261) para exibir a GUI de monitoração em uma caixa de diálogo. Para terminar a captura, os usuários podem pressionar o botão **OK** ou fechar a caixa de diálogo (linhas 254 a 261). Para alguns dispositivos de captura de vídeo, o `Processor` precisa estar em um estado `Stopped` para habilitar a monitoração do processo de salvamento e captura. Até aqui, discutimos os recursos do JMF para acessar, apresentar e salvar conteúdo de mídia. Nosso exemplo final de JMF demonstra como enviar mídia entre computadores com as capacidades de *streaming* do JMF.

22.4 Streaming com RTP

A *mídia de streaming* se refere à mídia que é transferida de um servidor para um cliente em um fluxo contínuo de bytes. O cliente pode começar a reproduzir a mídia enquanto ainda está baixando a mídia do servidor. Os arquivos de mídia de áudio e vídeo frequentemente têm vários megabytes de tamanho. Eventos ao vivo, como transmissões de concertos ou de jogos de futebol, podem ter tamanhos indeterminados. Os usuários podem esperar até que a gravação de um concerto ou jogo seja divulgada, depois baixar a gravação inteira. Entretanto, com as velocidades atuais de conexão da Internet, baixar uma transmissão destas poderia levar dias e normalmente os usuários preferem escutar transmissões ao vivo enquanto elas acontecem. A mídia de *streaming* permite que os aplicativos clientes reproduzam a mídia através da Internet ou através de uma rede, sem baixar o arquivo de mídia inteiro de uma vez só.

Em um aplicativo com mídia de *streaming*, o cliente normalmente se conecta a um servidor que envia um fluxo de bytes contendo a mídia de volta para o cliente. O aplicativo cliente coloca em um *buffer* (i.e., armazena localmente) uma parte da mídia, a qual o cliente começa a reproduzir depois que uma certa quantidade tenha sido recebida. O cliente coloca continuamente mais mídia no *buffer*, fornecendo aos usuários um clipe ininterrupto, enquanto o tráfego na rede não impedir o aplicativo cliente de colocar bytes adicionais no *buffer*. Com o uso de *buffer*, o usuário reproduz a mídia segundos depois que inicia o *streaming*, embora todo o material ainda não tenha sido recebido.



Dica de desempenho 22.3

Fazer *streaming* de mídia para um cliente permite que o cliente reproduza a mídia mais rapidamente do que se o cliente esperar para baixar um arquivo de mídia inteiro.

A demanda por multimídia poderosa em tempo real está aumentando drasticamente, à medida que aumenta a velocidade das conexões na Internet. O acesso à Internet com banda larga, que fornece conexões de rede em alta velocidade com a Internet para tantos usuários, está se tornando mais popular, embora o número de usuários perma-

neça relativamente pequeno em comparação ao número total de usuários da Internet. Com conexões mais rápidas, a mídia de *streaming* pode possibilitar uma experiência melhor com multimídia. Os usuários com conexões mais lentas ainda podem experimentar a multimídia, mas com qualidade inferior. A ampla variedade de aplicativos que usam mídia de *streaming* está crescendo. Os aplicativos que fazem *streaming* de clipes de vídeo para os clientes se expandiram para fornecer transmissões em tempo real. Milhares de estações de rádio transmitem música em *stream* continuamente através da Internet. Os aplicativos clientes como o RealPlayer se concentraram no conteúdo de mídia de *streaming* com transmissões de rádio ao vivo. Os aplicativos não se limitam ao *streaming* de áudio e vídeo de servidor para cliente. Por exemplo, os aplicativos de teleconferência e videoconferência aumentam a eficiência nos negócios do dia-a-dia, reduzindo a necessidade de se viajar grandes distâncias para se participar de reuniões. O JMF fornece um pacote de mídia de *streaming* em alguns dos formatos discutidos antes neste capítulo. Para obter uma lista completa de formatos, consulte o *site* oficial do JMF:

```
java.sun.com/products/java-media/jmf/2.1.1/formats.html
```

O JMF usa o padrão de mercado *Real-Time Transport Protocol (RTP)* para controlar a transmissão de mídia. O RTP foi projetado especificamente para transmitir dados de mídia em tempo real. Os dois mecanismos para fazer *streaming* de mídia suportada por RTP são passá-la através de um **DataSink** e colocá-la em um *buffer*.

O mecanismo mais fácil de se usar é um **DataSink**, que escreve o conteúdo de um *stream* em um *host* de destino (i.e., um computador cliente), através das mesmas técnicas mostradas na Fig. 22.2 para salvar mídia capturada em um arquivo. Neste caso, entretanto, o URL do **MediaLocator** do destino seria especificado no seguinte formato:

```
rtp://host:porta/tipoDeConteúdo
```

onde *host* é o endereço IP ou nome de *host* do servidor, *porta* é o número da porta na qual o servidor está fazendo *streaming* da mídia e *tipoDeConteúdo* é **audio** ou **video**.

Usar um **DataSink** como especificado permite que somente um *stream* seja enviado de cada vez. Para enviar vários *streams* (por exemplo, como seria enviado um vídeo de karaokê com trilhas separadas para vídeo e áudio) para vários *hosts*, um aplicativo servidor precisa usar *gerenciadores de sessão de RTP*. O **RTPManager** (pacote **javax.media.rtp**) permite maior controle sobre o processo de *streaming*, permitindo a especificação de tamanhos de *buffers*, verificação de erros e relatórios de *streaming* sobre o atraso de propagação (o tempo que leva para os dados chegarem a seu destino).

O programa nas Figs. 22.3 e 22.4 demonstra o *streaming* com o gerenciador de sessão de RTP. Este exemplo suporta o envio de vários *streams* em paralelo, de modo que clientes separados precisam ser abertos para cada *stream*. Este exemplo não mostra um cliente que pode receber o *stream* de RTP. O programa na Fig. 22.1 (**SimplePlayer**) pode testar o servidor RTP especificando um endereço de sessão de RTP

```
rtp://127.0.0.1:4000/audio
```

como o endereço que o **SimplePlayer** deve abrir para começar a reproduzir áudio. Para executar o servidor de mídia de *streaming* em um computador diferente, substitua **127.0.0.1** pelo endereço IP ou pelo nome de *host* do computador servidor.

O objetivo da classe **RTPServer** é fazer *streaming* de conteúdo de mídia. Como ocorre em exemplos anteriores, o **RTPServer** (Fig. 22.3) configura a mídia, processa-a e formata-a, e depois a envia para a saída. Os **ControllerEvents** e os diversos estados do processo de *streaming* conduzem este processo. O processamento da mídia tem três partes distintas – inicialização do **Processor**, configuração do **Format** e transmissão dos dados. O código neste exemplo contém inúmeras mensagens de confirmação exibidas no *prompt* de comando e uma ênfase na verificação de erros. Um problema durante o *streaming* provavelmente terminará o processo inteiro e será necessário recomençar.

```
1 // Fig. 22.3: RTPServer.java
2 // Oferece recursos de configuração e transmissão
3 // para arquivos de mídia suportados por RTP
4
5 // Pacotes do núcleo de Java
6 import java.io.*;
```

Fig. 22.3 Servindo mídia de *streaming* com gerenciadores de sessão RTP (parte 1 de 6).

```

7  import java.net.*;
8
9  // Pacotes de extensão de Java
10 import javax.media.*;
11 import javax.media.protocol.*;
12 import javax.media.control.*;
13 import javax.media.rtp.*;
14 import javax.media.format.*;
15
16 public class RTPServer {
17
18     // endereço IP, arquivo ou nome do MediaLocator, número da porta
19     private String ipAddress, fileName;
20     private int port;
21
22     // processador que controla o fluxo de dados
23     private Processor processor;
24
25     // dados de saída do processador a serem enviados
26     private DataSource outSource;
27
28     // controles configuráveis das trilhas de mídia
29     private TrackControl tracks[];
30
31     // gerenciador de sessão de RTP
32     private RTPManager rtpManager[];
33
34     // construtor para RTPServer
35     public RTPServer( String locator, String ip, int portNumber )
36     {
37         fileName = locator;
38         port = portNumber;
39         ipAddress = ip;
40     }
41
42     // inicializa e configura o processador
43     // devolve true se bem-sucedido, false caso contrário
44     public boolean beginSession()
45     {
46         // obtém MediaLocator de endereço específico
47         MediaLocator mediaLocator = new MediaLocator( fileName );
48
49         if ( mediaLocator == null ) {
50             System.err.println(
51                 "No MediaLocator found for " + fileName );
52
53             return false;
54         }
55
56         // cria processador a partir de MediaLocator
57         try {
58             processor = Manager.createProcessor( mediaLocator );
59
60             // registra um ControllerListener para o processador
61             // para esperar eventos de transição de estado
62             processor.addControllerListener(
63                 new ProcessorEventHandler() );
64
65             System.out.println( "Processor configuring..." );
66

```

Fig. 22.3 Servindo mídia de *streaming* com gerenciadores de sessão RTP (parte 2 de 6).

```

67      // configura o processador antes de ajustá-lo
68      processor.configure();
69  }
70
71      // erro de conexão com a fonte
72      catch ( IOException ioException ) {
73          ioException.printStackTrace();
74          return false;
75      }
76
77      // exceção disparada quando nenhum processador
78      // pode ser encontrado para fonte de dados específica
79      catch ( NoProcessorException noProcessorException ) {
80          noProcessorException.printStackTrace();
81          return false;
82      }
83
84      return true;
85
86  } // fim do método beginSession
87
88      // tratador ControllerListener para o processador
89      private class ProcessorEventHandler
90          extends ControllerAdapter {
91
92          // configura formato de saída e realiza
93          // o processador configurado
94          public void configureComplete(
95              ConfigureCompleteEvent configureCompleteEvent )
96          {
97              System.out.println( "\nProcessor configured." );
98
99              setOutputFormat();
100
101              System.out.println( "\nRealizing Processor...\n" );
102
103              processor.realize();
104          }
105
106          // começa a enviar quando o processador está realizado
107          public void realizeComplete(
108              RealizeCompleteEvent realizeCompleteEvent )
109          {
110              System.out.println(
111                  "\nInitialization successful for " + fileName );
112
113              if ( transmitMedia() == true )
114                  System.out.println( "\nTransmission setup OK" );
115
116              else
117                  System.out.println( "\nTransmission failed." );
118          }
119
120          // faz parar a sessão de RTP quando não há mídia a enviar
121          public void endOfMedia( EndOfMediaEvent mediaEndEvent )
122          {
123              stopTransmission();
124              System.out.println ( "Transmission completed." );
125          }
126

```

Fig. 22.3 Servindo mídia de *streaming* com gerenciadores de sessão RTP (parte 3 de 6).

```

127     } // fim da classe interna ProcessorEventHandler
128
129     // configura o formato de saída de todas as trilhas na mídia
130     public void setOutputFormat()
131     {
132         // configura o tipo de conteúdo da saída para formato suportado por RTP
133         processor.setContentDescriptor(
134             new ContentDescriptor( ContentDescriptor.RAW_RTP ) );
135
136         // obtém todos os controles de trilha do processador
137         tracks = processor.getTrackControls();
138
139         // formatos de uma trilha suportados por RTP
140         Format rtpFormats[];
141
142         // configura cada trilha para o primeiro formato suportado
143         // por RTP encontrado naquela trilha
144         for ( int i = 0; i < tracks.length; i++ ) {
145
146             System.out.println( "\nTrack #" +
147                 ( i + 1 ) + " supports " );
148
149             if ( tracks[ i ].isEnabled() ) {
150
151                 rtpFormats = tracks[ i ].getSupportedFormats();
152
153                 // se existirem formatos da trilha suportados, exibe
154                 // todos os formatos suportados por RTP e configura
155                 // o formato de trilha para o primeiro formato suportado
156                 if ( rtpFormats.length > 0 ) {
157
158                     for ( int j = 0; j < rtpFormats.length; j++ )
159                         System.out.println( rtpFormats[ j ] );
160
161                     tracks[ i ].setFormat( rtpFormats[ 0 ] );
162
163                     System.out.println ( "Track format set to " +
164                         tracks[ i ].getFormat() );
165                 }
166
167                 else
168                     System.err.println (
169                         "No supported RTP formats for track!" );
170
171             } // fim do if
172
173         } // fim do laço for
174
175     } // fim do método setOutputFormat
176
177     // envia mídia com valor booleano indicando sucesso
178     public boolean transmitMedia()
179     {
180         outSource = processor.getDataOutput();
181
182         if ( outSource == null ) {
183             System.out.println ( "No data source from media!" );
184
185             return false;

```

Fig. 22.3 Servindo mídia de *streaming* com gerenciadores de sessão RTP (parte 4 de 6).

```

186     }
187
188     // gerenciadores de stream de RTP para cada trilha
189     rtpManager = new RTPManager[ tracks.length ];
190
191     // endereços de sessão de RTP de destino e local
192     SessionAddress localAddress, remoteAddress;
193
194     // stream RTP que está sendo enviado
195     SendStream sendStream;
196
197     // endereço IP
198     InetAddress ip;
199
200     // inicializa endereços de transmissão e envia a mídia para a saída
201     try {
202
203         // transmite todas as trilhas na mídia
204         for ( int i = 0; i < tracks.length; i++ ) {
205
206             // instancia um RTPManager
207             rtpManager[ i ] = RTPManager.newInstance();
208
209             // adiciona 2 para especificar o número de porta do próximo controle;
210             // (o RTP Session Manager usa 2 portas)
211             port += ( 2 * i );
212
213             // obtém endereço IP do host a partir do string ipAddress
214             ip = InetAddress.getByName( ipAddress );
215
216             // encapsula par de endereços IP para controle e
217             // dados com duas portas dentro do endereço de sessão local
218             localAddress = new SessionAddress(
219                 ip.getLocalHost(), port );
220
221             // obtém o endereço de sessão remoteAddress
222             remoteAddress = new SessionAddress( ip, port );
223
224             // inicializa a sessão
225             rtpManager[ i ].initialize( localAddress );
226
227             // abre a sessão de RTP para o destino
228             rtpManager[ i ].addTarget( remoteAddress );
229
230             System.out.println( "\nStarted RTP session: "
231                 + ipAddress + " " + port );
232
233             // cria stream de envio na sessão de RTP
234             sendStream =
235                 rtpManager[ i ].createSendStream( outSource, i );
236
237             // começa a enviar o stream
238             sendStream.start();
239
240             System.out.println( "Transmitting Track #" +
241                 ( i + 1 ) + " ... " );
242
243         } // fim do laço for
244

```

Fig. 22.3 Servindo mídia de *streaming* com gerenciadores de sessão RTP (parte 5 de 6).

```

245         // começa carga da mídia
246         processor.start();
247
248     } // fim de try
249
250     // endereço local desconhecido ou endereço remoto não pode ser resolvido
251     catch ( InvalidSessionAddressException addressError ) {
252         addressError.printStackTrace();
253         return false;
254     }
255
256     // erro de conexão com a DataSource
257     catch ( IOException ioException ) {
258         ioException.printStackTrace();
259         return false;
260     }
261
262     // formato não configurado ou formato inválido configurado na fonte de stream
263     catch ( UnsupportedFormatException formatException ) {
264         formatException.printStackTrace();
265         return false;
266     }
267
268     // transmissão inicializada com sucesso
269     return true;
270
271 } // fim do método transmitMedia
272
273 // faz parar a transmissão e fecha recursos
274 public void stopTransmission()
275 {
276     if ( processor != null ) {
277
278         // faz parar o processador
279         processor.stop();
280
281         // descarta o processador
282         processor.close();
283
284         if ( rtpManager != null )
285
286             // fecha alvos de destino
287             // e descarta gerenciadores de RTP
288             for ( int i = 0; i < rtpManager.length; i++ ) {
289
290                 // fecha streams para todos os destinos
291                 // com um motivo para terminar
292                 rtpManager[ i ].removeTargets(
293                     "Session stopped." );
294
295                 // libera os recursos da sessão de RTP
296                 rtpManager[ i ].dispose();
297             }
298
299     } // fim de if
300
301     System.out.println ( "Transmission stopped." );
302
303 } // fim do método stopTransmission
304
305 } // fim da classe RTPServer

```

Fig. 22.3 Servindo mídia de *streaming* com gerenciadores de sessão RTP (parte 6 de 6).

Para testar **RTPServer**, a classe **RTPServerTest** (Fig. 22.4) cria um novo objeto **RTPServer** e passa para o seu construtor (linhas 35 a 40) três argumentos – um **String** que representa o endereço da mídia, um **String** que representa o endereço IP do cliente e um número de porta para o conteúdo de *streaming*. Estes argumentos contêm as informações que a classe **RTPServer** precisa para obter a mídia e configurar o processo de *streaming*. Seguindo a abordagem genérica delineada em **SimplePlayer** (Fig. 22.1) e **CapturePlayer** (Fig. 22.2), a classe **RTPServer** obtém uma fonte de mídia, configura a fonte através de um tipo de **Controller** e envia os dados de saída para um destino especificado.

RTPServerTest chama o método **beginSession** de **RTPServer** (linhas 44 a 86) para configurar o **Processor** que controla o fluxo de dados. A linha 47 cria um **MediaLocator** e o inicializa com o endereço da mídia armazenado em **fileName**. A linha 58 cria um **Processor** para os dados especificados por aquele **MediaLocator**.

Diferentemente do programa na Fig. 22.2, este **Processor** não é configurado previamente por um **Manager**. Até que a classe **RTPServer** configure e realize o **Processor**, a mídia não pode ser formatada. As linhas 62 e 63 registram um **ProcessorEventHandler** para reagir aos **ControllerEvents** de **processor**. Os métodos da classe **ProcessorEventHandler** (linhas 89 a 127) controlam a configuração da mídia enquanto o **Processor** muda de estados. A linha 68 invoca o método **configure** de **Processor** para colocar o **Processor** no estado *Configuring*. A configuração ocorre quando o **Processor** pede ao sistema e à mídia as informações necessárias para programar o **Processor** para executar a tarefa correta. Ocorre um **ConfigureCompleteEvent** quando o **Processor** completa a configuração. O método **configureComplete** de **ProcessEventHandler** (linhas 94 a 104) responde a esta transição. O método **configureComplete** chama o método **setOutputFormat** (linhas 130 a 175) e depois realiza o **Processor** (linha 103). Quando a linha 99 invoca o método **setOutputFormat**, ela configura cada trilha de mídia para um formato de mídia de *streaming* de RTP. As linhas 133 e 134 no método **setOutputFormat** especificam o tipo de conteúdo da saída chamando o método **setContentDescriptor** de **Processor**. O método recebe como argumento um **ContentDescriptor** inicializado com a constante **ContentDescriptor.RAW_RTP**. O tipo de conteúdo da saída de RTP restringe o **Processor** para suportar somente formatos de trilhas de mídia preparadas para RTP. O tipo de conteúdo da saída do **Processor** deve ser configurado antes que as trilhas da mídia sejam configuradas.

Uma vez que o **Processor** esteja configurado, os formatos das trilhas de mídia devem ser ajustados. A linha 137 invoca o método **getTrackControls** de **Processor** para obter um **array** que contém o objeto **TrackControl** correspondente (pacote **javax.media.control**) para cada trilha da mídia. Para cada **TrackControl** habilitado, as linhas 144 a 173 obtêm um **array** de todos os **Formats** de mídia RTP suportados (linha 151) e depois configura o primeiro formato RTP suportado como o formato preferencial para *streaming* com RTP para aquela trilha (linha 161). Quando o método **setOutputFormat** retorna, a linha 103 no método **configureComplete** realiza o **Processor**.

Como ocorre em qualquer realização de controlador, o **Processor** pode enviar mídia para a saída assim que ele tenha terminado de realizar a si mesmo. Quando o **Processor** passa para o estado *Realized*, o **ProcessorEventHandler** invoca o método **realizeComplete** (linhas 107 a 118). A linha 113 invoca o método **transmitMedia** (linhas 178 a 271), que cria as estruturas necessárias para transmitir a mídia para o **Processor**. Este método obtém a **DataSource** do **Processor** (linha 180), depois declara um **array** de **RTPManagers** que são capazes de iniciar e controlar uma sessão de RTP (linha 189). **RTPManagers** usam um par de objetos **SessionAddress** com endereços IP idênticos, mas números de porta diferentes – um para controle do *stream* e um para dados da mídia de *streaming*. O **RTPManager** recebe cada endereço IP e número de porta como um objeto **SessionAddress**. A linha 192 declara os objetos **SessionAddress** usados no processo de *streaming*. O objeto que implementa a interface **SendStream** (linha 195) faz o *streaming* com RTP.



Observação de engenharia de software 22.4

Para os vídeos que têm múltiplas trilhas, cada **SendStream** deve ter seu próprio **RTPManager** gerenciando sua sessão. Cada **Track** tem seu próprio **SendStream**.

O bloco **try** (linhas 201 a 248) do método **transmitMedia** envia para a saída cada trilha da mídia como um *stream* de RTP. Primeiro, devem ser criados gerenciadores para as sessões. A linha 207 invoca o método **newInstance** de **RTPManager** para instanciar um **RTPManager** para cada *stream* de trilha. A linha 211 atribui ao número de porta um valor 2 a mais do que o número de porta anterior, porque cada trilha usa um número de porta para o controle do *stream* e um para realmente fazer *streaming* dos dados. As linhas 218 e 219 instanciam um novo

endereço local de sessão para onde o *stream* está localizado (i.e., o endereço de RTP que os clientes usam para obter o *stream* de mídia) com o endereço IP local e um número de porta como parâmetros. A linha 219 invoca o método `getLocalHost` de `InetAddress` para obter o endereço IP local. A linha 222 instancia o endereço de sessão do cliente, que o `RTPManager` usa como o alvo de destino do *stream*. Quando a linha 225 chama o método `initialize` de `RTPManager`, o método direciona a sessão de *streaming* local para usar o endereço de sessão local. Usando o objeto `remoteAddress` como parâmetro, a linha 228 chama o método `addTarget` de `RTPManager` para abrir a sessão de destino no endereço especificado. Para fazer *stream* de mídia para vários clientes, chama o método `addTarget` de `RTPManager` para cada endereço de destino. O método `addTarget` deve ser chamado após a sessão ser inicializada e antes que qualquer *stream* seja criado durante a sessão.

Agora o programa pode criar os *streams* na sessão e começar a enviar dados. Os *streams* são criados na sessão de RTP atual com a `DataSource outSource` (obtida na linha 180) e o índice de fonte de *stream* (i.e., índice de trilha da mídia) nas linhas 234 e 235. Invocar o método `start` sobre o `SendStream` (linha 238) e sobre o `Processor` (linha 246) inicia a transmissão dos *streams* de mídia, o que pode provocar exceções. Ocorre uma `InvalidSessionAddressException` quando o endereço de sessão especificado é inválido. Ocorre uma `UnsupportedFormatException` quando um formato de mídia não-suportado é especificado ou se o `Format` da `DataSource` não foi configurado. Ocorre uma `IOException` se o aplicativo encontra problemas na rede. Durante o processo de *streaming*, `RTPManagers` podem ser usados com classes relacionadas dos pacotes `javax.media.rtp` e `javax.media.rtp.event` para controlar o processo de *streaming* e enviar relatórios para o aplicativo.

O programa deve fechar as conexões e parar a transmissão em *streaming* quando ele atinge o fim da mídia de *streaming* ou quando o programa termina. Quando o `Processor` encontra o fim da mídia, ele gera um `EndOfMediaEvent`. Em resposta, o programa chama o método `endOfMedia` (linhas 121 a 125). A linha 123 invoca o método `stopTransmission` (linhas 274 a 303) para parar e fechar o `Processor` (linhas 279 a 282). Depois de chamar `stopTransmission`, não é possível retomar o *streaming* porque ele descarta o `Processor` e os recursos da sessão de RTP. As linhas 288 a 297 invocam o método `removeTargets` de `RTPManager` (linhas 292 e 293) para fechar o *streaming* para todos os destinos. O método `dispose` de `RTPManager` (linha 296) também é invocado, liberando os recursos mantidos pelas sessões de RTP. A classe `RTPServerTest` (Fig. 22.4) invoca explicitamente o método `stopTransmission` quando o usuário termina o aplicativo servidor (linha 40).

```

1  // Fig. 22.4: RTPServerTest.java
2  // Testa a classe para RTPServer
3
4  // Pacotes do núcleo de Java
5  import java.awt.event.*;
6  import java.io.*;
7  import java.net.*;
8
9  // Pacotes de extensão de Java
10 import javax.swing.*;
11
12 public class RTPServerTest extends JFrame {
13
14     // objeto que trata de streaming com RTP
15     private RTPServer rtpServer;
16
17     // fontes de mídia e endereços de destino
18     private int port;
19     private String ip, mediaLocation;
20     private File mediaFile;
21
22     // botões da GUI
23     private JButton transmitFileButton, transmitUrlButton;
24
25     // construtor para RTPServerTest
26     public RTPServerTest()
27     {

```

Fig. 22.4 Aplicativo para testar a classe `RTPServer` da Fig. 22.3 (parte 1 de 5).

```

28     super( "RTP Server Test" );
29
30     // registra um WindowListener para eventos da frame
31     addWindowListener(
32
33         // classe interna anônima para tratar WindowEvents
34         new WindowAdapter() {
35
36             public void windowClosing(
37                 WindowEvent windowEvent )
38             {
39                 if ( rtpServer != null )
40                     rtpServer.stopTransmission();
41             }
42
43         } // fim de WindowAdpater
44
45     ); // fim da chamada para o método addWindowListener
46
47     // painel contendo GUI de botão
48     JPanel buttonPanel = new JPanel();
49     getContentPane().add( buttonPanel );
50
51     // GUI de botão para transmitir arquivo
52     transmitFileButton = new JButton( "Transmit File" );
53     buttonPanel.add( transmitFileButton );
54
55     // registra ActionListener para eventos de transmitFileButton
56     transmitFileButton.addActionListener(
57         new ButtonHandler() );
58
59     // GUI para botão de URL de transmissão
60     transmitUrlButton = new JButton( "Transmit Media" );
61     buttonPanel.add( transmitUrlButton );
62
63     // registra ActionListener para eventos de transmitUrlButton
64     transmitUrlButton.addActionListener(
65         new ButtonHandler() );
66
67 } // fim do construtor
68
69 // classe interna que trata eventos do botão de transmissão
70 private class ButtonHandler implements ActionListener {
71
72     // abre e tenta enviar arquivo para destino digitado pelo usuário
73     public void actionPerformed( ActionEvent actionEvent )
74     {
75         // se transmitFileButton foi invocado, obtém string de URL do arquivo
76         if ( actionEvent.getSource() == transmitFileButton ) {
77
78             mediaFile = getFile();
79
80             if ( mediaFile != null )
81
82                 // obtém string de URL do arquivo
83                 try {
84                     mediaLocation = mediaFile.toURL().toString();
85                 }
86
87                 // caminho para o arquivo não pode ser resolvido
88                 catch ( MalformedURLException badURL ) {

```

Fig. 22.4 Aplicativo para testar a classe RTPServer da Fig. 22.3 (parte 2 de 5).

```

89         badURL.printStackTrace();
90     }
91
92     else
93         return;
94
95 } // fim do if
96
97 // senão, transmitMediaButton foi invocado, obtém endereço
98 else
99     mediaLocation = getMediaLocation();
100
101 if ( mediaLocation == null )
102     return;
103
104 // obtém endereço IP
105 ip = getIP();
106
107 if ( ip == null )
108     return;
109
110 // obtém número de porta
111 port = getPort();
112
113 // verifica se o número de porta é positivo e válido
114 if ( port <= 0 ) {
115
116     if ( port != -999 )
117         System.err.println( "Invalid port number!" );
118
119     return;
120 }
121
122 // instancia novo servidor de streaming RTP
123 rtpServer = new RTPServer( mediaLocation, ip, port );
124
125 rtpServer.beginSession();
126
127 } // fim do método actionPerformed
128
129 } // fim da classe interna ButtonHandler
130
131 // obtém arquivo do computador
132 public File getFile()
133 {
134     JFileChooser fileChooser = new JFileChooser();
135
136     fileChooser.setFileSelectionMode(
137         JFileChooser.FILES_ONLY );
138
139     int result = fileChooser.showOpenDialog( this );
140
141     if ( result == JFileChooser.CANCEL_OPTION )
142         return null;
143
144     else
145         return fileChooser.getSelectedFile();
146 }
147
148 // obtém endereço da mídia do usuário
149 public String getMediaLocation()
150 {

```

Fig. 22.4 Aplicativo para testar a classe RTPServer da Fig. 22.3 (parte 3 de 5).

```

151         String input = JOptionPane.showInputDialog(
152             this, "Enter MediaLocator" );
153
154         // se o usuário pressiona OK sem digitar algo
155         if ( input != null && input.length() == 0 ) {
156             System.err.println( "No input!" );
157             return null;
158         }
159
160         return input;
161     }
162
163     // método que obtém string de IP do usuário
164     public String getIP()
165     {
166         String input = JOptionPane.showInputDialog(
167             this, "Enter IP Address: " );
168
169         // se o usuário pressiona OK sem digitar algo
170         if ( input != null && input.length() == 0 ) {
171             System.err.println( "No input!" );
172             return null;
173         }
174
175         return input;
176     }
177
178     // obtém número de porta
179     public int getPort()
180     {
181         String input = JOptionPane.showInputDialog(
182             this, "Enter Port Number: " );
183
184         // devolve valor indicador se o usuário clica em OK sem digitar algo
185         if ( input != null && input.length() == 0 ) {
186             System.err.println( "No input!" );
187             return -999;
188         }
189
190         // devolve valor indicador se o usuário clicou em CANCEL
191         if ( input == null )
192             return -999;
193
194         // senão, devolve dados digitados
195         return Integer.parseInt( input );
196     }
197     // fim do método getPort
198
199     // executa o aplicativo
200     public static void main( String args[] )
201     {
202         RTPServerTest serverTest = new RTPServerTest();
203
204         serverTest.setSize( 250, 70 );
205         serverTest.setLocation( 300, 300 );

```

Fig. 22.4 Aplicativo para testar a classe RTPServer da Fig. 22.3 (parte 4 de 5).

```

206     serverTest.setDefaultCloseOperation( EXIT_ON_CLOSE );
207     serverTest.setVisible( true );
208 }
209
210 } // fim da classe RTPServerTest

```

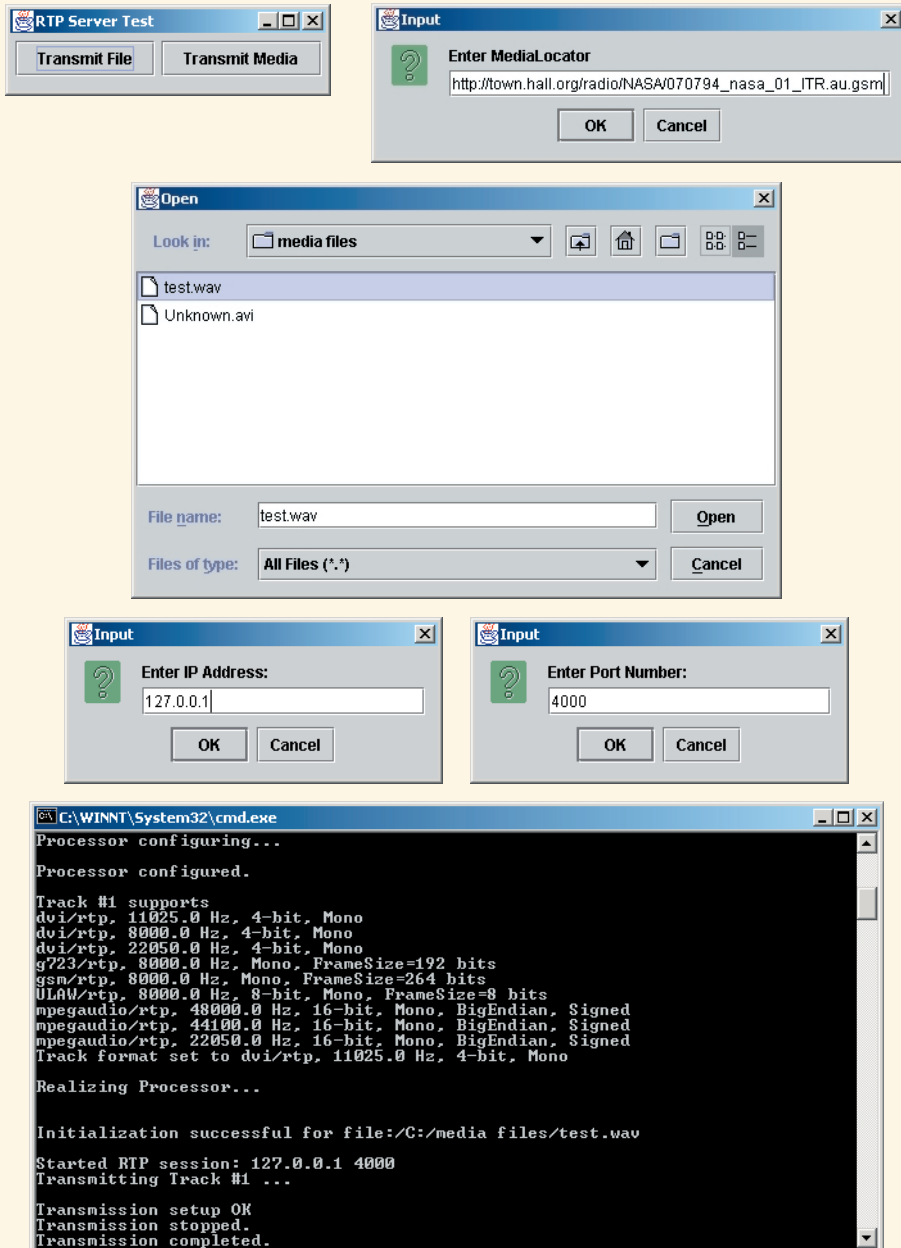


Fig. 22.4 Aplicativo para testar a classe RTPServer da Fig. 22.3 (parte 5 de 5).

22.5 Java Sound

Muitos dos programas de computador de hoje em dia atraem a atenção do usuário com recursos de áudio. Até mesmo *applets* e aplicativos básicos podem melhorar a experiência do usuário com sons ou clipes simples de música. Com interfaces de programação de som, os desenvolvedores podem criar aplicativos que reproduzem sons em resposta às interações do usuário. Por exemplo, em muitos aplicativos, quando ocorre um erro e uma caixa de diálogo aparece na tela, o diálogo frequentemente é acompanhado por um som. Os usuários, portanto, recebem tanto indicações visuais quanto auditivas de que ocorreu um problema. Como outro exemplo, os programadores de jogos usam recursos de áudio extensivos para melhorar a experiência dos jogadores.

A API *Java Sound* é uma maneira de incorporar mídia de áudio em aplicativos mais simples do que o *Java Media Framework*. A API *Java Sound* vem junto com o *Java 2 Software Development Kit* versão 1.3. A API consiste em quatro pacotes – `javax.sound.midi`, `javax.sound.midi.spi`, `javax.sound.sampled` e `javax.sound.sampled.spi`. As próximas duas seções se concentram nos pacotes `javax.sound.midi` e `javax.sound.sampled`, que fornecem classes e interfaces para acessar, manipular e reproduzir áudio *Musical Instrument Digital Interface* (MIDI) e como amostra. Os pacotes que terminam com `.spi` fornecem aos desenvolvedores ferramentas para adicionar o suporte ao *Java Sound* a formatos de áudio adicionais que estão além do escopo deste livro.

A API *Java Sound* fornece acesso à *Java Sound Engine*, que cria áudio digitalizado e captura mídia a partir dos dispositivos de som discutidos na Seção 22.3. O *Java Sound* exige uma placa de som para reproduzir áudio. Um programa que usa *Java Sound* irá disparar uma exceção se ele acessar recursos de áudio do sistema em um computador que não tem uma placa de som.

22.6 Reproduzindo amostras de áudio

Esta seção apresenta os recursos do pacote `javax.sound.sampled` para reproduzir formatos de arquivos de amostras de som, que incluem *Sun Audio* (`.au`), *Windows Waveform* (`.wav`) e *Macintosh Audio Interchange File Format* (`.aiff`). O programa nas Fgs. 22.5 e 22.6 mostra como se reproduz áudio como estes formatos de arquivo.

Ao se processar dados de áudio, uma *linha* fornece o caminho através do qual o áudio flui em um sistema. Um exemplo de uma linha é um par de fones de ouvido conectados a um reproduzidor de CD.

A classe `ClipPlayer` (Fig. 22.5) é um exemplo de como as linhas podem ser usadas. Ela contém um objeto que implementa a interface `Clip`, que por sua vez estende a interface `DataLine`. O `Clip` é uma linha que processa um arquivo de áudio inteiro em vez de ler continuamente de um *stream* de áudio. As `DataLines` melhoram as `Lines` fornecendo métodos adicionais (como `start` e `stop`) para controlar o fluxo de dados, e os `Clips` melhoram as `DataLines` fornecendo métodos para abrir `Clips` e métodos para controle preciso sobre reprodução e repetição do áudio.

```

1  // Fig. 22.5: ClipPlayer.java
2  // Reproduz arquivos de clipes de áudio dos tipos WAV, AU, AIFF
3
4  // Pacotes do núcleo de Java
5  import java.io.*;
6
7  // Pacotes de extensão de Java
8  import javax.sound.sampled.*;
9
10 public class ClipPlayer implements LineListener {
11
12     // stream de entrada de áudio
13     private AudioInputStream soundStream;
14
15     // linha de clipe de amostra de áudio
16     private Clip clip;
17

```

Fig. 22.5 `ClipPlayer` reproduz um arquivo de áudio (parte 1 de 4).

```

18 // arquivo de clipe de áudio
19 private File soundFile;
20
21 // booleana que indica repetição do áudio
22 private boolean replay = false;
23
24 // construtor para ClipPlayer
25 public ClipPlayer( File audioFile )
26 {
27     soundFile = audioFile;
28 }
29
30 // abre arquivo de música, devolvendo true se bem-sucedido
31 public boolean openFile()
32 {
33     // obtém stream de áudio do arquivo
34     try {
35         soundStream =
36             AudioSystem.getAudioInputStream( soundFile );
37     }
38
39     // arquivo de áudio não-suportado por JavaSound
40     catch ( UnsupportedOperationException audioException ) {
41         audioException.printStackTrace();
42         return false;
43     }
44
45     // erro de E/S quando tentava obter stream
46     catch ( IOException ioException ) {
47         ioException.printStackTrace();
48         return false;
49     }
50
51     // invoca loadClip, devolvendo true se carga bem-sucedida
52     return loadClip();
53
54 } // fim do método openFile
55
56 // carrega clipe de som
57 public boolean loadClip ()
58 {
59     // obtém linha de clipe para o arquivo
60     try {
61
62         // obtém formato de áudio do arquivo de som
63         AudioFormat audioFormat = soundStream.getFormat();
64
65         // define informações da linha com base no tipo de linha,
66         // codificação e tamanhos de frame do arquivo de áudio
67         DataLine.Info dataLineInfo = new DataLine.Info(
68             Clip.class, AudioSystem.getTargetFormats(
69                 AudioFormat.Encoding.PCM_SIGNED, audioFormat ),
70                 audioFormat.getFrameSize(),
71                 audioFormat.getFrameSize() * 2 );
72
73         // assegura que o sistema de som suporta linha de dados
74         if ( !AudioSystem.isLineSupported( dataLineInfo ) ) {
75
76             System.err.println( "Unsupported Clip File!" );
77             return false;

```

Fig. 22.5 ClipPlayer reproduz um arquivo de áudio (parte 2 de 4).

```

78         }
79
80         // obtém recurso para a linha de clipe
81         clip = ( Clip ) AudioSystem.getLine( dataLineInfo );
82
83         // espera por eventos da linha de clipe
84         clip.addLineListener( this );
85
86         // abre clipe de áudio e obtém os recursos necessários do sistema
87         clip.open( soundStream );
88
89     } // fim do try
90
91     // recurso de linha não-disponível
92     catch ( LineUnavailableException noLineException ) {
93         noLineException.printStackTrace();
94         return false;
95     }
96
97     // erro de E/S durante interpretação dos dados de áudio
98     catch ( IOException ioException ) {
99         ioException.printStackTrace();
100        return false;
101    }
102
103    // arquivo de clipe carregado com sucesso
104    return true;
105
106 } // fim do método loadClip
107
108 // inicia reprodução do clipe de áudio
109 public void play()
110 {
111     clip.start();
112 }
113
114 // método listener de eventos de linha para parar ou repetir no fim do clipe
115 public void update( LineEvent lineEvent )
116 {
117     // se o clipe chega ao fim, fecha o clipe
118     if ( lineEvent.getType() == LineEvent.Type.STOP &&
119         !replay )
120         close();
121
122     // se repetição ligada, repete para sempre
123     else
124
125         if ( lineEvent.getType() == LineEvent.Type.STOP &&
126             replay ) {
127
128             System.out.println( "replay" );
129
130             // repete o clipe para sempre
131             clip.loop( Clip.LOOP_CONTINUOUSLY );
132         }
133     }
134
135     // configura repetição do clipe
136     public void setReplay( boolean value )

```

Fig. 22.5 ClipPlayer reproduz um arquivo de áudio (parte 3 de 4).

```

137     {
138         replay = value;
139     }
140
141     // pára e fecha o clipe, devolvendo recursos do sistema
142     public void close()
143     {
144         if ( clip != null ) {
145             clip.stop();
146             clip.close();
147         }
148     }
149
150 } // fim da classe ClipPlayer

```

Fig. 22.5 ClipPlayer reproduz um arquivo de áudio (parte 4 de 4).

Todas as **Lines** geram **LineEvents**, que podem ser tratados por **LineListeners**. Os **LineEvents** ocorrem quando se está iniciando, parando, reproduzindo e fechando um objeto **Line**. Embora uma **Line** pare a reprodução automaticamente quando ela atinge o fim de um arquivo de áudio, a classe **ClipPlayer** implementa a interface **LineListener** (linha 10) e pode fechar o **Clip** permanentemente ou repetir o **Clip** (discutido em seguida). Os **LineListeners** são úteis para as tarefas que precisam ser sincronizadas com os estados de **LineEvent** de uma linha.

O **Clip** lê dados de áudio de um **AudioInputStream** (uma subclasse de **InputStream**), que fornece acesso ao conteúdo de dados do *stream*. Este exemplo carrega os cliques dos dados de áudio antes de tentar reproduzi-lo e, portanto, é capaz de determinar o tamanho do clipe em *frames*. Cada *frame* representa dados em um intervalo de tempo específico no arquivo de áudio. Para reproduzir os arquivos de amostra de áudio com o Java Sound, o programa precisa obter um **AudioInputStream** de um arquivo de áudio, obter uma linha de **Clip** formatada, carregar o **AudioInputStream** na linha de **Clip** e iniciar o fluxo de dados na linha de **Clip**.

Para reproduzir a amostra de áudio, o *stream* de áudio deve ser obtido de um arquivo de áudio. O método **openFile** de **ClipPlayer** (linhas 31 a 54) obtém áudio do **soundFile** (inicializado no construtor de **ClipPlayer** nas linhas 25 a 28). As linhas 35 e 36 chamam o método **static getAudioInputStream** de **AudioSystem** para obter um **AudioInputStream** para **soundFile**. A classe **AudioSystem** facilita o acesso a muitos dos recursos necessários para reproduzir e manipular arquivos de som. O método **getAudioInputStream** dispara uma **UnsupportedAudioFileException** se o arquivo de som especificado não for um arquivo de áudio ou se ele contiver um formato que não é suportado por Java Sound.

A seguir, o programa precisa fornecer uma linha através da qual os dados de áudio podem ser processados. A linha 52 invoca o método **loadClip** (linhas 57 a 106) para abrir uma linha de **Clip** e carregar o *stream* de áudio para reprodução. A linha 81 invoca o método **static getLine** de **AudioSystem** para obter uma linha de **Clip** para reprodução de áudio. O método **getLine** exige um objeto **Line.Info** como um argumento, para especificar os atributos da linha que o **AudioSystem** deve devolver. A linha deve ser capaz de processar cliques de áudio de todos os formatos de amostra de áudio suportados, de modo que o objeto **DataLine.Info** deve especificar uma linha de dados de **Clip** e um formato de codificação genérica. O intervalo de *buffer* também deve ser especificado para que o programa possa determinar o melhor tamanho de *buffer*. O construtor de **DataLine.Info** recebe quatro argumentos. Os dois primeiros são o formato (do tipo **AudioFormat.Encoding**) para o qual o programa deve converter os dados de áudio e o **AudioFormat** da fonte de áudio. O **AudioFormat** configura o formato suportado pela linha, de acordo com o formato de áudio do *stream*. A linha 63 obtém o **AudioFormat** do **AudioInputStream**, que contém especificações de formato que o sistema subjacente usa para traduzir os dados em sons. As linhas 68 e 69 chamam o método **getTargetFormats** de **AudioSystem** para obter um *array* com os **AudioFormats** suportados. O terceiro argumento do construtor **DataLine.Info**, que especifica o tamanho mínimo do *buffer*, é configurado com o número de *bytes* em cada *frame* do *stream* de áudio. A linha 70 invoca o método **getFrameSize** de **AudioFormat** para obter o tamanho de cada *frame* no *stream* de áudio. O tamanho máximo do *buffer* deve ser equivalente a duas *frames* do *stream* de áudio (linha 71). Usando o objeto **DataLine.Info**, a linha 74 verifica se o sistema de áudio subjacente suporta a linha especificada. Se suportar, a linha 81 obtém

a linha do sistema de áudio. Quando um clipe de áudio começa a ser reproduzido e termina, o programa precisa ser alertado. A linha 84 registra um `LineListener` para os `LineEvents` do `Clip`. Se ocorre um `LineEvent`, o programa chama o método `update` de `LineListener` (linhas 115 a 133) para processá-lo. Os quatro tipos de `LineEvent`, como definidos na classe `LineEvent.Type`, são `OPEN`, `CLOSE`, `START` e `STOP`. Quando o tipo de evento é `LineEvent.Type.STOP` e a variável `replay` é `false`, a linha 120 chama o método `close` de `ClipPlayer` (linhas 142 a 148) para parar a reprodução de áudio e fechar o `Clip`. Todos os recursos de áudio obtidos anteriormente pelo `Clip` são liberados quando a reprodução do áudio pára. Quando o tipo de evento é `LineEvent.Type.STOP` e a variável `replay` é `true`, a linha 131 chama o método `loop` de `Clip` com o parâmetro `Clip.LOOP_CONTINUOUSLY`, fazendo com que o `Clip` seja repetido até que o usuário termine o aplicativo. Invocar o método `stop` da interface `Clip` para a atividade de dados na `Line`. Invocar o método `start` retoma a atividade de dados.

Assim que o programa termina de validar o `Clip`, a linha 87 chama o método `open` de `Clip` com o `AudioInputStream soundStream` como argumento. O `Clip` obtém os recursos do sistema necessários para reprodução de áudio. O método `getLine` de `AudioSystem` e o método `open` de `Clip` disparam `LineUnavailableExceptions` se outro aplicativo está usando o recurso de áudio solicitado. O método `open` de `Clip` também dispara uma `IOException` se o `Clip` não consegue ler o `AudioInputStream` especificado. Quando o programa de teste (Fig. 22.6) chama o método `play` de `ClipPlayer` (linhas 109 a 112), o método `start` de `Clip` inicia a reprodução do áudio.

A classe `ClipPlayerTest` (Fig. 22.6) permite especificar um arquivo de áudio a ser reproduzido clicando no botão **Open Audio Clip**. Quando os usuários clicam no botão, o método `actionPerformed` (linhas 37 a 58) solicita nome e endereço de arquivo de áudio (linha 39) e cria um `ClipPlayer` para o arquivo de áudio especificado (linha 44). A linha 47 invoca o método `openFile` de `ClipPlayer`, que devolve `true` se o `ClipPlayer` consegue abrir o arquivo de áudio. Se for assim, a linha 50 chama o método `play` de `ClipPlayer` para reproduzir o áudio e a linha 53 chama o método `setReplay` de `ClipPlayer` para indicar que o áudio não deve ser repetido continuamente.



Dica de desempenho 22.4

Os arquivos grandes de áudio exigem um tempo longo para carregar, dependendo da velocidade do computador. Uma maneira alternativa de reproduzir é colocar o áudio em um buffer, carregando uma parte dos dados para começar a reprodução e continuando a carregar o restante à medida que o áudio é reproduzido. Isto é semelhante à capacidade de streaming fornecida pelo JMF.

```

1  // Fig. 22.6: ClipPlayerTest.java
2  // Testa arquivo para ClipPlayer
3
4  // Pacotes do núcleo de Java
5  import java.awt.*;
6  import java.awt.event.*;
7  import java.io.*;
8
9  // Pacotes de extensão de Java
10 import javax.swing.*;
11
12 public class ClipPlayerTest extends JFrame {
13
14     // objeto para reproduzir clipes de áudio
15     private ClipPlayer clipPlayer;
16
17     // construtor para ClipPlayerTest
18     public ClipPlayerTest()
19     {
20         super( "Clip Player" );
21     }

```

Fig. 22.6 `ClipPlayerTest` permite especificar o nome e o endereço do áudio a ser reproduzido com `ClipPlayer` (parte 1 de 3).

```

22 // painel contendo botões
23 JPanel buttonPanel = new JPanel();
24 getContentPane().add( buttonPanel );
25
26 // botão de abrir arquivo
27 JButton openFile = new JButton( "Open Audio Clip" );
28 buttonPanel.add( openFile, BorderLayout.CENTER );
29
30 // registra ActionListener para eventos de openFile
31 openFile.addActionListener(
32
33     // classe interna anônima para tratar o ActionEvent de openFile
34     new ActionListener() {
35
36         // tenta abrir e reproduzir um arquivo de clipe de áudio
37         public void actionPerformed((ActionEvent event) )
38         {
39             File mediaFile = getFile();
40
41             if ( mediaFile != null ) {
42
43                 // instancia novo ClipPlayer com mediaFile
44                 clipPlayer = new ClipPlayer( mediaFile );
45
46                 // se o ClipPlayer foi aberto corretamente
47                 if ( clipPlayer.openFile() == true ) {
48
49                     // reproduz o clipe carregado
50                     clipPlayer.play();
51
52                     // sem repetição
53                     clipPlayer.setReplay( false );
54                 }
55
56             } // fim do if mediaFile
57
58         } // fim de actionPerformed
59
60     } // fim de ActionListener
61
62 ); // fim da chamada para addActionListener
63
64 } // fim do construtor
65
66 // obtém o arquivo do computador
67 public File getFile()
68 {
69     JFileChooser fileChooser = new JFileChooser();
70
71     fileChooser.setFileSelectionMode(
72         JFileChooser.FILES_ONLY );
73     int result = fileChooser.showOpenDialog( this );
74
75     if ( result == JFileChooser.CANCEL_OPTION )
76         return null;
77
78     else
79         return fileChooser.getSelectedFile();
80 }

```

Fig. 22.6 ClipPlayerTest permite especificar o nome e o endereço do áudio a ser reproduzido com ClipPlayer (parte 2 de 3).

```

81
82 // executa o aplicativo
83 public static void main( String args[] )
84 {
85     ClipPlayerTest test = new ClipPlayerTest();
86
87     test.setSize( 150, 70 );
88     test.setLocation( 300, 300 );
89     test.setDefaultCloseOperation( EXIT_ON_CLOSE );
90     test.setVisible( true );
91 }
92
93 } // fim da classe ClipPlayerTest

```

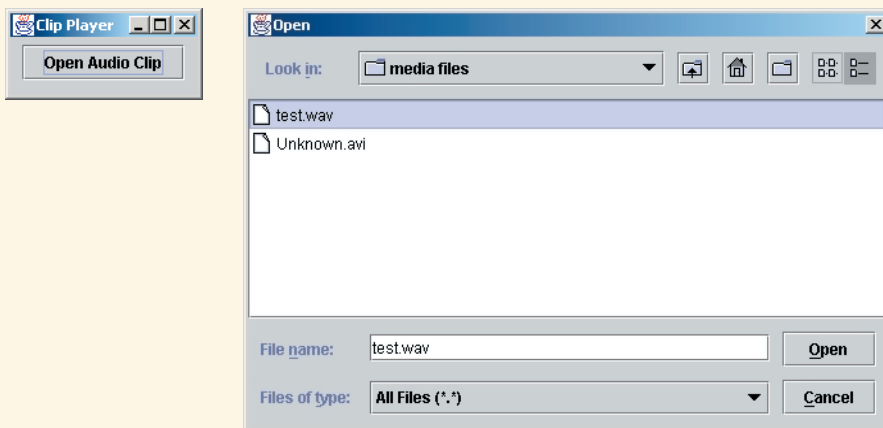


Fig. 22.6 `ClipPlayerTest` permite especificar o nome e o endereço do áudio a ser reproduzido com `ClipPlayer` (parte 3 de 3).

22.7 Musical Instrument Digital Interface (MIDI)

A *Musical Instrument Digital Interface* (MIDI) é o formato-padrão para a música eletrônica. Pode-se criar música MIDI através de um instrumento digital, como um teclado eletrônico, ou através de *software*. A interface MIDI permite criar música digital sintetizada que reproduz a verdadeira música. Os músicos podem compartilhar suas criações musicais com aficionados por música do mundo inteiro. Um *sintetizador* MIDI é um dispositivo que pode produzir sons e música MIDI.

Os programas podem manipular facilmente dados MIDI. Como ocorre com outros tipos de áudio, os dados MIDI têm um formato bem-definido que os reprodutores de MIDI podem interpretar, reproduzir e usar para criar novos dados MIDI. A *Complete Detailed MIDI 1.0 Specification* fornece informações detalhadas sobre os arquivos MIDI. Visite o site oficial sobre MIDI na Web em www.midi.org para obter informações sobre MIDI e sua especificação. Os pacotes para MIDI de Java Sound (`javax.sound.midi` e `javax.sound.midi.spi`) permitem acessar dados MIDI.

A interpretação de dados MIDI varia entre os sintetizadores, de modo que um arquivo pode soar bem diferente quando reproduzido em sintetizadores diferentes daquele no qual foram criados. Os sintetizadores suportam tipos e números de sons instrumentais que variam e quantidades diferentes de sons simultâneos. Usualmente, os sintetizadores baseados em *hardware* são capazes de produzir música sintetizada de qualidade mais alta do que os sintetizadores baseados em *software*.

Muitos sites da Web e jogos usam MIDI para reprodução de música, pois ele permite que os desenvolvedores divirtam os usuários com arquivos enormes de música digitalizada, que não exigem muita memória. Em compara-

ção, arquivos de amostra de áudio podem se tornar bastante grandes. O pacote `javax.sound.midi` permite manipular, reproduzir e sintetizar MIDI. O Java Sound suporta arquivos MIDI com extensões `mid` e `rmf` (*Rich Music Format* ou RMF). O exemplo apresentado nas Seções 22.7.1 a 22.7.4 abrange a síntese, a reprodução, a gravação e o salvamento de MIDI. A classe **MidiDemo** (Fig. 22.10) é a classe principal do aplicativo, que usa as classes **MidiData** (Fig. 22.7), **MidiRecord** (Fig. 22.8) e **MidiSynthesizer** (Fig. 22.9). A classe **MidiSynthesizer** fornece recursos para gerar sons e transmiti-los para outros dispositivos MIDI, como gravadores. A classe **MidiData** trata da reprodução de MIDI, da inicialização de trilhas e das informações sobre eventos. A classe **MidiRecord** fornece recursos para gravação de MIDI. A classe **MidiDemo** reúne as outras classes com uma GUI interativa que inclui um teclado de piano simulado, botões de reproduzir e gravar, e um painel de controle para configurar opções de MIDI. A classe **MidiDemo** também usa a sincronização de eventos MIDI para reproduzir um arquivo MIDI e destacar as teclas apropriadas no piano, imitando alguém tocando no teclado.

Uma parte integrante deste exemplo MIDI é a sua GUI, que permite aos usuários tocar notas musicais em um teclado do piano simulado (ver captura de tela na Fig. 22.10). Quando o mouse passa sobre uma tecla do piano, o programa reproduz a nota correspondente. Nesta seção, referimo-nos a isto como *síntese pelo usuário*. O botão **Play MIDI** na GUI permite selecionar um arquivo MIDI a ser reproduzido. O botão **Record** grava as notas tocadas no piano (síntese pelo usuário). Os usuários podem salvar o MIDI gravado em um arquivo usando o botão **Save MIDI** e reproduzir o MIDI gravado o botão **Playback**. Os usuários podem clicar no botão **Piano Player** para abrir um arquivo MIDI, depois reproduzir aquele arquivo de volta através de um sintetizador. O programa indica a sincronização de nota e das teclas do piano destacando a tecla que corresponde ao número da nota. Esta capacidade de reprodução e sincronização se chama “tocador de piano”. Enquanto o “tocador de piano” está sendo executado, os usuários podem sintetizar as notas adicionais e gravar tanto o material de áudio antigo quanto as novas notas sintetizadas pelo usuário, clicando no botão **Record**. A **JComboBox** no canto superior esquerdo da GUI permite selecionar um instrumento para síntese. Os componentes adicionais da GUI incluem um controle de volume para notas sintetizadas pelo usuário e um controle de tempo para controlar a velocidade do “tocador de piano”.



Dica de teste e depuração 22.1

Para testar as funções de reprodução de arquivos MIDI são necessários uma placa de som e um arquivo de áudio no formato MIDI.

22.7.1 Reprodução de MIDI

Esta seção discute como reproduzir arquivos MIDI e como acessar e interpretar o conteúdo de arquivos MIDI. A classe **MidiData** (Fig. 22.7) contém os métodos que carregam um arquivo MIDI para reprodução. A classe também fornece as informações sobre as trilhas MIDI exigidas pelo recurso de “tocador de piano”. Usa-se um seqüenciador MIDI para reproduzir e manipular os dados de áudio. Frequentemente, os dados MIDI são conhecidos como seqüência, porque os dados musicais em um arquivo MIDI são compostos por uma seqüência de eventos. As etapas executadas na reprodução de MIDI são acessar um seqüenciador, carregar uma seqüência ou arquivo MIDI naquele seqüenciador e iniciar o seqüenciador.

```

1 // Fig. 22.7: MidiData.java
2 // Contém informações sobre seqüências de MIDI
3 // com método de acesso e métodos de reprodução de MIDI
4
5 // Pacotes do núcleo de Java
6 import java.io.*;
7
8 // Pacotes de extensão de Java
9 import javax.sound.midi.*;
10
11 public class MidiData {
12
13     // dados das trilhas MIDI

```

Fig. 22.7 **MidiData** carrega arquivos MIDI para reprodução (parte 1 de 4).

```

14     private Track track;
15
16     // reprodutor para seqüências MIDI
17     private Sequencer sequencer;
18
19     // seqüência MIDI
20     private Sequence sequence;
21
22     // eventos MIDI contendo tempo e MidiMessages
23     private MidiEvent currentEvent, nextEvent;
24
25     // mensagem MIDI usualmente contendo mensagens de som
26     private ShortMessage noteMessage;
27
28     // mensagens MIDI short, meta ou sysex
29     private MidiMessage message;
30
31     // índice do evento MIDI na trilha, comando em uma mensagem MIDI
32     private int eventIndex = 0, command;
33
34     // método para reproduzir seqüência MIDI através de um seqüenciador
35     public void play()
36     {
37         // inicia seqüenciador-padrão
38         try {
39
40             // obtém seqüenciador de MidiSystem
41             sequencer = MidiSystem.getSequencer();
42
43             // abre recursos do seqüenciador
44             sequencer.open();
45
46             // carrega MIDI no seqüenciador
47             sequencer.setSequence( sequence );
48
49             // reproduz seqüência
50             sequencer.start();
51         }
52
53         // erro de disponibilidade de recurso MIDI
54         catch ( MidiUnavailableException noMidiException ) {
55             noMidiException.printStackTrace();
56         }
57
58         // encontrado arquivo MIDI corrompido ou inválido
59         catch ( InvalidMidiDataException badMidiException ) {
60             badMidiException.printStackTrace();
61         }
62     }
63
64 } // fim do método play
65
66 // método que devolve tempo/resolução ajustados do MIDI
67 public int getResolution()
68 {
69     return 500 / sequence.getResolution();
70 }
71
72 // obtém MIDI e prepara trilha em MIDI para ser acessada
73 public boolean initialize( File file )

```

Fig. 22.7 MidiData carrega arquivos MIDI para reprodução (parte 2 de 4).

```

74     {
75         // coloca MIDI válido do arquivo na sequência
76         try {
77             sequence = MidiSystem.getSequence( file );
78         }
79
80         // arquivo MIDI ilegível ou MIDI não-suportado
81         catch ( InvalidMidiDataException badMIDI ) {
82             badMIDI.printStackTrace();
83             return false;
84         }
85
86         // erro de E/S gerado durante a leitura do arquivo
87         catch ( IOException ioException ) {
88             ioException.printStackTrace();
89             return false;
90         }
91
92         return true;
93     } // fim do método initialize
94
95     // prepara trilha mais longa para ser lida e obtém primeiro evento MIDI
96     public boolean initializeTrack()
97     {
98         // obtém todas as trilhas da sequência
99         Track tracks[] = sequence.getTracks();
100
101         if ( tracks.length == 0 ) {
102             System.err.println( "No tracks in MIDI sequence!" );
103
104             return false;
105         }
106
107         track = tracks[ 0 ];
108
109         // encontra trilha mais longa
110         for ( int i = 0; i < tracks.length; i++ )
111
112             if ( tracks[ i ].size() > track.size() )
113                 track = tracks[ i ];
114
115         // configura evento MIDI corrente para primeiro evento na trilha
116         currentEvent = track.get( eventIndex );
117
118         // obtém mensagem MIDI do evento
119         message = currentEvent.getMessage();
120
121         // inicialização de trilha bem sucedida
122         return true;
123     } // fim do método initializeTrack
124
125     // prossegue para próximo evento na trilha
126     public void goNextEvent()
127     {
128         eventIndex++;
129         currentEvent = track.get( eventIndex );
130         message = currentEvent.getMessage();
131     }
132
133 }

```

Fig. 22.7 MidiData carrega arquivos MIDI para reprodução (parte 3 de 4).

```

134
135 // obtém intervalo de tempo entre eventos
136 public int getEventDelay()
137 {
138     // o primeiro intervalo de tempo do evento é a sua duração
139     if ( eventIndex == 0 )
140         return ( int ) currentEvent.getTick();
141
142     // diferença de tempo entre evento corrente e o próximo
143     return ( int ) ( track.get( eventIndex + 1 ).getTick() -
144         currentEvent.getTick() );
145 }
146
147 // retorna se a trilha terminou
148 public boolean isTrackEnd()
149 {
150     // se eventIndex é menor do que o número de eventos da trilha
151     if ( eventIndex + 1 < track.size() )
152         return false;
153
154     return true;
155 }
156
157 // obtém comando ShortMessage corrente do evento
158 public int getEventCommand()
159 {
160     if ( message instanceof ShortMessage ) {
161
162         // obtém MidiMessage para fins de acesso
163         noteMessage = ( ShortMessage ) message;
164         return noteMessage.getCommand();
165     }
166
167     return -1;
168 }
169
170 // obtém o número da nota do evento corrente
171 public int getNote()
172 {
173     if ( noteMessage != null )
174         return noteMessage.getData1();
175
176     return -1;
177 }
178
179 // obtém o volume do evento corrente
180 public int getVolume()
181 {
182     return noteMessage.getData2();
183 }
184
185 } // fim da classe MidiData

```

Fig. 22.7 MidiData carrega arquivos MIDI para reprodução (parte 4 de 4).

Para reproduzir um arquivo MIDI com uma sequência, o programa precisa obter a sequência MIDI e verificar os aspectos de compatibilidade. O método `initialize` de `MidiData` (linhas 73 a 94) obtém uma *Sequence* de dados MIDI de um arquivo com o método `getSequence` de `MidiSystem` (linha 77). A *Sequence* contém trilhas MIDI, as quais, por sua vez, contêm eventos MIDI. Cada evento encapsula uma mensagem MIDI de instruções para os dispositivos MIDI. As trilhas individuais de uma sequência MIDI são análogas às trilhas em um CD.

Entretanto, enquanto as trilhas de CD são reproduzidas na ordem, as trilhas MIDI são reproduzidas em paralelo. A trilha MIDI é uma sequência de dados gravada. As MIDIs normalmente contêm múltiplas trilhas. O método `getSequence` também pode obter uma sequência MIDI de um URL ou um `InputStream`. O método `getSequence` dispara uma `InvalidMidiDataException` se o sistema MIDI detecta um arquivo MIDI incompatível.



Dica de portabilidade 22.2

Devido à incompatibilidade entre os reconhecedores de arquivos em sistemas operacionais diferentes, os seqüenciadores podem não conseguir reproduzir arquivos RMF.

Após obter uma sequência MIDI válida, o programa precisa obter um seqüenciador e carregar a sequência no seqüenciador. O método `play` (linhas 35 a 64) na classe `MidiData` chama o método `getSequencer` de `MidiSystem` (linha 41) para obter um `Sequencer` para reproduzir a `Sequence`. A interface `Sequencer`, que estende a interface `MidiDevice` (a super-interface para todos os dispositivos MIDI), fornece o dispositivo seqüenciador padrão para reproduzir os dados MIDI. Se um outro programa está usando o mesmo objeto `Sequencer`, o método `getSequencer` dispara uma `MidiUnavailableException`. A linha 44 chama o método `open` de `Sequencer` para se preparar para reproduzir uma `Sequence`. O método `setSequence` de `Sequencer` (linha 47) carrega uma `Sequence` MIDI no `Sequencer` e dispara uma `InvalidMidiException` se o `Sequencer` detectar uma sequência MIDI irreconhecível. A linha 50 começa a reproduzir a sequência MIDI chamando o método `start` do `Sequencer`.

Além dos métodos de reprodução de MIDI, a classe `MidiData` também fornece métodos que permitem a um programa acessar os eventos e as mensagens de uma sequência MIDI. Como veremos, a classe `MidiDemo` (Fig. 22.10) usa a classe `MidiData` para acessar os dados em um arquivo MIDI para sincronizar o destaque das teclas do piano. Os eventos MIDI são armazenados nas trilhas de MIDI, que são instâncias da classe `Track` (pacote `javax.sound.midi`). Os eventos MIDI em trilhas MIDI são representados pela classe `MidiEvent` (pacote `javax.sound.midi`). Cada evento MIDI contém uma instrução e o tempo em que ela deve ocorrer. Os eventos individuais em uma trilha contêm mensagens do tipo `MidiMessage` que especificam as instruções MIDI para um `MidiDevice`. Existem três tipos de mensagens MIDI – `ShortMessage`, `SysexMessage` e `MetaMessage`. `ShortMessages` são instruções musicais explícitas, como as notas específicas a tocar e mudanças de frequência. As outras duas mensagens, menos usadas, são as `SysexMessages`, mensagens de uso exclusivo do sistema para dispositivos MIDI, e `MetaMessages`, que podem indicar para um dispositivo MIDI que o MIDI atingiu o fim de uma trilha. Esta seção trata exclusivamente de `ShortMessages` que reproduzem notas específicas.

A seguir, o programa precisa obter as trilhas e ler seus eventos. O método `initializeTrack` de `MidiData` (linhas 97 a 125) invoca o método `getTracks` da `Sequence` (linha 100) para obter todas as trilhas na sequência MIDI. As linhas 108 a 114 determinam a trilha mais longa no MIDI e a configuram como aquela a ser reproduzida. A linha 117 obtém o primeiro evento MIDI na `Track` invocando seu método `get` com o índice do evento na trilha como o parâmetro. Neste ponto, `eventIndex` é ajustado para 0 (linha 32). A linha 120 obtém a mensagem MIDI do evento MIDI usando o método `getMessage` da classe `MidiEvent`. Para ajudar um programa a passar por cada evento nas trilhas, o programa pode chamar o método `goNextEvent` de `MidiData` (linhas 128 a 133) para carregar o próximo evento e a mensagem. O método `goNextEvent` incrementa `eventIndex` na `Track` MIDI carregada e encontra a próxima `MidiMessage` do evento. Além de ler os eventos, o programa também precisa determinar quanto tempo cada evento dura e o espaçamento entre eventos. O método `getEventDelay` (linhas 136 a 145) devolve a duração de um `MidiEvent` como a diferença de tempo entre dois eventos na sequência MIDI (linhas 143 e 144). O método `getTick` de `MidiEvent` fornece o tempo específico em que o evento ocorre (também chamado de *time stamp*). As linhas 139 e 140 devolvem a *time stamp* do primeiro `MidiEvent` como a duração do evento.

A classe `MidiData` fornece outros métodos para devolver os comandos, os números das notas e o volume de `ShortMessages` relacionadas com notas. O método `getEventCommand` (linhas 158 a 168) determina o número do comando que representa a instrução de comando. A linha 160 do método `getEventCommand` indica se a `MidiMessage` atualmente carregada é uma `ShortMessage`. Se for, a linha 163 atribui a `ShortMessage` ao objeto `noteMessage` e a linha 164 devolve o *byte* de estado de comando da `ShortMessage` invocando o método `getCommand` de `ShortMessage`. O método `getEventCommand` devolve -1 se o evento não contiver uma `ShortMessage`. O método `getNote` de `MidiData` (linhas 171 a 177) invoca o método `getData1` de `ShortMessage` (linha 174) para devolver o número da nota. O método `getVolume` (linhas 180 a 183) invoca o método `getData2` de `ShortMessage` para devolver o volume. A classe `MidiData` também fornece uma indicação do

fim de uma trilha no método `isTrackEnd` (linhas 148 a 155), que determina se o índice de evento ultrapassou o número de eventos na trilha (linha 151).

22.7.2 Gravação de MIDI

O programa pode gravar MIDI usando um seqüenciador. A classe `MidiRecord` (fig. 22.8) trata das funções de gravação desta demonstração de MIDI usando um objeto que implementa a interface `Sequencer` como gravador MIDI. Desde que os dispositivos MIDI estejam configurados corretamente, a interface `Sequencer` fornece métodos simples para gravação. A classe `MidiRecord` tem um construtor (linhas 29 a 32) que recebe como argumento um objeto que implementa a interface `Transmitter`. O `Transmitter` envia mensagens MIDI para um dispositivo MIDI que implementa a interface `Receiver`. Pense nos `Transmitters` e `Receivers` como portas de saída e de entrada, respectivamente, para dispositivos MIDI.

```

1  // Fig. 22.8: MidiRecord.java
2  // Permite gravação e reprodução
3  // de MIDI sintetizado
4
5  // Pacotes do núcleo de Java
6  import java.io.*;
7
8  // Pacotes de extensão de Java
9  import javax.sound.midi.*;
10
11 public class MidiRecord {
12
13     // trilha MIDI
14     private Track track;
15
16     // seqüenciador MIDI para reproduzir e acessar música
17     private Sequencer sequencer;
18
19     // seqüência MIDI
20     private Sequence sequence;
21
22     // receptor de eventos MIDI
23     private Receiver receiver;
24
25     // transmissor para transmitir mensagens MIDI
26     private Transmitter transmitter;
27
28     // construtor para MidiRecord
29     public MidiRecord( Transmitter transmit )
30     {
31         transmitter = transmit;
32     }
33
34     // inicializa seqüenciador de gravação, configura seqüência de gravação
35     public boolean initialize()
36     {
37         // cria seqüência MIDI vazia e configura fiação do seqüenciador
38         try {
39
40             // cria seqüência baseada em tempo de 10 pulsos por compasso
41             sequence = new Sequence( Sequence.PPQ, 10 );
42
43             // obtém seqüência e a abre
44             sequencer = MidiSystem.getSequencer();

```

Fig. 22.8 `MidiRecord` permite a um programa gravar uma seqüência MIDI (parte 1 de 3).

```

45         sequencer.open();
46
47         // obtém receptor do seqüenciador
48         receiver = sequencer.getReceiver();
49
50         if ( receiver == null ) {
51             System.err.println(
52                 "Receiver unavailable for sequencer" );
53             return false;
54         }
55
56         // configura receptor para o transmissor enviar MidiMessages
57         transmitter.setReceiver( receiver );
58
59         makeTrack();
60     }
61
62     // especificação de divisão de temporização inválida para nova seqüência
63     catch ( InvalidMidiDataException invalidMidiException ) {
64         invalidMidiException.printStackTrace();
65         return false;
66     }
67
68     // seqüenciador ou receptor indisponível
69     catch ( MidiUnavailableException noMidiException ) {
70         noMidiException.printStackTrace();
71         return false;
72     }
73
74     // inicialização do gravador MIDI bem-sucedida
75     return true;
76 } // fim do método initialize
77
78 // cria nova trilha vazia para seqüência
79 public void makeTrack()
80 {
81     // se existe trilha anterior, primeiro a apaga
82     if ( track != null )
83         sequence.deleteTrack( track );
84
85     // cria trilha na seqüência
86     track = sequence.createTrack();
87 }
88
89 // inicia reprodução da seqüência carregada
90 public void play()
91 {
92     sequencer.start();
93 }
94
95 // começa gravação para a seqüência
96 public void startRecord()
97 {
98     // carrega seqüência no gravador e começa a gravar
99     try {
100         sequencer.setSequence( sequence );
101
102         // configura a trilha como habilitada para gravação e canal padrão
103

```

Fig. 22.8 MidiRecord permite a um programa gravar uma seqüência MIDI (parte 2 de 3).

```

104         sequencer.recordEnable( track, 0 );
105
106         sequencer.startRecording();
107     }
108
109     // sequência contém dados MIDI inválidos
110     catch ( InvalidMidiDataException badMidiException ) {
111         badMidiException.printStackTrace();
112     }
113
114 } // fim do método startRecord
115
116 // pára gravação MIDI
117 public void stopRecord()
118 {
119     sequencer.stopRecording();
120 }
121
122 // salva sequência MIDI no arquivo
123 public void saveSequence( File file )
124 {
125     // obtém todos os tipos de arquivos suportados por MIDI
126     int[] fileTypes = MidiSystem.getMidiFileTypes( sequence );
127
128     if ( fileTypes.length == 0 ) {
129         System.err.println( "No supported MIDI file format!" );
130         return;
131     }
132
133     // escreve sequência gravada no arquivo MIDI
134     try {
135         MidiSystem.write( sequence, fileTypes[ 0 ], file );
136     }
137
138     // erro durante gravação no arquivo
139     catch ( IOException ioException ) {
140         ioException.printStackTrace();
141     }
142
143 } // fim do método saveSequence
144
145 } // fim da classe MidiRecord

```

Fig. 22.8 **MidiRecord** permite a um programa gravar uma sequência MIDI (parte 3 de 3).

A primeira etapa da gravação de dados MIDI é semelhante à do mecanismo de reprodução na classe **MidiData**. Além de obter uma sequência vazia e um sequenciador, o programa de gravação de MIDI precisa conectar os transmissores e receptores. Depois de “ligar os fios” do receptor do sequenciador com a “porta de entrada”, o gravador carrega a sequência vazia no sequenciador para começar a gravar uma nova trilha na sequência. A discussão a seguir abrange estas etapas.

O método **initialize** (linhas 35 a 77) da classe **MidiRecord** configura o sequenciador para gravação. A linha 41 do método **initialize** instancia uma sequência vazia. **MidiRecord** irá gravar dados na sequência vazia assim que o transmissor seja conectado ao receptor. A linha 48 obtém o **receiver** do sequenciador de gravação e a linha 57 especifica que **transmitter** vai enviar suas mensagens para **receiver**.

As mensagens MIDI precisam ser colocadas em uma trilha, de modo que o método **initialize** invoca o método **makeTrack** (linhas 80 a 88) para apagar a **track** anterior existente (linha 84) e para criar uma **Track** vazia (linha 87). O método **makeTrack** também pode ser chamado de uma classe externa para gravar uma nova sequência sem instanciar novos sequenciadores e uma nova sequência.

Depois de configurar um seqüenciador e uma seqüência vazia, chamar o método **startRecord** de **MidiRecord** (linhas 97 a 115) inicia o processo de gravação. A linha 101 carrega uma seqüência vazia no seqüenciador. O método **recordEnable** de **Sequencer** é chamado e lhe são passados o objeto **track** e um número de canal como argumentos (linha 104), o que habilita a gravação naquela **track**. A linha 106 invoca o método **startRecording** do **Sequencer** para iniciar a gravação de eventos MIDI enviados do transmissor. O método **stopRecording** do **Sequencer** faz parar a gravação e é chamado no método **stopRecord** de **MidiRecord** (linhas 118 a 121).

A classe **MidiRecord** também pode suportar o salvamento de uma seqüência gravada em um arquivo MIDI com seu método **saveSequence** (linhas 124 a 144). Embora a maioria das seqüências MIDI possam suportar *arquivos MIDI tipo 0* (o tipo mais comum de arquivo MIDI), deve-se verificar a seqüência para saber que outros tipos de arquivos são suportados. A linha 127 obtém um *array* de tipos de arquivos MIDI suportados pelo sistema para gravar uma seqüência em um arquivo. Os tipos de arquivos MIDI são representados por valores inteiros 0, 1 ou 2. Usando o primeiro tipo de arquivo suportado, o **MidiSystem** grava a seqüência em um **File** especificado (linha 136) passado para o método **saveSequence** como argumento. O método **play** de **MidiRecord** (linhas 91 a 94) habilita o programa a reproduzir a seqüência recém-gravada.

22.7.3 Síntese de MIDI

Este programa **MidiDemo** fornece um piano interativo que gera notas de acordo com as teclas pressionadas pelo usuário. A classe **MidiSynthesizer** (Fig. 22.9) gera estas notas diretamente e as envia para outro dispositivo. Especificamente, ela envia as notas para um **receiver** de um seqüenciador através de um **transmitter** para gravar a seqüência MIDI. A classe **MidiSynthesizer** usa um objeto que implementa a interface **Synthesizer** (uma subinterface de **MidiDevice**) para acessar a geração de som, instrumentos, recursos de canal e bancos de som padrão do sintetizador. O **SoundBank** é o contêiner para diversos **Instruments**, que instrui o computador sobre como fazer o som para uma nota específica. Notas diferentes feitas por vários instrumentos são tocadas através de um **MidiChannel** em trilhas diferentes simultaneamente para produzir melodias sinfônicas.

```

1  // Fig. 22.9: MidiSynthesizer.java
2  // Acessando recursos do sintetizador
3
4  // Pacotes de extensão de Java
5  import javax.sound.midi.*;
6
7  public class MidiSynthesizer {
8
9      // sintetizador principal acessa recursos
10     private Synthesizer synthesizer;
11
12     // instrumentos disponíveis para uso na síntese
13     private Instrument instruments[];
14
15     // canais através dos quais as notas soam
16     private MidiChannel channels[];
17     private MidiChannel channel; // canal atual
18
19     // transmissor para transmitir mensagens
20     private Transmitter transmitter;
21
22     // lado receptor de mensagens
23     private Receiver receiver;
24
25     // mensagem curta contendo comandos de som, nota, volume
26     private ShortMessage message;
27

```

Fig. 22.9 **MidiSynthesizer** pode gerar notas e enviá-las para um outro dispositivo MIDI (parte 1 de 3).

```

28 // construtor para MidiSynthesizer
29 public MidiSynthesizer()
30 {
31     // abre sintetizador, configura receptor,
32     // obtém canais e instrumentos
33     try {
34         synthesizer = MidiSystem.getSynthesizer();
35
36         if ( synthesizer != null ) {
37
38             synthesizer.open();
39
40             // obtém transmissor do sintetizador
41             transmitter = synthesizer.getTransmitter();
42
43             if ( transmitter == null )
44                 System.err.println( "Transmitter unavailable" );
45
46             // obtém receptor do sintetizador
47             receiver = synthesizer.getReceiver();
48
49             if ( receiver == null )
50                 System.out.println( "Receiver unavailable" );
51
52             // obtém todos os instrumentos disponíveis no
53             // banco de sons ou sintetizador padrão
54             instruments = synthesizer.getAvailableInstruments();
55
56             // obtém todos os 16 canais do sintetizador
57             channels = synthesizer.getChannels();
58
59             // atribui primeiro canal como canal padrão
60             channel = channels[ 0 ];
61         }
62     }
63     else
64         System.err.println( "No Synthesizer" );
65 }
66
67 // sintetizador, receptor ou transmissor não-disponível
68 catch ( MidiUnavailableException noMidiException ) {
69     noMidiException.printStackTrace();
70 }
71
72 } // fim do construtor
73
74 // devolve instrumentos disponíveis
75 public Instrument[] getInstruments()
76 {
77     return instruments;
78 }
79
80 // devolve transmissor do sintetizador
81 public Transmitter getTransmitter()
82 {
83     return transmitter;
84 }
85
86 // ativa nota de som através do canal

```

Fig. 22.9 MidiSynthesizer pode gerar notas e enviá-las para um outro dispositivo MIDI (parte 2 de 3).

```

87     public void midiNoteOn( int note, int volume )
88     {
89         channel.noteOn( note, volume );
90     }
91
92     // desliga nota de som através do canal
93     public void midiNoteOff( int note )
94     {
95         channel.noteOff( note );
96     }
97
98     // muda para o instrumento selecionado
99     public void changeInstrument( int index )
100    {
101        Patch patch = instruments[ index ].getPatch();
102
103        channel.programChange( patch.getBank(),
104                               patch.getProgram() );
105    }
106
107    // envia mensagens MIDI de praxe através do transmissor
108    public void sendMessage( int command, int note, int volume )
109    {
110        // envia uma ShortMessage MIDI usando os parâmetros deste método
111        try {
112            message = new ShortMessage();
113
114            // configura nova mensagem de comando (NOTE_ON, NOTE_OFF),
115            // número da nota, volume
116            message.setMessage( command, note, volume );
117
118            // envia mensagem através do receptor
119            receiver.send( message, -1 );
120        }
121
122        // valores de mensagem inválidos configurados
123        catch ( InvalidMidiDataException badMidiException ) {
124            badMidiException.printStackTrace();
125        }
126    } // fim do método sendMessage
127
128 } // fim da classe MidiSynthesizer

```

Fig. 22.9 `MidiSynthesizer` pode gerar notas e enviá-las para um outro dispositivo MIDI (parte 3 de 3).

O construtor de `MidiSynthesizer` (linhas 29 a 72) adquire o sintetizador e inicializa recursos relacionados. A linha 34 obtém um objeto `Synthesizer` do `MidiSystem` e a linha 38 abre o `Synthesizer`. Para permitir que sons sejam reproduzidos e gravados ao mesmo tempo, as linhas 41 a 47 obtêm o `Transmitter` e o `Receiver` do `Synthesizer`. Quando uma mensagem MIDI é enviada para o `receiver` do `synthesizer`, o `synthesizer` executa a instrução da mensagem, gerando notas, e o `transmitter` envia aquela mensagem para `Receivers` designados de outros dispositivos MIDI.



Erro comum de programação 22.3

Ocorre uma `MidiUnavailableException` quando o programa tenta adquirir recursos de `MidiDevice` indisponíveis, como sintetizadores e transmissores.

As mensagens MIDI são enviadas de `MidiDemo` para o `MidiSynthesizer` após se pressionar uma tecla do piano ou um `MidiEvent` na trilha previamente carregada de `MidiData`. Pode-se gerar uma nota acessando os

channels do **synthesizer** diretamente. Para simplificar, **MidiSynthesizer** usa somente o primeiro canal (de 16 possíveis) para emitir notas. A linha 57 invoca o método **getChannels** de **Synthesizer** para obter todos os 16 canais de **synthesizer** e a linha 60 configura o canal *default* para o primeiro canal. O **MidiChannel** emite uma nota chamando seu método **noteOn** com o número da nota (0 a 127) e um número de volume como argumentos. O método **noteOff** de **MidiChannel** desliga uma nota apenas com o número da nota como argumento. O **MidiSynthesizer** acessa estes métodos de **MidiChannel** através dos métodos **midiNoteOn** (linhas 87 a 90) e **midiNoteOff** (linhas 93 a 96), respectivamente.

O sintetizador pode usar seus instrumentos *default* para emitir notas. A linha 54 obtém o instrumento *default* disponível através do sintetizador ou através de um banco de sons *default* invocando o método **getAvailableInstruments** de **Synthesizer**. Banco de sons normalmente tem 128 instrumentos. Os instrumentos em uso podem ser trocados com o método **changeInstrument** de **MidiSynthesizer** (linhas 99 a 105). As linhas 103 e 104 invocam o método **programChange** do **MidiChannel** para carregar o programa do instrumento desejado, com o número do banco e do programa obtidos de **patch** (linha 104) como parâmetros. O **Patch** é o endereço de um instrumento carregado.



Dica de desempenho 22.5

Um programa pode importar mais instrumentos carregando um banco de sons personalizado através do método **loadAllInstruments** de **Synthesizer** com um objeto **SoundBank**.

Enviando **MidiMessages** para o **Receiver** de um **Synthesizer**, o programa pode invocar o sintetizador para emitir notas sem usar seus canais. Enviar **MidiMessages** para um **Receiver** de um **MidiDevice** também permite aos **Transmitters** do dispositivo enviar estas mensagens para o **Receiver** de um outro **MidiDevice**.

No método **sendMessage** de **MidiSynthesizer** (linhas 108 a 127), as linhas 112 a 116 criam uma nova **ShortMessage** a partir dos parâmetros do método **sendMessage** e enviam a mensagem para o receptor do sintetizador (linha 119). A linha 116 do método **sendMessage** invoca o método **setMessage** de **ShortMessage** para configurar o conteúdo das instruções da mensagem usando três argumentos **int**: um comando, a nota a ser tocada e o volume da nota. O método **setMessage** dispara uma **InvalidMidiDataException** se os valores de comando e os parâmetros designados forem inválidos.

Ao se criar uma nova **ShortMessage** com o método **setMessage**, o significado do segundo e terceiro argumentos variam dependendo do comando. O comando **ShortMessage.NOTE_ON** designa o segundo parâmetro como o número da nota e o terceiro argumento como a velocidade (i.e., volume) da nota. O comando **ShortMessage.PROGRAM_CHANGE** designa o segundo argumento como o programa do instrumento a usar e ignora o terceiro argumento.

A linha 119 envia a **ShortMessage** criada para o **receiver** do **synthesizer** chamando o método **send** de **Receiver** com a **MidiMessage** e um *time stamp* como argumentos. O **MidiSynthesizer** não se envolve com a complexidade da temporização na síntese de MIDI. O receptor envia um valor **-1** para o parâmetro *time stamp* para indicar que o *time stamp* deve ser ignorado. O gravador de sequência na classe **MidiRecord** toma conta dos aspectos de temporização quando ele recebe as mensagens.

Até este ponto, discutimos as ferramentas necessárias para criar nosso piano MIDI. Resumidamente, a classe **MidiDemo** (Fig. 22.10) utiliza a classe **MidiSynthesizer** para gerar sons e acessar canais e instrumentos. **MidiDemo** utiliza **MidiData** para reproduzir arquivos MIDI e acessar as informações de trilhas de MIDI. **MidiRecord** fornece a função de gravação para **MidiDemo**, que recebe mensagens de **MidiSynthesizer**.

22.7.4 A classe **MidiDemo**

Apresentamos agora a classe **MidiDemo** (Fig. 22.10), que fornece a GUI para nosso piano, e outros componentes GUI para controlar os recursos deste exemplo.

Usando um laço **for**, o método utilitário **makeKeys** (linhas 86 a 155) na classe **MidiDemo** cria 64 botões que representam 64 teclas diferentes de piano. Sempre que o mouse passa sobre uma tecla, o programa emite a nota designada. O método **makeKeys** organiza as teclas no fundo da *frame* usando o método **setBounds** de cada botão (linha 106) para indicar a posição e o tamanho dos botões. O programa organiza os botões horizontalmente de acordo com seu índice no *array* **noteButton**.

```

1  // Fig. 22.10: MidiDemo.java
2  // Simula um teclado musical para tocar diversos instrumentos,
3  // permitindo também gravação, reprodução de arquivos MIDI
4  // e simulação de reprodução MIDI com o teclado
5
6  // Pacotes do núcleo de Java
7  import java.awt.*;
8  import java.awt.event.*;
9  import java.io.*;
10
11 // Pacotes de extensão de Java
12 import javax.swing.*;
13 import javax.swing.event.*;
14 import javax.sound.midi.*;
15
16 public class MidiDemo extends JFrame {
17
18     // gravando dados MIDI
19     private MidiRecord midiRecord;
20
21     // funções de sintetização de MIDI
22     private MidiSynthesizer midiSynthesizer;
23
24     // dados MIDI em arquivo MIDI
25     private MidiData midiData;
26
27     // timer para simular MIDI no piano
28     private Timer pianoTimer;
29
30     // teclas do piano
31     private JButton noteButton[];
32
33     // controles deslizantes de tempo e volume
34     private JSlider volumeSlider, resolutionSlider;
35
36     // contêineres e painel contendo GUI
37     private Container container;
38     private JPanel controlPanel, buttonPanel;
39
40     // seletor de instrumento e GUI de botões
41     private JComboBox instrumentBox;
42     private JButton playButton, recordButton,
43         saveButton, pianoPlayerButton, listenButton;
44
45     // tempo, última tecla do piano invocada, volume de MIDI
46     private int resolution, lastKeyOn = -1, midiVolume = 40;
47
48     // valor booleano que indica se o programa está no modo de gravação
49     private boolean recording = false;
50
51     // número da primeira nota da primeira tecla do piano, número máximo de teclas
52     private static int FIRST_NOTE = 32, MAX_KEYS = 64;
53
54     // construtor para MidiDemo
55     public MidiDemo()
56     {
57         super( "MIDI Demo" );
58     }

```

Fig. 22.10 MidiDemo fornece a GUI que permite que os usuários interajam com o aplicativo (parte 1 de 13).

```

59     container = getContentPane();
60     container.setLayout( new BorderLayout() );
61
62     // sintetizador precisa ser instanciado para habilitar síntese
63     midiSynthesizer = new MidiSynthesizer();
64
65     // cria as teclas do piano
66     makeKeys();
67
68     // adiciona painel de controle à frame
69     controlPanel = new JPanel( new BorderLayout() );
70     container.add( controlPanel, BorderLayout.NORTH );
71
72     makeConfigureControls();
73
74     // adiciona painel de botões à frame
75     buttonPanel = new JPanel( new GridLayout( 5, 1 ) );
76     controlPanel.add( buttonPanel, BorderLayout.EAST );
77
78     // cria a GUI
79     makePlaySaveButtons();
80     makeRecordButton();
81     makePianoPlayerButton();
82
83 } // fim do construtor
84
85 // método utilitário criando teclas do piano
86 private void makeKeys()
87 {
88     // painel contendo as teclas
89     JPanel keyPanel = new JPanel( null );
90     container.add( keyPanel, BorderLayout.CENTER );
91
92     // teclas do piano
93     noteButton = new JButton[ MAX_KEYS ];
94
95     // adiciona botões MAX_KEYS e as notas que eles emitem
96     for ( int i = 0; i < MAX_KEYS; i++ ) {
97
98         final int note = i;
99
100        noteButton[ i ] = new JButton();
101
102        // configurando as teclas brancas
103        noteButton[ i ].setBackground( Color.white );
104
105        // configurando o espaçamento correto para os botões
106        noteButton[ i ].setBounds( ( i * 11 ), 1, 11, 40 );
107        keyPanel.add( noteButton[ i ] );
108
109        // registra um ouvinte de mouse para eventos do mouse
110        noteButton[ i ].addMouseListener(
111
112            // classe interna anônima para tratar eventos do mouse
113            new MouseAdapter() {
114
115                // invoca nota da tecla quando o mouse toca na tecla
116                public void mouseEntered( MouseEvent mouseEvent )
117                {

```

Fig. 22.10 MidiDemo fornece a GUI que permite que os usuários interajam com o aplicativo (parte 2 de 13).

```

118         // se estiver gravando, envia mensagem para o receptor
119         if ( recording )
120             midiSynthesizer.sendMessage(
121                 ShortMessage.NOTE_ON,
122                 note + FIRST_NOTE, midiVolume );
123
124         // senão, só emite a nota
125         else
126             midiSynthesizer.midiNoteOn(
127                 note + FIRST_NOTE, midiVolume );
128
129         // muda a cor da tecla para azul
130         noteButton[ note ].setBackground(
131             Color.blue );
132     }
133
134     // desliga a nota da tecla quando o mouse sai da tecla
135     public void mouseExited( MouseEvent mouseEvent )
136     {
137         if ( recording )
138             midiSynthesizer.sendMessage(
139                 ShortMessage.NOTE_OFF,
140                 note + FIRST_NOTE, midiVolume );
141         else
142             midiSynthesizer.midiNoteOff(
143                 note + FIRST_NOTE );
144
145         noteButton[ note ].setBackground(
146             Color.white );
147     }
148
149     } // fim de MouseAdapter
150
151     ); // fim da chamada para addMouseListener
152
153 } // fim do laço for
154
155 } // fim do método makeKeys
156
157 // ajusta controles de configuração
158 private void makeConfigureControls()
159 {
160     JPanel configurePanel =
161         new JPanel( new GridLayout( 5, 1 ) );
162
163     controlPanel.add( configurePanel, BorderLayout.WEST );
164
165     instrumentBox = new JComboBox(
166         midiSynthesizer.getInstruments() );
167
168     configurePanel.add( instrumentBox );
169
170     // registra um ActionListener para eventos de instrumentBox
171     instrumentBox.addActionListener(
172
173         // classe interna anônima para tratar seletor de instrumentos
174         new ActionListener() {
175
176             // troca o programa de instrumento atual

```

Fig. 22.10 MidiDemo fornece a GUI que permite que os usuários interajam com o aplicativo (parte 3 de 13).

```

177         public void actionPerformed((ActionEvent event) )
178         {
179             // troca instrumento no sintetizador
180             midiSynthesizer.changeInstrument(
181                 instrumentBox.getSelectedIndex() );
182         }
183
184     } // fim de ActionListener
185
186 ); // fim da chamada para o método addActionListener
187
188 JLabel volumeLabel = new JLabel( "volume" );
189 configurePanel.add( volumeLabel );
190
191 volumeSlider = new JSlider(
192     SwingConstants.HORIZONTAL, 5, 80, 30 );
193
194 // registra um ChangeListener para eventos de controles deslizantes
195 volumeSlider.addChangeListener(
196
197     // classe interna anônima para tratar de eventos do controle de volume
198     new ChangeListener() {
199
200         // muda volume
201         public void stateChanged( ChangeEvent changeEvent )
202         {
203             midiVolume = volumeSlider.getValue();
204         }
205
206     } // fim da classe ChangeListener
207
208 ); // fim da chamada para o método addChangeListener
209
210 configurePanel.add( volumeSlider );
211
212 JLabel tempoLabel = new JLabel( "tempo" );
213 configurePanel.add( tempoLabel );
214
215 resolutionSlider = new JSlider(
216     SwingConstants.HORIZONTAL, 1, 10, 1 );
217
218 // registra um ChangeListener para eventos do controle de tempo
219 resolutionSlider.addChangeListener(
220
221     // classe interna anônima para tratar eventos do controle de tempo
222     new ChangeListener() {
223
224         // muda resolução se o valor mudou
225         public void stateChanged( ChangeEvent changeEvent )
226         {
227             resolution = resolutionSlider.getValue();
228         }
229
230     } // fim de ChangeListener
231
232 ); // fim da chamada para o método addChangeListener
233
234 resolutionSlider.setEnabled( false );

```

Fig. 22.10 MidiDemo fornece a GUI que permite que os usuários interajam com o aplicativo (parte 4 de 13).

```

235     configurePanel.add( resolutionSlider );
236
237 } // fim do método makeConfigureControls
238
239 // configura botões de tocar e salvar
240 private void makePlaySaveButtons()
241 {
242     playButton = new JButton( "Playback" );
243
244     // registra um ActionListener para eventos do playButton
245     playButton.addActionListener(
246
247         // classe interna anônima para tratar de eventos do playButton
248         new ActionListener() {
249
250             // reproduz o último MIDI gravado
251             public void actionPerformed( ActionEvent event )
252             {
253                 if ( midiRecord != null )
254                     midiRecord.play();
255             }
256
257         } // fim de ActionListener
258
259     ); // fim da chamada para o método addActionListener
260
261     buttonPanel.add( playButton );
262     playButton.setEnabled( false );
263
264     listenButton = new JButton( "Play MIDI" );
265
266     // registra um ActionListener para eventos do listenButton
267     listenButton.addActionListener(
268
269         // classe interna anônima para tratar de eventos do listenButton
270         new ActionListener() {
271
272             // reproduz arquivo MIDI
273             public void actionPerformed( ActionEvent event )
274             {
275                 File midiFile = getFile();
276
277                 if ( midiFile == null )
278                     return;
279
280                 midiData = new MidiData();
281
282                 // prepara trilha MIDI
283                 if ( midiData.initialize( midiFile ) == false )
284                     return;
285
286                 // reproduz dados MIDI
287                 midiData.play();
288             }
289
290         } // fim de ActionListener
291
292     ); // fim da chamada para o método addActionListener
293

```

Fig. 22.10 MidiDemo fornece a GUI que permite que os usuários interajam com o aplicativo (parte 5 de 13).

```

294     buttonPanel.add( listenButton );
295
296     saveButton = new JButton( "Save MIDI" );
297
298     // registra um ActionListener para eventos do saveButton
299     saveButton.addActionListener(
300
301         // classe interna anônima para tratar de eventos do saveButton
302         new ActionListener() {
303
304             // obtém arquivo de salvamento e salva MIDI gravado
305             public void actionPerformed( ActionEvent event )
306             {
307                 File saveFile = getSaveFile();
308
309                 if ( saveFile != null )
310                     midiRecord.saveSequence( saveFile );
311             }
312         } // fim de ActionListener
313
314     ); // fim da chamada para o método addActionListener
315
316     buttonPanel.add( saveButton );
317     saveButton.setEnabled( false );
318
319 } // fim do método makePlaySaveButtons
320
321 // cria botão de gravação
322 private void makeRecordButton()
323 {
324     recordButton = new JButton( "Record" );
325
326     // registra um ActionListener para eventos do recordButton
327     recordButton.addActionListener(
328
329         // classe interna anônima para tratar de eventos do recordButton
330         new ActionListener() {
331
332             // iniciar ou parar a gravação
333             public void actionPerformed( ActionEvent event )
334             {
335                 // grava MIDI quando o botão é "record"
336                 if ( recordButton.getText().equals("Record") ) {
337
338                     if ( midiRecord == null ) {
339
340                         // cria nova instância do gravador passando
341                         // a ela o transmissor do sintetizador
342                         midiRecord = new MidiRecord(
343                             midiSynthesizer.getTransmitter() );
344
345                         if ( midiRecord.initialize() == false )
346                             return;
347                     }
348
349                     else
350                         midiRecord.makeTrack();
351                 }
352             }
353         }
354     );
355
356     buttonPanel.add( recordButton );
357     recordButton.setEnabled( false );
358 }

```

Fig. 22.10 MidiDemo fornece a GUI que permite que os usuários interajam com o aplicativo (parte 6 de 13).

```

353         midiRecord.startRecord();
354
355         // inibe reprodução durante a gravação
356         playButton.setEnabled( false );
357
358         // muda botão de gravação para "stop"
359         recordButton.setText( "Stop" );
360         recording = true;
361
362     } // fim do if
363
364     // parar a gravação quando o botão é "stop"
365     else {
366         midiRecord.stopRecord();
367
368         recordButton.setText( "Record" );
369         recording = false;
370
371         playButton.setEnabled( true );
372         saveButton.setEnabled( true );
373     }
374
375     } // fim do método actionPerformed
376
377     } // fim de ActionListener
378
379 ); // fim da chamada para o método addActionListener
380
381 buttonPanel.add( recordButton );
382
383 } // fim do método makeRecordButton
384
385 // cria botão e funcionalidade do Tocador de Piano
386 private void makePianoPlayerButton()
387 {
388     pianoPlayerButton = new JButton( "Piano Player" );
389
390     // registra um ActionListener para eventos do pianoPlayerButton
391     pianoPlayerButton.addActionListener(
392
393         // classe interna anônima para tratar do pianoPlayerButton
394         new ActionListener() {
395
396             // inicializa dados MIDI e timer do tocador de piano
397             public void actionPerformed( ActionEvent event )
398             {
399                 File midiFile = getFile();
400
401                 if ( midiFile == null )
402                     return;
403
404                 midiData = new MidiData();
405
406                 // prepara trilha MIDI
407                 if ( midiData.initialize( midiFile ) == false )
408                     return;
409
410                 if ( midiData.initializeTrack() == false )
411                     return;

```

Fig. 22.10 MidiDemo fornece a GUI que permite que os usuários interajam com o aplicativo (parte 7 de 13).

```

412
413         // ajusta resolução inicial de MIDI
414         resolution = midiData.getResolution();
415
416         // nova instância de timer para tratar sons do
417         // piano e pressionamento de teclas com tempo
418         pianoTimer = new Timer(
419             midiData.getEventDelay() * resolution,
420             new TimerHandler() );
421
422         listenButton.setEnabled( false );
423         pianoPlayerButton.setEnabled( false );
424         resolutionSlider.setEnabled( true );
425
426         pianoTimer.start();
427
428     } // fim do método actionPerformed
429
430 } // fim de ActionListener
431
432 ); // fim da chamada para o método addActionListener
433
434 buttonPanel.add( pianoPlayerButton );
435
436 } // fim do método makePianoPlayerButton
437
438 // classe interna trata de eventos MIDI temporizados
439 private class TimerHandler implements ActionListener {
440
441     // simula a nota da tecla do evento se presente, pula para o
442     // próximo evento na trilha e ajusta novo intervalo de retardo do
443     // método do timer quando o timer atinge a hora do próximo evento
444     public void actionPerformed((ActionEvent actionEvent) )
445     {
446         // se uma última tecla válida estiver ligada, muda-a para branco
447         if ( lastKeyOn != -1 )
448             noteButton[ lastKeyOn ].setBackground(
449                 Color.white );
450
451         noteAction();
452         midiData.goNextEvent();
453
454         // faz parar o tocador de piano quando acha o fim da trilha MIDI
455         if ( midiData.isTrackEnd() == true ) {
456
457             if ( lastKeyOn != -1 )
458                 noteButton[ lastKeyOn ].setBackground(
459                     Color.white );
460
461             pianoTimer.stop();
462
463             listenButton.setEnabled( true );
464             pianoPlayerButton.setEnabled( true );
465             resolutionSlider.setEnabled( false );
466
467             return;
468
469         } // fim de if isTrackEnd
470

```

Fig. 22.10 MidiDemo fornece a GUI que permite que os usuários interajam com o aplicativo (parte 8 de 13).

```

471         // ajusta intervalo antes do próximo evento de som
472         pianoTimer.setDelay(
473             midiData.getEventDelay() * resolution );
474
475     } // fim do método actionPerformed
476
477 } // fim da classe interna TimerHandler
478
479 // determina que nota emitir, de
480 // acordo com as mensagens MIDI
481 private void noteAction()
482 {
483     // durante a mensagem Note On, emite a nota e pressiona tecla
484     if ( midiData.getEventCommand() ==
485         ShortMessage.NOTE_ON ) {
486
487         // assegura que nota válida está no intervalo das teclas
488         if ( ( midiData.getNote() >= FIRST_NOTE ) &&
489             ( midiData.getNote() < FIRST_NOTE + MAX_KEYS ) ) {
490
491             lastKeyOn = midiData.getNote() - FIRST_NOTE;
492
493             // ajusta a cor da tecla para vermelho
494             noteButton[ lastKeyOn ].setBackground( Color.red );
495
496             // envia e faz soar a nota através do sintetizador
497             midiSynthesizer.sendMessage( 144,
498                 midiData.getNote(), midiData.getVolume() );
499
500         } // fim do if
501
502         // senão, não há última tecla pressionada
503         else
504             lastKeyOn = -1;
505
506     } // fim do if
507
508     // receber a mensagem Note Off faz parar de emitir
509     // a nota e mudar a cor da tecla de volta para branco
510     else
511
512         // se o comando da mensagem for desligar a nota
513         if ( midiData.getEventCommand() ==
514             ShortMessage.NOTE_OFF ) {
515
516             if ( ( midiData.getNote() >= FIRST_NOTE ) &&
517                 ( midiData.getNote() < FIRST_NOTE + MAX_KEYS ) ) {
518
519                 // ajusta a tecla apropriada para branco
520                 noteButton[ midiData.getNote() -
521                     FIRST_NOTE ].setBackground( Color.white );
522
523                 // envia mensagem de desligar a nota para o receptor
524                 midiSynthesizer.sendMessage( 128,
525                     midiData.getNote(), midiData.getVolume() );
526             }
527
528         } // fim do if
529

```

Fig. 22.10 MidiDemo fornece a GUI que permite que os usuários interajam com o aplicativo (parte 9 de 13).

```

530     } // fim do método noteAction
531
532     // obtém arquivo de salvamento do computador
533     public File getSaveFile()
534     {
535         JFileChooser fileChooser = new JFileChooser();
536
537         fileChooser.setFileSelectionMode(
538             JFileChooser.FILES_ONLY );
539         int result = fileChooser.showSaveDialog( this );
540
541         if ( result == JFileChooser.CANCEL_OPTION )
542             return null;
543
544         else
545             return fileChooser.getSelectedFile();
546     }
547
548     // obtém arquivo do computador
549     public File getFile()
550     {
551         JFileChooser fileChooser = new JFileChooser();
552
553         fileChooser.setFileSelectionMode(
554             JFileChooser.FILES_ONLY );
555         int result = fileChooser.showOpenDialog( this );
556
557         if ( result == JFileChooser.CANCEL_OPTION )
558             return null;
559
560         else
561             return fileChooser.getSelectedFile();
562     }
563
564     // executa o aplicativo
565     public static void main( String args[] )
566     {
567         MidiDemo midiTest = new MidiDemo();
568
569         midiTest.setSize( 711, 225 );
570         midiTest.setDefaultCloseOperation ( EXIT_ON_CLOSE );
571         midiTest.setVisible( true );
572     }
573
574 } // fim da classe MidiDemo

```

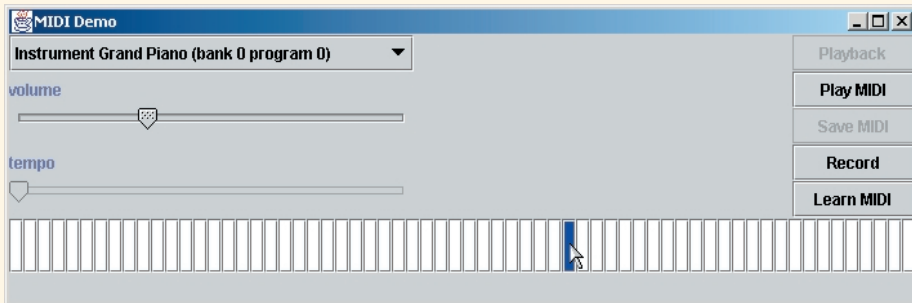


Fig. 22.10 *MidiDemo* fornece a GUI que permite que os usuários interajam com o aplicativo (parte 10 de 13).

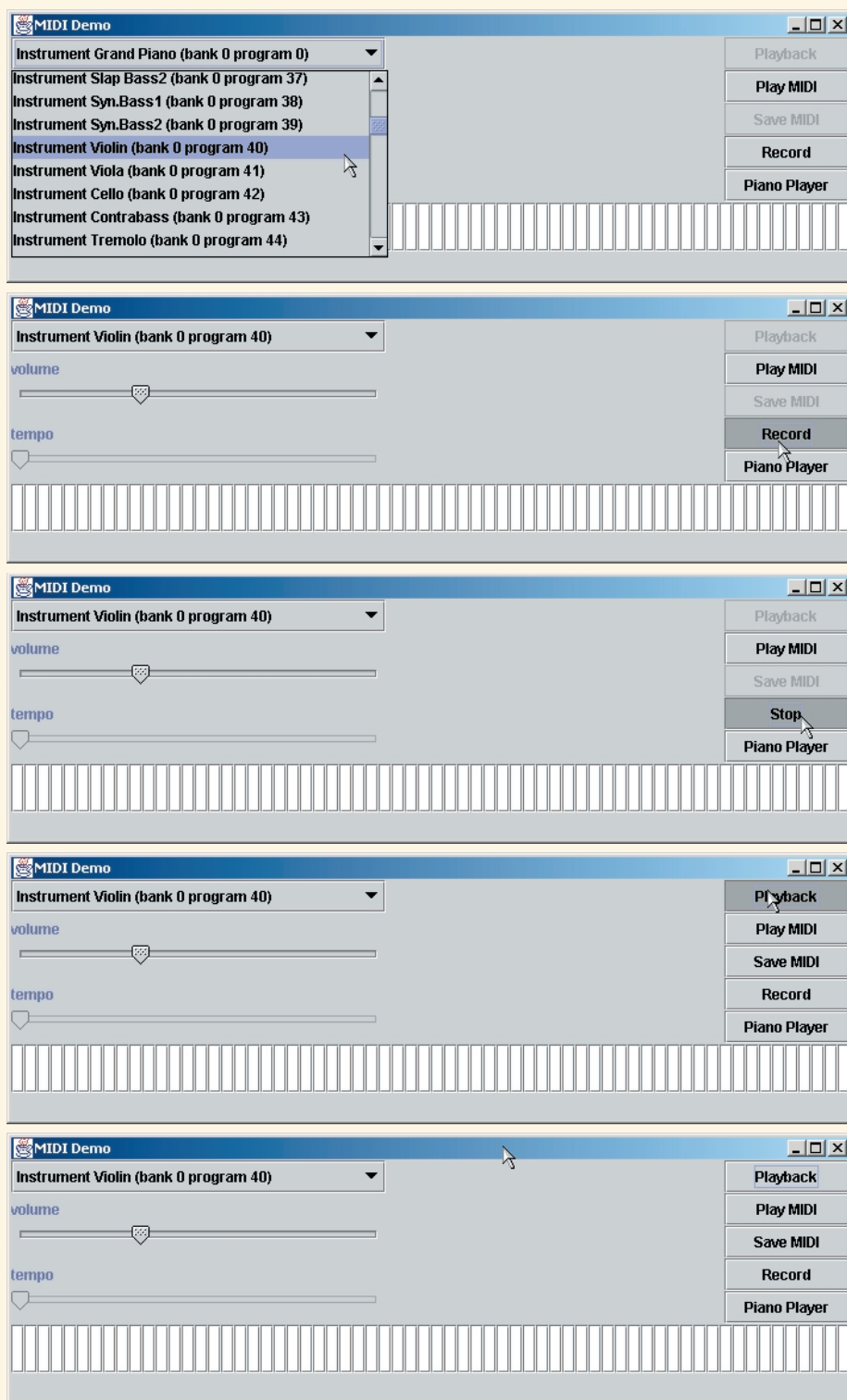


Fig. 22.10 *MidiDemo* fornece a GUI que permite que os usuários interajam com o aplicativo (parte 11 de 13).

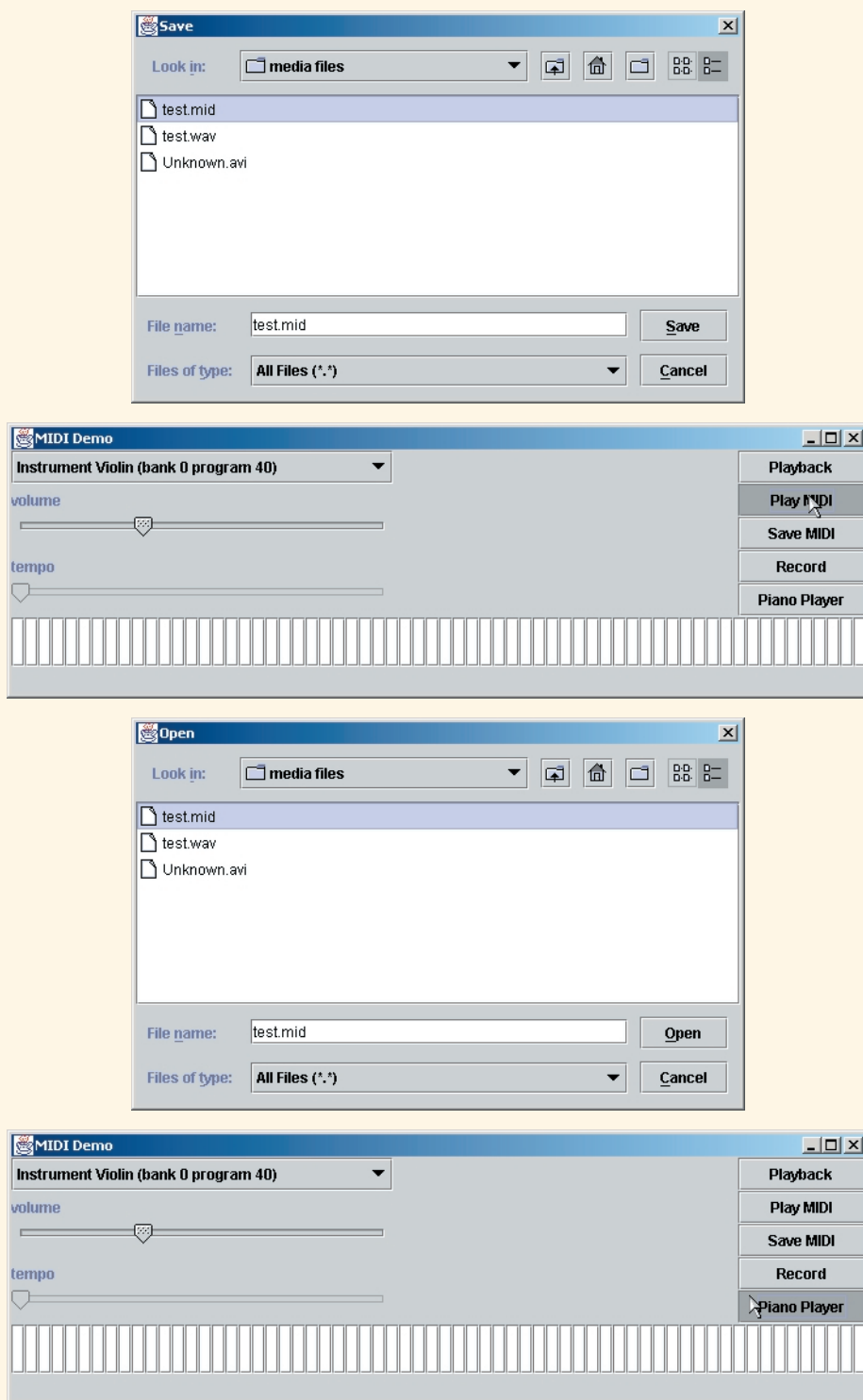


Fig. 22.10 MidiDemo fornece a GUI que permite que os usuários interajam com o aplicativo (parte 12 de 13).

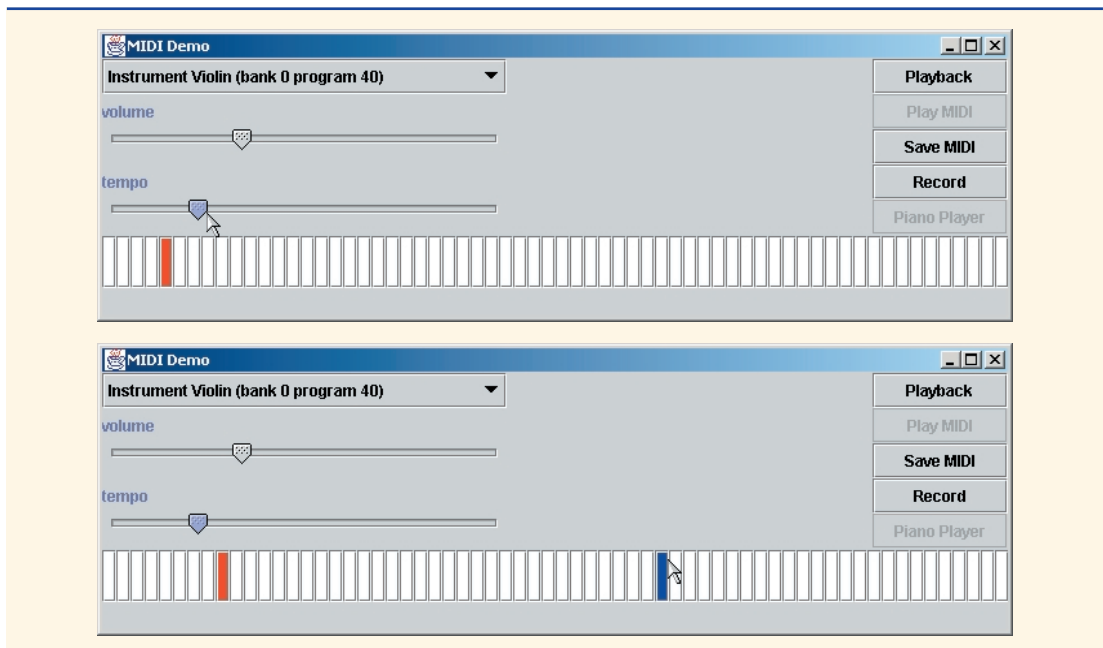


Fig. 22.10 *MidiDemo* fornece a GUI que permite que os usuários interajam com o aplicativo (parte 13 de 13).



Observação de aparência e comportamento 22.3

Para dispor componentes GUI em posições específicas sem a ajuda de gerenciadores de layout, configure o layout do painel que contém os componentes como `null`. Por default, o `JPanel` configura o `LayoutManager FlowLayout` quando o painel é instanciado sem argumentos.

As linhas 110 a 151 registram `MouseListeners` para cada botão de tecla do piano. O programa chama o método `mouseEntered` (linhas 116 a 132) quando o mouse passa sobre aquele botão. Se o programa não estiver no modo de gravação, o método `mouseEntered` acessa diretamente os canais na classe `MidiSynthesizer` para emitir a nota (linhas 125 a 127). Caso contrário, o método `mouseEntered` invoca o método `sendMessage` do `MidiSynthesizer` para enviar uma mensagem de nota para o sintetizador e para o dispositivo de gravação (linhas 119 a 122). As linhas 130 e 131 ajustam a cor de fundo do botão para azul para indicar que a nota está sendo tocada. Quando o mouse não está mais passando sobre o botão, o programa chama o método `mouseExited` (linhas 135 a 147) para desligar a nota e mudar o fundo do botão para sua cor original.

Das 128 notas possíveis, somente as 64 notas do meio são acessáveis através do piano no exemplo. O intervalo de notas pode ser mudado modificando-se as constantes na linha 52. A constante `FIRST_NOTE` é o valor da primeira tecla e a soma de `FIRST_NOTE` e `MAX_KEYS` é o valor da última tecla. A constante `MAX_KEYS` especifica o número de teclas do piano a criar.

A classe `MidiDemo` invoca o método `makeConfigureControls` (linhas 158 a 237) para configurar os controles de MIDI do programa, que consistem em uma `JComboBox` seletora de instrumento, um `JSlider` para mudar o volume de síntese pelo usuário e um `JSlider` para mudar o tempo do “tocador de piano”. Quando os usuários selecionam um instrumento, o programa chama o método `actionPerformed` de `instrumentBox` (linhas 177 a 182) para mudar o programa do instrumento selecionado invocando o método `changeInstrument` de `MidiSynthesizer` com o índice do instrumento selecionado como parâmetro.

Quando os usuários arrastam o controle deslizante de volume, o programa chama o método `stateChanged` de `volumeSlider` (linhas 201 a 204) para mudar o volume. Observe que mudar o volume afeta somente o volume das notas MIDI sintetizadas pelo usuário. Quando os usuários arrastam o controle deslizante de tempo, o programa chama o método `stateChanged` de `resolutionSlider` (linhas 225 a 228) para ajustar o tempo.

Invocar o método **makePlaySaveButtons** (linhas 240 a 320) configura os botões **Play MIDI**, **Playback** e **Save**. Clicar em **Play MIDI** invoca o método **actionPerformed** do **listenButton** (linhas 273 a 288) para reproduzir em sua totalidade um arquivo MIDI aberto com a classe **MidiData**. A linha 275 obtém um arquivo de um diálogo de escolha de arquivo com o método **getFile** de **MidiDemo** (linhas 549 a 562). As linhas 280 a 284 inicializam e reproduzem o arquivo MIDI com o objeto **midiData** instanciado. Quando o usuário clica em **Playback**, a linha 254 reproduz o MIDI gravado. Este botão fica habilitado somente se alguma gravação foi feita. Clicar no botão **Save** permite salvar a sequência gravada em um arquivo (linhas 307 a 310).

O método **makeRecordButton** (linhas 323 a 383) cria o botão **Record** e um ouvinte para ele. Clicar no botão quando ele está ajustado para modo de gravação (linha 337) cria um novo gravador com a classe **MidiRecord** (linhas 339 a 348). Se já foi criado um gravador, a linha 351 invoca o método **makeTrack** de **MidiRecord** para criar uma nova trilha para o objeto **midiRecord**. Quando a gravação começa (linha 353), as linhas 356 a 360 mudam o botão de gravação para um botão de parar e inibem o **playButton** temporariamente. Quando os usuários param a gravação clicando no botão de gravação novamente, a GUI retorna para seu estado anterior à gravação e o usuário pode reproduzir e salvar a sequência MIDI (linhas 365 a 373).

“Tocador de piano”

A função “tocador de piano” deste exemplo sincroniza a reprodução de uma nota com o destaque da tecla correspondente. O programa primeiro obtém uma trilha MIDI de um arquivo MIDI especificado pelo usuário com a classe **MidiData**. O **Timer** sincroniza os eventos MIDI na sequência MIDI. Quando o **Timer** atinge o *time stamp* de um evento MIDI, ele restaura seu *retardo* para o *time stamp* do próximo evento a ser sincronizado. O retardo de um *timer* é o período de tempo que ele espera antes de provocar um evento.

O “acionador do tocador de piano” responsável pela sincronização está localizado na classe principal **MidiDemo**. O método **makePianoPlayerButton** (linhas 386 a 436) carrega o arquivo MIDI e inicializa o **Timer** que controla a temporização das notas musicais. Assim como ocorre com o **listenButton**, o método **actionPerformed** (linhas 397 a 428) de **makePianoPlayerButton** usa a classe **MidiData** para carregar dados MIDI. As linhas 399 a 408 abrem um arquivo a partir de uma caixa de diálogo de arquivo e carregam os dados MIDI do arquivo. A linha 410 invoca o método **initializeTrack** de **MidiData** para obter a trilha mais longa do MIDI carregado e a primeira mensagem de evento MIDI da trilha.

A linha 414 invoca o método **getResolution** de **MidiData** (linhas 67 a 70 na Fig. 22.7) para obter o tempo *default* do “tocador de piano”. As linhas 418 a 420 instanciam um novo objeto **Timer** que aciona a reprodução das notas no momento de cada evento MIDI. A linha 419 ajusta o retardo do *timer* para ser a duração do primeiro **MidiEvent** na trilha, invocando o método **getEventDelay** de **MidiData**. Para permitir mudanças de tempo especificadas pelo usuário para o “tocador de piano”, a linha 424 habilita o controle deslizante de tempo e a linha 419 ajusta o retardo do *timer* para ser multiplicado pelo valor de **resolution**. A variável **resolution**, que especifica o tempo do “tocador de piano”, é multiplicada pelo intervalo de tempo do evento para se obter um novo intervalo de tempo usado por **pianoTimer**. O retardo de **pianoTimer** é ajustado somente para a duração do primeiro evento a disparar o *timer*. Mais tarde, o método **actionPerformed** (linhas 444 a 475) da classe interna **TimerHandler** (linhas 439 a 477) restaura o retardo do *timer* para a duração do próximo **MidiEvent** (linhas 472 e 473). A linha 426 dispara o **pianoTimer**.

No momento do próximo evento MIDI, o método **actionPerformed** da classe **TimerHandler** sintetiza uma nota da trilha e “pressiona” uma tecla no piano. Usando **pianoTimer** e seu tratador de eventos, o programa percorre a trilha, simulando eventos de notas quando ele encontra **MidiEvents** apropriados. A classe interna **TimerHandler** aciona a sincronização iterando através de todos os **MidiEvents** em uma dada trilha e chamando o método **actionPerformed** para tocar o piano. A linha 451 do método **actionPerformed** invoca o método utilitário **noteAction** (linhas 481 a 530) para emitir o som da nota e mudar a cor da tecla do piano específica, desde que o evento contenha uma mensagem de nota e que a nota esteja dentro do intervalo designado para as teclas do piano neste exemplo.

As linhas 484 e 485 do método **noteAction** determinam se o **MidiEvent** atual contém um comando de nota invocando o método **getEventCommand** da classe **MidiData**. As linhas 488 e 489 invocam o método **getNote** de **MidiData** para determinar se o número da nota especificado na **noteMessage** é de uma nota válida e dentro do intervalo de teclas possíveis do piano, especificado nas constantes **FIRST_NOTE** e **MAX_KEYS** (linha 52).

Se o comando for **ShortMessage.NOTE_ON** e estiver dentro do intervalo das teclas do piano, o sintetizador recebe a mensagem de nota especificada (linhas 497 e 498) e a cor da tecla do piano correspondente fica verme-

lha pela duração do evento (linha 494). A linha 491 obtém o número de botão do piano que deve ficar vermelho de forma correspondente (`lastKeyOn`).

Se o comando `for ShortMessage.NOTE_OFF` e estiver dentro do intervalo permitido para as teclas do piano (linhas 513 a 517), as linhas 520 e 521 mudam o fundo da tecla do piano específica de volta para branco. As linhas 524 e 525 enviam a `ShortMessage` para o sintetizador para que a nota pare de soar. Como nem todas as `ShortMessages` `NOTE_ON` são seguidas por uma `ShortMessage` `NOTE_OFF`, como se poderia esperar, o programa precisa mudar a tecla da última `NOTE_ON` de volta para sua cor original no momento do próximo evento. Para este fim, o método `noteAction` atribui a `lastKeyOn` o valor do último botão do piano invocado. O objeto `lastKeyOn` inicializado com `-1` permanece `-1` se a nota do comando `NOTE_ON` estiver fora do intervalo. Isto limita o acesso somente à teclas no intervalo de nosso teclado simulado. Quando `pianoTimer` atinge o próximo evento, o programa muda o fundo da última tecla do piano “pressionada” de volta para branco (linhas 447 a 449).

Quando o programa termina de executar o método `noteAction`, a linha 452 invoca o método `goNextEvent` de `MidiData` para passar para o próximo evento na trilha. Toda vez que o método `actionPerformed` do tratador termina de carregar o próximo evento, a linha 455 determina se o próximo evento é o último evento na trilha invocando o método `isTrackEnd` de `MidiData`, assumindo-se que o último evento é o `MetaEvent` fim de trilha. Se o próximo evento for o último evento, as linhas 457 a 459 mudam a cor de fundo da última tecla “pressionada” para branco e a linha 461 faz parar o `pianoTimer`. As linhas 463 a 465 reabilitam os botões que foram desabilitados durante a função “tocador de piano”.

22.8 Recursos na Internet e na World Wide Web

Esta seção apresenta diversos recursos na Internet e na Web para a Java Media Framework e outros *sites* relacionados com multimídia.

java.sun.com/products/java-media/jmf/

A homepage da Java Media Framework no site de Java na Web. Aqui você pode baixar a última implementação pela Sun do JMF. O site também contém a documentação para o JMF.

www.nasa.gov/gallery/index.html

A NASA *multimedia gallery* contém uma ampla variedade de imagens, clipes de áudio e vídeos que você pode baixar e usar para testar seus programas de multimídia em Java.

sunsite.sut.ac.jp/multimed/

A *Sunsite Japan Multimedia Collection* também fornece uma ampla variedade de imagens, clipes de áudio e vídeos que você pode baixar para fins educacionais.

www.anbg.gov.au/anbg/index.html

O site na Web do *Australian National Botanic Gardens* fornece links para sons de muitos animais. Experimente o link *Common Birds*.

www.midi.com

Midi.com é um site de recursos para MIDI com um mecanismo de busca de MIDI, links para outros sites, uma lista de livros relacionados com MIDI e outras informações sobre MIDI.

www.streamingmedia.com

Streamingmedia.com fornece muitos artigos do setor de mídia de *streaming* e informações técnicas sobre a tecnologia de mídia de *streaming*.

www.harmony-central.com/MIDI/

A seção de recursos para MIDI da *Harmony Central* contém muitos documentos úteis sobre MIDI, links e fóruns que podem ser úteis para um programador MIDI.

22.9 (Estudo de caso opcional) Pensando em objetos: animação e som na visão

Este estudo de caso se concentrou principalmente no modelo do sistema do elevador. Agora que completamos nosso projeto do modelo, voltamos nossa atenção para a *visão*, que fornece a apresentação visual do modelo. Em nosso estudo de caso, a visão – chamada de `ElevatorView` – é um objeto `JPanel` que contém outros objetos `JPanel` “filhos”, cada um representando um objeto único no modelo (por exemplo, uma `Person`, um `Button`, o

Elevator). A classe **ElevatorView** é a maior classe no estudo de caso. Nesta seção, discutimos as classes de gráficos e som usadas pela classe **ElevatorView**. Apresentamos e explicamos o resto do código nesta classe no Apêndice I.

Na Seção 3.7, construímos o diagrama de classes para nosso modelo localizando os substantivos e as frases com substantivos da definição do problema da Seção 2.7. Ignoramos diversos destes substantivos, porque eles não estavam associados com o modelo. Agora, listamos os substantivos e as frases com substantivos que se aplicam à exibição do modelo:

- exibição
- áudio
- música de elevador

O substantivo “exibição” corresponde à visão, ou a apresentação visual do modelo. Como descrito na Seção 13.17, a classe **ElevatorView** agrega diversas classes que compreendem a visão. O “áudio” se refere aos efeitos de som que nossa simulação gera quando ocorrem diversas ações – criamos a classe **SoundEffects** para gerar estes efeitos de som. A frase “música de elevador” se refere à música tocada enquanto a **Person** anda no **Elevator** – criamos a classe **ElevatorMusic** para tocar esta música.

A visão deve exibir todos os objetos no modelo. Criamos a classe **ImagePanel** para representar objetos imóveis no modelo, como o **ElevatorShaft**. Criamos a classe **MovingPanel**, que estende **ImagePanel**, para representar objetos em movimento, como o **Elevator**. Finalmente, criamos a classe **AnimatedPanel**, que estende **MovingPanel**, para representar objetos que se movem cujas imagens correspondentes mudam continuamente, como uma **Person** (usamos diversas *frames* de animação para mostrar a **Person** caminhando e depois pressionando um botão). Usando estas classes, apresentamos o diagrama de classes da visão para nossa simulação na Fig. 22.11.

As anotações indicam o papel que as classes desempenham no sistema. De acordo com o diagrama de classes, a classe **ElevatorView** representa a visão, as classes **ImagePanel**, **MovingPanel** e **AnimatedPanel** estão relacionadas com os gráficos e as classes **SoundEffects** e **ElevatorMusic** estão relacionados com o áudio. A classe **ElevatorView** contém diversas instâncias das classes **ImagePanel**, **MovingPanel** e **AnimatedPanel** e uma instância de cada uma das classes **SoundEffects** e **ElevatorMusic**. No Apêndice I, associamos cada objeto no modelo com uma classe correspondente na visão.

Nesta seção, discutimos as classes **ImagePanel**, **MovingPanel** e **AnimatedPanel** para explicar os gráficos e a animação. Depois discutimos as classes **SoundEffects** e **ElevatorMusic** para explicar a funcionalidade de áudio.

Image Panel

A **ElevatorView** usa objetos de subclasses de **JPanel** para representar e exibir cada objeto no modelo (como o **Elevator**, uma **Person**, o **ElevatorShaft**, etc.). A classe **ImagePanel** (Fig. 22.12) é uma subclasse de **JPanel** capaz de exibir uma imagem em uma posição da tela dada. A **ElevatorView** usa objetos **ImagePanel** para representar objetos imóveis no modelo, como o **ElevatorShaft** e os dois **Floors**. A classe **ImagePanel** contém um atributo inteiro – **ID** (linha 16) – que define um identificador único usado para monitorar o **ImagePanel** na visão se necessário. Este monitoramento é útil quando diversos objetos da mesma classe existem no modelo, como diversos objetos **Person**. A classe **ImagePanel** contém o objeto **Point2D.Double position** (linha 19) para representar a posição na tela do **ImagePanel**. Veremos mais tarde que **MovingPanel**, que estende **ImagePanel**, define a velocidade com **doubles** – usar o tipo **double** garante velocidade e posição altamente precisas. Convertemos as coordenadas de **position** para **ints** para posicionar o **ImagePanel** na tela (Java representa as coordenadas na tela como **ints**) no método **setPosition** (linhas 90 a 94). A classe **ImagePanel** também contém um objeto **ImageIcon** chamado **imageIcon** (linha 22) – o método **paintComponent** (linhas 54 a 60) exibe **imageIcon** na tela. As linhas 41 e 42 inicializam **imageIcon** com um parâmetro **String** que contém o nome da imagem. Finalmente, a classe **ImagePanel** contém o **Set panelChildren** (linha 25) que armazena quaisquer objetos-filhos da classe **ImagePanel** (ou objetos de uma subclasse de **ImagePanel**). Os objetos-filhos são exibidos sobre o **ImagePanel** de seu pai – por exemplo, uma **Person** andando dentro do **Elevator**. O primeiro método **add** (linhas 63 a 67) anexa um objeto a **panelChildren**. O segundo método **add** (linhas 70 a 74) insere um objeto em **panelChildren** em um índice determinado. O método **setIcon** (linhas 84 a 87) configura **imageIcon** com uma nova imagem. Os objetos da classe **AnimatedPanel** usam

o método **setIcon** repetidamente para mudar a imagem exibida, o que provoca a animação para a visão – discutimos a animação mais tarde na seção.

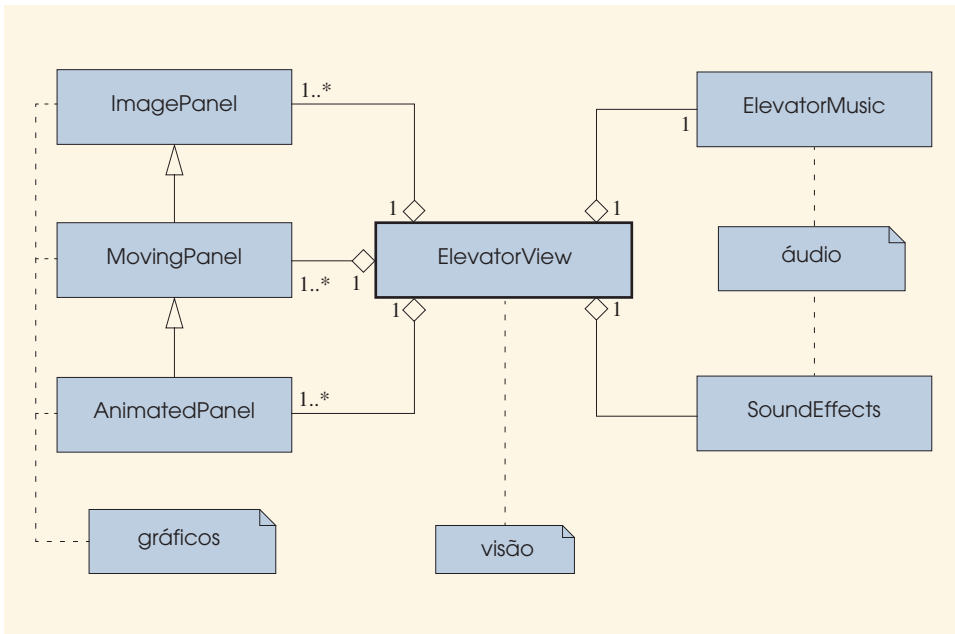


Fig. 22.11 Diagrama de classes da visão da simulação do elevador.

```

1  // ImagePanel.java
2  // Subclasse de JPanel para posicionar e exibir ImageIcon
3  package com.deitel.jhtp4.elevator.view;
4
5  // Pacotes do núcleo de Java
6  import java.awt.*;
7  import java.awt.geom.*;
8  import java.util.*;
9
10 // Pacotes de extensão de Java
11 import javax.swing.*;
12
13 public class ImagePanel extends JPanel {
14
15     // identificador
16     private int ID;
17
18     // posição na tela
19     private Point2D.Double position;
20
21     // ImageIcon para pintar na tela
22     private ImageIcon imageIcon;
23
24     // armazena todos os filhos de ImagePanel
25     private Set panelChildren;
26
27     // construtor inicializa posição e imagem
28     public ImagePanel( int identifier, String imageName )

```

Fig. 22.12 A classe **ImagePanel** representa e exibe um objeto imóvel do modelo (parte 1 de 3).

```

29     {
30         super( null );    // especifica leiaute nulo
31         setOpaque( false );    // torna transparente
32
33         // configura identificador único
34         ID = identifier;
35
36         // configura posição
37         position = new Point2D.Double( 0, 0 );
38         setLocation( 0, 0 );
39
40         // cria ImageIcon com o imageName dado
41         imageIcon = new ImageIcon(
42             getClass().getResource( imageName ) );
43
44         Image image = imageIcon.getImage();
45         setSize(
46             image.getWidth( this ), image.getHeight( this ) );
47
48         // cria Set para armazenar filhos do Panel
49         panelChildren = new HashSet();
50
51     } // fim do construtor ImagePanel
52
53     // pinta Panel na tela
54     public void paintComponent( Graphics g )
55     {
56         super.paintComponent( g );
57
58         // se a imagem está pronta, pinta-a na tela
59         imageIcon.paintIcon( this, g, 0, 0 );
60     }
61
62     // adiciona filho de ImagePanel a ImagePanel
63     public void add( ImagePanel panel )
64     {
65         panelChildren.add( panel );
66         super.add( panel );
67     }
68
69     // adiciona filho de ImagePanel a ImagePanel em um índice determinado
70     public void add( ImagePanel panel, int index )
71     {
72         panelChildren.add( panel );
73         super.add( panel, index );
74     }
75
76     // remove filho de ImagePanel do ImagePanel
77     public void remove( ImagePanel panel )
78     {
79         panelChildren.remove( panel );
80         super.remove( panel );
81     }
82
83     // configura ImageIcon corrente para ser exibido
84     public void setIcon( ImageIcon icon )
85     {
86         imageIcon = icon;
87     }
88

```

Fig. 22.12 A classe `ImagePanel` representa e exibe um objeto imóvel do modelo (parte 2 de 3).

```

89 // configura posição na tela
90 public void setPosition( double x, double y )
91 {
92     position.setLocation( x, y );
93     setLocation( ( int ) x, ( int ) y );
94 }
95
96 // devolve identificador do ImagePanel
97 public int getID()
98 {
99     return ID;
100 }
101
102 // obtém posição do ImagePanel
103 public Point2D.Double getPosition()
104 {
105     return position;
106 }
107
108 // obtém ImageIcon
109 public ImageIcon getImageIcon()
110 {
111     return imageIcon;
112 }
113
114 // obtém Set de filhos de ImagePanel
115 public Set getChildren()
116 {
117     return panelChildren;
118 }
119 }

```

Fig. 22.12 A classe `ImagePanel` representa e exibe um objeto imóvel do modelo (parte 3 de 3).

Moving Panel

A classe `MovingPanel` (Fig. 22.13) é uma subclasse de `ImagePanel` capaz de mudar sua posição na tela de acordo com sua `xVelocity` e `yVelocity` (linhas 20 e 21). A `ElevatorView` utiliza objetos `MovingPanel` para representar objetos do modelo que se movimentam, como o `Elevator`.

```

1 // MovingPanel.java
2 // Subclasse de JPanel com capacidade de se mover na tela
3 package com.deitel.jhttp4.elevator.view;
4
5 // Pacotes do núcleo de Java
6 import java.awt.*;
7 import java.awt.geom.*;
8 import java.util.*;
9
10 // Pacotes de extensão de Java
11 import javax.swing.*;
12
13 public class MovingPanel extends ImagePanel {
14
15     // o MovingPanel deve mudar de posição?
16     private boolean moving;
17

```

Fig. 22.13 A classe `MovingPanel` representa e exibe um objeto do modelo em movimento (parte 1 de 3).

```

18 // número de pixels que MovingPanel se move na direção das
19 // coordenadas x e y por milissegundos de animationDelay
20 private double xVelocity;
21 private double yVelocity;
22
23 // construtor inicializa posição, velocidade e imagem
24 public MovingPanel( int identifier, String imageName )
25 {
26     super( identifier, imageName );
27
28     // configura velocidade do MovingPanel
29     xVelocity = 0;
30     yVelocity = 0;
31
32 } // fim do construtor MovingPanel
33
34 // atualiza posição e frame de animação do MovingPanel
35 public void animate()
36 {
37     // atualiza posição de acordo com a velocidade do MovingPanel
38     if ( isMoving() ) {
39         double oldXPosition = getPosition().getX();
40         double oldYPosition = getPosition().getY();
41
42         setPosition( oldXPosition + xVelocity,
43                     oldYPosition + yVelocity );
44     }
45
46     // atualiza todos os filhos de MovingPanel
47     Iterator iterator = getChildren().iterator();
48
49     while ( iterator.hasNext() ) {
50         MovingPanel panel = ( MovingPanel ) iterator.next();
51         panel.animate();
52     }
53 } // fim do método animate
54
55 // o MovingPanel está se movendo na tela?
56 public boolean isMoving()
57 {
58     return moving;
59 }
60
61 // configura o MovingPanel para se mover na tela
62 public void setMoving( boolean move )
63 {
64     moving = move;
65 }
66
67 // configura as velocidades em x e y de MovingPanel
68 public void setVelocity( double x, double y )
69 {
70     xVelocity = x;
71     yVelocity = y;
72 }
73
74 // devolve a velocidade em x de MovingPanel
75 public double getXVelocity()
76 {

```

Fig. 22.13 A classe `MovingPanel` representa e exibe um objeto do modelo em movimento (parte 2 de 3).

```

77     return xVelocity;
78 }
79
80 // devolve a velocidade em y de MovingPanel
81 public double getYVelocity()
82 {
83     return yVelocity;
84 }
85 }

```

Fig. 22.13 A classe **MovingPanel** representa e exibe um objeto do modelo em movimento (parte 3 de 3).

O método **animate** (linhas 35 a 53) move o **MovingPanel** de acordo com os valores atuais dos atributos **xVelocity** e **yVelocity**. Se a variável booleana **moving** (linha 16) for **true**, as linhas 38 a 44 usam os atributos **xVelocity** e **yVelocity** para determinar a próxima posição para o **MovingPanel**. As linhas 47 a 52 repetem o processo para qualquer filho. Em nossa simulação, **ElevatorView** invoca o método **animate** e o método **paintComponent** da classe **ImagePanel** a cada 50 milissegundos. Estas chamadas em rápida sucessão movem o objeto **MovingPanel**.

Animated Panel

A classe **AnimatedPanel** (Fig. 22.14), que estende a classe **MovingPanel**, representa um objeto animado do modelo (i.e., objetos em movimento cuja imagem correspondente muda continuamente), como uma **Person**. A **ElevatorView** anima um objeto **AnimatedPanel** mudando a imagem associada com **imageIcon**.

```

1  // AnimatedPanel.java
2  // Subclasse de MovingPanel com capacidade de animação
3  package com.deitel.jhttp4.elevator.view;
4
5  // Pacotes do núcleo de Java
6  import java.awt.*;
7  import java.util.*;
8
9  // Pacotes de extensão de Java
10 import javax.swing.*;
11
12 public class AnimatedPanel extends MovingPanel {
13
14     // o ImageIcon deve reciclar as frames
15     private boolean animating;
16
17     // taxa de reciclagem das frames (i.e., taxa de avanço para a próxima frame)
18     private int animationRate;
19     private int animationRateCounter;
20     private boolean cycleForward = true;
21
22     // ImageIcons individuais usados para as frames de animação
23     private ImageIcon imageIcons[];
24
25     // armazenamento para todas as seqüências de frames
26     private java.util.List frameSequences;
27     private int currentAnimation;
28

```

Fig. 22.14 A classe **AnimatedPanel** representa e exibe um objeto animado do modelo (parte 1 de 4).

```

29 // deve repetir (continuar) animação no fim do ciclo?
30 private boolean loop;
31
32 // a animação deve exibir a última frame no fim da animação?
33 private boolean displayLastFrame;
34
35 // ajuda a determinar a próxima frame a exibir
36 private int currentFrameCounter;
37
38 // construtor recebe um array de nomes de arquivos e posições na tela
39 public AnimatedPanel( int identifier, String imageName[] )
40 {
41     super( identifier, imageName[ 0 ] );
42
43     // cria objetos ImageIcon a partir do array de strings imageName
44     imageIcons = new ImageIcon[ imageName.length ];
45
46     for ( int i = 0; i < imageIcons.length; i++ ) {
47         imageIcons[ i ] = new ImageIcon(
48             getClass().getResource( imageName[ i ] ) );
49     }
50
51     frameSequences = new ArrayList();
52
53 } // fim do construtor AnimatedPanel
54
55 // atualiza posição do ícone e frame da animação
56 public void animate()
57 {
58     super.animate();
59
60     // reproduz próxima frame da animação se countador > taxa de animação
61     if ( frameSequences != null && isAnimating() ) {
62
63         if ( animationRateCounter > animationRate ) {
64             animationRateCounter = 0;
65             determineNextFrame();
66         }
67         else
68             animationRateCounter++;
69     }
70 } // fim do método animate
71
72 // determina a próxima frame da animação
73 private void determineNextFrame()
74 {
75     int frameSequence[] =
76         ( int[] ) frameSequences.get( currentAnimation );
77
78     // se não há mais frames de animação, determina a frame final,
79     // a menos que seja especificada repetição
80     if ( currentFrameCounter >= frameSequence.length ) {
81         currentFrameCounter = 0;
82
83         // se isLoop for false, termina animação
84         if ( !isLoop() ) {
85
86             setAnimating( false );
87

```

Fig. 22.14 A classe `AnimatedPanel` representa e exibe um objeto animado do modelo (parte 2 de 4).

```

88         if ( isDisplayLastFrame() )
89             // exibe última frame da sequência
90             currentFrameCounter = frameSequence.length - 1;
91         }
92     }
93 }
94
95 // configura frame de animação atual
96 setCurrentFrame( frameSequence[ currentFrameCounter ] );
97 currentFrameCounter++;
98
99 // fim do método determineNextFrame
100
101 // adiciona sequência de frames (animação) à ArrayList frameSequences
102 public void addFrameSequence( int frameSequence[] )
103 {
104     frameSequences.add( frameSequence );
105 }
106
107 // pergunta se AnimatedPanel está sendo animado (reciclando frames)
108 public boolean isAnimating()
109 {
110     return animating;
111 }
112
113 // configura AnimatedPanel para fazer animação
114 public void setAnimating( boolean animate )
115 {
116     animating = animate;
117 }
118
119 // configura o ImageIcon atual
120 public void setCurrentFrame( int frame )
121 {
122     setIcon( imageIcons[ frame ] );
123 }
124
125 // configura a taxa de animação
126 public void setAnimationRate( int rate )
127 {
128     animationRate = rate;
129 }
130
131 // obtém a taxa de animação
132 public int getAnimationRate()
133 {
134     return animationRate;
135 }
136
137 // configura se a animação deve ser repetida
138 public void setLoop( boolean loopAnimation )
139 {
140     loop = loopAnimation;
141 }
142
143 // verifica se a animação deve ser repetida
144 public boolean isLoop()
145 {
146     return loop;

```

Fig. 22.14 A classe `AnimatedPanel` representa e exibe um objeto animado do modelo (parte 3 de 4).

```

147     }
148
149     // verifica se deve exibir última frame no fim da animação
150     private boolean isDisplayLastFrame()
151     {
152         return displayLastFrame;
153     }
154
155     // configura se deve exibir última frame no fim da animação
156     public void setDisplayLastFrame( boolean displayFrame )
157     {
158         displayLastFrame = displayFrame;
159     }
160
161     // começa a reproduzir a sequência de animação do índice determinado
162     public void playAnimation( int frameSequence )
163     {
164         currentAnimation = frameSequence;
165         currentFrameCounter = 0;
166         setAnimating( true );
167     }
168 }

```

Fig. 22.14 A classe **AnimatedPanel** representa e exibe um objeto animado do modelo (parte 4 de 4).

A classe **AnimatedPanel** escolhe o objeto **ImageIcon** a ser desenhado na tela entre diversos objetos **ImageIcon** armazenados no array **imageIcons** (linha 23). A classe **AnimatedPanel** determina o objeto **ImageIcon** de acordo com uma série de referências *seqüência de frames*, armazenadas na **List frameSequences** (linha 26). A seqüência de *frames* é um array de inteiros que guarda a seqüência apropriada para exibir os objetos **ImageIcon**; especificamente, cada inteiro representa o índice de um objeto **ImageIcon** em **imageIcons**. A Fig. 22.15 demonstra o relacionamento entre **imageIcons** e **frameSequences** (isto não é um diagrama da UML). Por exemplo, a seqüência de *frames* número

2 = { 2, 1, 0 }

se refere a { **imageIcon**[2], **imageIcon**[1], **imageIcon**[0] }, que leva à seqüência de imagens { C, B, A }. Na visão, cada imagem é um arquivo .png único. O método **addFrameSequence** (linhas 102 a 105) adiciona uma seqüência de *frames* à **List frameSequences**. O método **playAnimation** (linhas 162 a 167) inicia a animação associada com o parâmetro **frameSequence**. Por exemplo, considere um objeto **AnimatedPanel** chamado **personAnimatedPanel** na classe **ElevatorView**. O segmento de código

```
animatedPanel.playAnimation( 1 );
```

geraria a seqüência de imagens { A, B, D, B, A } a Fig. 22.15 como referência.

O método **animate** (linhas 56 a 70) sobreescreve o método **animate** da superclasse **MovingPanel**. As linhas 61 a 69 determinam a próxima *frame* da animação dependendo do atributo **animationRate**, que é inversamente proporcional à velocidade de animação – um valor mais alto para **animationRate** leva a uma taxa de *frames* mais lenta. Por exemplo, se **animationRate** for 5, **animate** passa para a próxima *frame* da animação a cada cinco vezes em que é invocado. Usando esta lógica, a taxa de animação é maximizada quando **animationRate** tem um valor de 1, porque a próxima *frame* é determinada cada vez que **animate** é executado.

O método **animate** chama **determineNextFrame** (linhas 73 a 99) para determinar a próxima *frame* (imagem) a ser exibida – especificamente, ele chama o método **setCurrentFrame** (linhas 120 a 123), que configura **imageIcon** (a imagem exibida atual) para a imagem devolvida da seqüência de *frames* atual. As linhas 84 a 92 de **determineNextFrame** são usadas para fins de “repetição” na animação. Se **loop** for **false**, a animação termina após uma iteração. A última *frame* na seqüência é exibida se **displayLastFrame** for **true**, e a primeira *frame* na seqüência é exibida se **displayLastFrame** for **false**. Explicamos em maiores detalhes no Apêndice I como **ElevatorView** usa **displayLastFrame** para os **AnimatedPanels** **Person** e **Door**.

para assegurar a exibição apropriada da imagem. Se **loop** for **true**, a animação é repetida até ser explicitamente parada.

Sound Effects

Discutimos agora como geramos áudio em nossa simulação do elevador. Dividimos a funcionalidade de áudio entre duas classes – **SoundEffects** e **ElevatorMusic** (estas classes não fazem parte dos pacotes de Java, embora **SoundEffects** use o pacote **java.applet** e **ElevatorMusic** use o pacote **javax.sound.midi**). A classe **SoundEffects** (Fig. 22.16) transforma os arquivos *audio* (.au) e *wave* (.wav), que contêm sons como o toque da campainha e os passos da pessoa, em objetos **java.applet.AudioClip**. No Apêndice I, listamos todos os **AudioClips** usados em nossa simulação. A classe **ElevatorMusic** (Fig. 22.17) reproduz um arquivo MIDI (.mid) quando a pessoa anda no elevador. O objeto **ElevatorView** irá reproduzir os objetos **AudioClip** e **ElevatorMusic** para gerar som. Todos os arquivos de som estão na estrutura do diretório

`com/deitel/jhttp4/elevator/view/sounds`

(i.e., o diretório **sounds** no qual as classes para a visão estão localizadas no sistema de arquivos). Em nossa simulação, usamos sons e arquivos **MIDI** fornecidos gratuitamente para *download* pela Microsoft no *site*:

`msdn.microsoft.com/downloads/default.asp`

Para baixar estes sons, clique em “*Graphics and Multimedia*”, “*Multimedia (General)*” e, depois, “*Sounds*”.

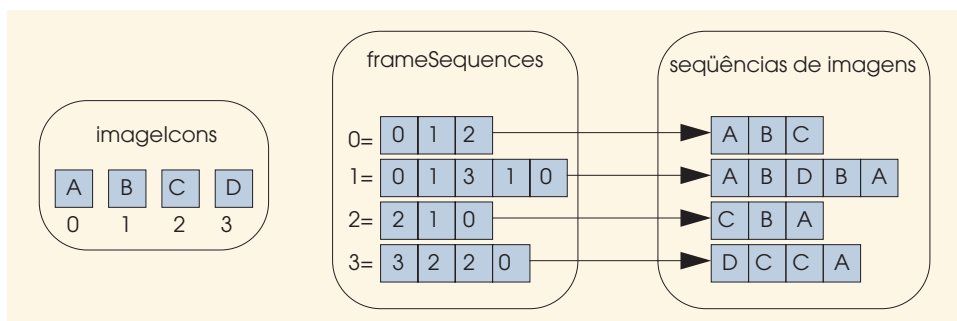


Fig. 22.15 Relacionamento entre o *array* `imageIcons` e a *List* `frameSequences`.

```

1  // SoundEffects.java
2  // Devolve objetos AudioClip
3  package com.deitel.jhttp4.elevator.view;
4
5  // Pacotes do núcleo de Java
6  import java.applet.*;
7
8  public class SoundEffects {
9
10     // localização dos arquivos de som
11     private String prefix = "";
12
13     public SoundEffects() {}
14
15     // obtém AudioClip associado com soundFile
16     public AudioClip getAudioClip( String soundFile )
17     {
18         try {

```

Fig. 22.16 A classe **SoundEffects** devolve objetos **AudioClip** (parte 1 de 2).

```

19         return Applet.newAudioClip( getClass().getResource(
20             prefix + soundFile ) );
21     }
22
23     // devolve null se soundFile não existe
24     catch ( NullPointerException nullPointerException ) {
25         return null;
26     }
27 }
28
29 // configura o prefixo do endereço de soundFile
30 public void setPathPrefix( String string )
31 {
32     prefix = string;
33 }
34 }

```

Fig. 22.16 A classe `SoundEffects` devolve objetos Courier (parte 2 de 2).

A classe `SoundEffects` contém o método `getAudioClip` (linhas 16 a 27), que usa o método `static newAudioClip` (da classe `java.applet.Applet`) para devolver um objeto `AudioClip` com o parâmetro `soundFile`. O método `setPrefix` (linhas 30 a 33) permite que se mude o diretório de um arquivo de som (útil se quisermos dividir nossos sons entre diversos diretórios).

```

1  // ElevatorMusic.java
2  // Permite usar recursos de reprodução de MIDI
3  package com.deitel.jhttp4.elevator.view;
4
5  // Pacotes do núcleo de Java
6  import java.io.*;
7  import java.net.*;
8
9  // Pacotes de extensão de Java
10 import javax.sound.midi.*;
11
12 public class ElevatorMusic implements MetaEventListener {
13
14     // seqüenciador MIDI
15     private Sequencer sequencer;
16
17     // a música deve parar de tocar?
18     private boolean endOfMusic;
19
20     // nome do arquivo de som
21     private String fileName;
22
23     // seqüência associada com o arquivo de som
24     private Sequence soundSequence;
25
26     // construtor abre um arquivo MIDI para reproduzir
27     public ElevatorMusic( String file )
28     {
29         // configura seqüenciador
30         try {
31             sequencer = MidiSystem.getSequencer();
32             sequencer.addMetaEventListener( this );
33             fileName = file;

```

Fig. 22.17 A classe `ElevatorMusic` toca música quando uma `Person` anda no Elevator (parte 1 de 3).

```

34     }
35
36     // trata exceção se MIDI não estiver disponível
37     catch ( MidiUnavailableException midiException ) {
38         midiException.printStackTrace();
39     }
40 } // fim do construtor ElevatorMusic
41
42 // abre arquivo de música
43 public boolean open()
44 {
45     try {
46
47         // obtém URL para o arquivo de mídia
48         URL url = getClass().getResource( fileName );
49
50         // obtém arquivo MIDI válido
51         soundSequence = MidiSystem.getSequence ( url );
52
53         // abre seqüenciador para arquivo especificado
54         sequencer.open();
55         sequencer.setSequence( soundSequence );
56     }
57
58     // trata exceção se URL não existir
59     catch ( NullPointerException nullPointerException ) {
60         nullPointerException.printStackTrace();
61         return false;
62     }
63
64     // trata exceção se os dados MIDI forem inválidos
65     catch ( InvalidMidiDataException midiException ) {
66         midiException.printStackTrace();
67         soundSequence = null;
68         return false;
69     }
70
71     // trata exceção de E/S
72     catch ( java.io.IOException ioException ) {
73         ioException.printStackTrace();
74         soundSequence = null;
75         return false;
76     }
77
78     // trata exceção se MIDI não estiver disponível
79     catch ( MidiUnavailableException midiException ) {
80         midiException.printStackTrace();
81         return false;
82     }
83
84     return true;
85 }
86
87 // reproduz trilha MIDI
88 public void play()
89 {
90     sequencer.start();
91     endOfMusic = false;
92 }

```

Fig. 22.17 A classe `ElevatorMusic` toca música quando uma `Person` anda no `Elevator` (parte 2 de 3).

```

93
94     // obtém sequenciador
95     public Sequencer getSequencer()
96     {
97         return sequencer;
98     }
99
100    // trata fim de trilha
101    public void meta( MetaMessage message )
102    {
103        if ( message.getType() == 47 ) {
104            endOfMusic = true;
105            sequencer.stop();
106        }
107    }
108 }

```

Fig. 22.17 A classe **ElevatorMusic** toca música quando uma **Person** anda no **Elevator** (parte 3 de 3).

Como discutimos na Seção 22.7, Java 2 oferece suporte a **MIDI**. A classe **ElevatorMusic** usa o pacote **javax.sound.midi** para reproduzir o arquivo MIDI. A classe **ElevatorMusic** espera por um evento **MetaMessage** do arquivo MIDI. O sequenciador gera um evento **MetaMessage**. O construtor da classe **ElevatorMusic** (linhas 27 a 40) inicializa o sequenciador MIDI do sistema e registra a classe **ElevatorMusic** para eventos **MetaMessage** do sequenciador. O método **open** (linhas 43 a 85) abre o sequenciador para um arquivo especificado e assegura que os dados MIDI são válidos. O método **play** (linhas 88 a 92) inicia o sequenciador e reproduz o arquivo MIDI.

Conclusão

Você completou um processo de projeto orientado a objetos (OOD) substancial que tinha por objetivo ajudar a prepará-lo para os desafios dos projetos “empresariais”. Esperamos que você tenha achado as seções opcionais “Pensando em objetos” informativas e úteis como suplemento para o material apresentado nos capítulos. Além disso, esperamos que você tenha gostado de projetar o sistema do elevador com a UML. A indústria mundial de *software* adotou a UML como padrão *de facto* para a modelagem de *software* orientado a objetos. Embora tenhamos completado o processo de projeto, simplesmente “arranhamos a superfície” do processo de implementação. Recomendamos enfaticamente que você leia os Apêndices G, H e I no CD que acompanha o livro, que implementam completamente o projeto. Estes apêndices traduzem os diagramas da UML em um programa Java de 3.465 linhas para a simulação do elevador. Nestes apêndices, apresentamos todo o código que não cobrimos nas seções “Pensando em objetos”.

1. O Apêndice G apresenta os arquivos Java que implementam eventos e ouvintes;
2. O Apêndice H apresenta os arquivos Java que implementam o modelo;
3. O Apêndice I apresenta os arquivos Java que implementam a visão.

Não apresentamos muitas novidades nem muito projeto em UML nestes apêndices – eles simplesmente servem para implementar os diagramas baseados em UML que apresentamos nos capítulos anteriores como um programa completo e que funciona. Estudar a implementação nos apêndices deve aprimorar as habilidades em programação que você desenvolveu ao longo do livro e reforçar sua compreensão do processo de projeto.

Resumo

- Através da API JMF, os programadores podem criar aplicativos Java que reproduzem, editam, fazem *streaming* e capturam muitos tipos de mídia populares e de alta qualidade.
- O JMF 2.1.1 suporta tipos de arquivos de mídia populares, como Microsoft Audio/Video Interleave (**.avi**), filmes Macromedia Flash 2 (**.swf**), áudio MPEG Layer 3 (**.mp3**), Musical Instrument Digital Interface (MIDI; **.mid**), vídeos MPEG-1 (**.mpeg**, **.mpg**), QuickTime (**.mov**) e Sun Audio (**.au**).

- A API Java Sound e seus extensos recursos de processamento de som. Java Sound é uma API de nível mais baixo que suporta muitos dos recursos de áudio internos do JMF.
- O **Player** é um tipo de **Controller** em JMF que pode processar e reproduzir clipes de mídia. Reproduzir clipes de mídia com a interface **Player** pode ser tão simples quanto especificar a fonte de mídia, criar um **Player** para a mídia, obter os componentes GUI para a mídia de saída e controles de **Player** e exibi-los. Além disso, os **Players** podem acessar mídia de um dispositivo de captura como um microfone e de um fluxo de *Real-time Transport Protocol* (RTP) – um fluxo de *bytes* enviados através de uma rede que pode ser colocado em um *buffer* e reproduzido no computador cliente.
- Reproduzir mídia envolve acessá-la, criar-lhe um **Controller** e enviá-la para a saída. Antes de enviar a mídia para a saída, existe a opção de formatá-la.
- JMF fornece geradores de vídeo peso-leve, compatíveis com os componentes Swing, usando o método **setHint** de **Manager** com os parâmetros **Manager.LIGHTWEIGHT_RENDERER** e **Boolean.TRUE**.
- O **MediaLocator** é semelhante a um URL, mas ele também suporta endereços de sessão de *streaming* em RTP e endereços de dispositivos de captura.
- Invoke o método **createPlayer** de **Manager** para criar um objeto **Player** que faz referência a um reprodutor de mídia. O método **createPlayer** abre a fonte de mídia especificada e determina o reprodutor apropriado para a fonte de mídia. Ocorre uma **NoPlayerException** se nenhum reprodutor apropriado para o clipe de mídia puder ser encontrado.
- A classe **Manager** fornece métodos **static** que habilitam os programas a acessar a maioria dos recursos do JMF.
- Ao longo de todo o processo de manipulação de mídia, os **Players** geram **ControllerEvents** que são esperados pelos **ControllerListeners**. A classe **ControllerAdapter** implementa os métodos da interface **ControllerListener**.
- Os **Controllers** usam transições de estado para confirmar sua posição no algoritmo de processamento de mídia.
- O método **realize** de **Player** confirma todos os recursos necessários para reproduzir mídia. O método **realize** coloca o **Player** em um estado *Realizing*, no qual o **Player** interage com suas fontes de mídia. Quando o **Player** completa a realização, ele gera um **RealizeCompletedEvent** – um tipo de **ControllerEvent** que ocorre quando o **Player** completa sua transição para o estado *Realized*.
- O método **prefetch** de **Player** faz com que o **Player** obtenha recursos de *hardware* para reproduzir a mídia e comece a colocar os dados da mídia no *buffer*. Colocar os dados da mídia em um *buffer* reduz o retardo antes que o clipe de mídia seja reproduzido, porque a leitura da mídia pode consumir um tempo muito longo.
- Invoke o método **getVisualComponent** de **Player** para obter o componente visual de um vídeo. Invoke o método **getControlPanelComponent** de **Player** para obter de volta os componentes GUI de controle do **Player**.
- Quando o clipe de mídia termina, o **Player** gera um **ControllerEvent** do tipo **EndOfMediaEvent**.
- O método de **Player** **setMediaTime** configura a posição da mídia para um tempo específico na mídia.
- Invocar o método **start** de **Player** inicia a reprodução da mídia. e também coloca dados no *buffer* e realiza o **Player** se isto ainda não foi feito.
- Os dispositivos de captura como microfones têm a capacidade de converter mídia analógica digitalizada. Este tipo de mídia é conhecida como mídia capturada.
- A classe **DataSource** abstrai a fonte de mídia para permitir que um programa a manipule e fornece uma conexão com a fonte de mídia.
- A interface **Processor** permite que um programa manipule dados nos diversos estágios de processamento. Ela estende a interface **Player** e fornece mais controle sobre o processamento da mídia.
- A monitoração permite que você escute ou veja a mídia capturada à medida que é capturada e salva. O **MonitorControl** e outros objetos de controle podem ser obtidos de **Controller** invocando o método **getControl**.
- JMF fornece a classe **Format** para descrever os atributos de um formato de mídia, como a taxa de amostragem (que controla a qualidade do som) e se a mídia deve ser em formato mono ou estéreo. Os objetos **FormatControl** nos permitem formatar os objetos que suportam controles de formato.
- A classe **CaptureDeviceManager** permite que um programa acesse informações sobre o dispositivo de captura.
- O objeto **CaptureDeviceInfo** fornece as informações essenciais necessárias sobre a **DataSource** de um dispositivo de captura.
- Invoke o método **createDataSource** de **Manager** para obter o objeto **DataSource** daquele endereço de mídia.
- O método **createRealizedProcessor** cria um objeto **Processor** realizado que pode começar a processar dados da mídia. O método exige como argumento um objeto **ProcessorModel** que contém as especificações do **Processor**.

- Use um **ContentDescriptor** para descrever o tipo de conteúdo de saída de um **Processor**. **FileTypeDescriptor** especifica o conteúdo de um arquivo de mídia.
- Chame o método **getTrackControls** de **Processor** para obter os controles de cada trilha.
- O objeto que implementa a interface **DataSink** permite que os dados de mídia sejam enviados para a saída em um endereço específico – na maioria das vezes um arquivo. O método **createDataSink** de **Manager** recebe a **DataSource** e o **MediaLocator** como argumentos para criar um objeto **DataSink**.
- Registre um **DataSinkListener** para esperar **DataSinkEvents** gerados por um **DataSink**. O programa pode chamar o método **dataSinkUpdate** de **DataSinkListener** quando ocorre cada **DataSinkEvent**. O **DataSink** causa um **EndOfStreamEvent** quando a conexão com o *stream* de captura fecha.
- Mídia de *streaming* se refere à mídia que é transferida de um servidor para um cliente em um fluxo contínuo de *bytes*. A tecnologia de mídia de *streaming* carrega dados da mídia em *buffers* antes de exibir a mídia.
- O JMF fornece um pacote para mídia de *streaming* que permite aos aplicativos Java enviar e receber streams de mídia nos formatos discutidos anteriormente neste capítulo. JMF usa o padrão Real-Time Transport Protocol (RTP) para controlar a transmissão da mídia. O RTP foi projetado especificamente para transmitir dados de mídia em tempo real.
- Use um **DataSink** ou um **RTPManager** para fazer *streaming* de mídia. Os **RTPManagers** oferecem mais controle e versatilidade para a transmissão. Se um aplicativo envia múltiplos *streams*, o aplicativo deve ter um **RTPManager** para cada sessão de *streaming* separada. Ambos requerem a **DataSource** obtida do método **getOutput** de **Processor**.
- O URL de *streams* de RTP está no formato: **rtp://<host>:<porta>/<tipoDeConteúdo>**
- A formatação da mídia só pode ser feita quando o **Processor** foi configurado. Para notificar o programa quando ele completa a configuração do **Processor**, registre um **ControllerListener** para notificar o programa de que ele completou a configuração. Ocorre um **ConfigureCompleteEvent** quando o **Processor** completa a configuração.
- O método **setContentDescriptor** de **Processor** configura o *stream* para um formato habilitado para RTP com o parâmetro **ContentDescriptor.RAW RTP**.
- A interface **TrackControl** permite que os formatos das trilhas da mídia sejam configurados.
- **SessionAddress** contém o endereço IP e o número de porta usados no processo de *streaming*. Os **RTPManagers** usam **SessionAddresses** para fazer *streaming* de mídia.
- Invoke o método **initialize** de **RTPManager** para inicializar a sessão de *streaming* local com o endereço de sessão local como parâmetro. Invoke o método **addTarget** de **RTPManager** para adicionar o endereço de sessão de destino como cliente que recebe o *stream* de mídia. Para fazer *streaming* de mídia para vários clientes, chame o método **addTarget** de **RTPManager** para cada endereço de destino.
- O método **removeTargets** de **RTPManager** fecha o *streaming* para destinos específicos. O método **dispose** de **RTPManager** libera os recursos mantidos pelas sessões de RTP.
- A API Java Sound fornece classes e interfaces para acessar, manipular e reproduzir áudio *Musical Instrument Digital Interface* (MIDI) e áudio amostrado.
- É necessária uma placa de som para reproduzir áudio com o Java Sound. o Java Sound dispara exceções quando ela acessa recursos de áudio do sistema para processar áudio em um computador que não tem uma placa de som.
- Os programadores podem usar o pacote **javax.sound.sampled** para reproduzir os arquivos em formatos de amostra de áudio, que incluem *Sun Audio* (**.au**), *Wave* (**.wav**) e *AIFF* (**.aiff**).
- Para processar dados de áudio, podemos usar uma linha de **Clip** que permite o fluxo de dados digitais brutos para dados de áudio que podemos escutar.
- O objeto **AudioInputStream** aponta para o *stream* de áudio. A classe **AudioInputStream** (uma subclasse de **InputStream**) fornece acesso ao conteúdo do *stream* de áudio.
- O tamanho de cliques de áudio e vídeo é medido em *frames*. Cada *frame* representa dados em um intervalo de tempo específico no arquivo de áudio.
- O algoritmo para reproduzir amostra de áudio suportado por Java Sound é o seguinte: obter um **AudioInputStream** de um arquivo de áudio, obter uma linha de **Clip** formatada, carregar o **AudioInputStream** na linha de **Clip**, iniciar o fluxo de dados na linha de **Clip**.
- Todas as **Lines** geram **LineEvents** que podem ser tratados por **LineListener**. A primeira etapa para reproduzir amostras de áudio envolve obter o *stream* de áudio a partir de um arquivo de áudio.
- A classe **AudioSystem** permite que um programa acesse muitos recursos de áudio do sistema necessários para reproduzir e manipular arquivos de som.
- O método **getAudioInputStream** dispara uma **UnsupportedAudioFileException** se o arquivo de som especificado não for um arquivo de áudio ou contiver um formato de clipe de som que não é suportado por Java Sound.

- O método `getLine` exige um objeto `Line.Info` como argumento, que especifica os atributos da linha que o `AudioSystem` deve obter.
- Podemos usar um objeto `DataLine.Info` que especifica uma linha de dados `Clip`, um formato genérico de codificação e um intervalo de *buffer*. Precisamos especificar um intervalo de *buffer* para que o programa possa determinar o melhor tamanho de *buffer* a partir de um intervalo preferencial dado.
- Os objetos `DataLine.Info` especificam informações sobre uma linha de `Clip`, como os formatos suportados pelo `Clip`. O construtor do objeto `DataLine.Info` recebe como argumentos a classe `Line`, os `AudioFormats` suportados pela linha, o tamanho de *buffer* mínimo e o tamanho de *buffer* máximo, em *bytes*.
- O método `getLine` de `AudioSystem` e o método `open` de `Clip` disparam `LineUnavailableExceptions` se outro aplicativo estiver usando o recurso de áudio solicitado. O método `open` de `Clip` também dispara uma `IOException` se `open` não consegue ler o `AudioInputStream` especificado.
- Invoque o método `start` de `Clip` para iniciar a reprodução de áudio.
- Quando ocorre um `LineEvent`, o programa chama o método `update` de `LineListener` para processar o evento. Os quatro tipos de `LineEvent` estão definidos na classe `LineEvent.Type`. Os tipos de eventos são `CLOSE`, `OPEN`, `START` e `STOP`.
- O método `close` da classe `Line` suspende a atividade de áudio e fecha a linha – o que libera quaisquer recursos para áudio obtidos anteriormente pela `Line`.
- O método `loop` de `Clip` pode ser chamado com o parâmetro `Clip.LOOP_CONTINUOUSLY` para repetir o clipe de áudio para sempre.
- Música MIDI (Musical Instrument Digital Interface) pode ser criada através de um instrumento digital, como um teclado eletrônico (sintetizador) ou através de sintetizadores em pacotes de *software*. O sintetizador MIDI é um dispositivo que pode produzir sons e música MIDI.
- A especificação MIDI fornece informações detalhadas sobre os formatos de um arquivo MIDI. Para obter informações detalhadas sobre MIDI e sua especificação, visite o seu *site* oficial na Web, no endereço www.midi.org. O pacote MIDI de Java Sound permite que os desenvolvedores acessem os dados que especificam o MIDI, mas ele não fornece suporte para a especificação.
- A interpretação de dados MIDI varia entre sintetizadores e irá soar diferente com instrumentos diferentes. O pacote `javax.sound.midi` permite que o programa manipule, reproduza e sintetize MIDI. Existem três tipos de MIDI – 0 (o mais comum), 1 e 2. O Java Sound suporta arquivos MIDI com as extensões `.mid` e `.rmf` (Rich Music Format).
- Alguns reconhecedores de arquivos em diversos sistemas operacionais são incapazes de interpretar o arquivo MIDI como um arquivo MIDI que Java pode reproduzir.
- A reprodução de MIDI é executada por um seqüenciador MIDI. Especificamente, os seqüenciadores podem reproduzir e manipular uma seqüência MIDI, que é a fórmula de dados que diz a um dispositivo como tratar os dados MIDI.
- Frequentemente, o MIDI é conhecido como uma seqüência, porque os dados musicais em MIDI são compostos por uma seqüência de eventos. A simplicidade dos dados de MIDI nos permite visualizar cada evento individualmente e aprender o propósito de cada evento. O processo de reprodução de MIDI envolve acessar um seqüenciador, carregar uma seqüência MIDI ou um arquivo MIDI em um seqüenciador e iniciar o seqüenciador.
- O método `getSequence` também pode obter uma seqüência MIDI de um URL ou um `InputStream`. O método `getSequence` dispara uma `InvalidMidiDataException` se o sistema MIDI detectar um arquivo MIDI incompatível.
- A interface `Sequencer`, que estende a interface `MidiDevice` (a super-interface para todos os dispositivos MIDI), representa o dispositivo-padrão para reproduzir dados MIDI.
- O método `open` de `Sequencer` prepara a reprodução de uma `Sequence`. O método `setSequence` de `Sequencer` carrega uma `Sequence` MIDI no `Sequencer` e dispara uma `InvalidMidiException` se o `Sequencer` detectar uma seqüência MIDI irreconhecível. O método `play` de `Sequencer` inicia a reprodução da seqüência MIDI.
- A trilha MIDI é uma seqüência de dados gravada; as MIDIs normalmente contêm múltiplas trilhas. As trilhas MIDI são semelhantes às trilhas de CD, exceto pelo fato de que os dados de música em MIDI são reproduzidos simultaneamente. A classe `Track` (pacote `javax.sound.midi`) fornece acesso aos dados de música MIDI armazenados nas trilhas MIDI.
- Os dados MIDI em trilhas MIDI são representados por eventos MIDI. Os eventos MIDI guardam a ação de MIDI e o tempo em que o comando MIDI deve ocorrer. Existem três tipos de mensagens MIDI – `ShortMessage`, `SysexMessage` e `MetaMessage`. `ShortMessages` fornecem instruções, como notas específicas a tocar, e podem configurar opções, como quando inicia uma MIDI. As outras duas mensagens, menos usadas, são mensagens exclusivas do sistema chamadas `SysexMessage` e `MetaMessages`, que podem dizer a um dispositivo que a MIDI atingiu o

fim de uma trilha. Esta seção trata exclusivamente de **ShortMessages** que tocam notas específicas. Cada **MidiMessage** é encapsulada em um **MidiEvent** e uma sequência de **MidiEvents** forma uma trilha MIDI.

- Cada método **getTick** de **MidiEvent** fornece o tempo em que o evento acontece (*time stamp*).
- O método **getCommand** de **ShortMessage** devolve o inteiro que representa o comando da mensagem. O método **getData1** de **ShortMessage** devolve o primeiro *byte* de estado da mensagem. O método **getData2** de **ShortMessage** devolve o segundo *byte* de estado. O primeiro e o segundo *bytes* de estado variam de interpretação de acordo com o tipo de comando na **ShortMessage**.
- A gravação de MIDI genérica é executada através de um sequenciador. A interface **Sequencer** fornece métodos simples para gravar – supondo que os transmissores e os receptores dos dispositivos MIDI estejam “conectados” corretamente.
- Depois de configurar um sequenciador e uma sequência vazia, o objeto **Sequencer** pode invocar seu método **startRecording** para habilitar e iniciar a gravação na trilha vazia. O método **recordEnable** da interface **Sequencer** recebe um objeto **Track** e um número de canal como parâmetros para permitir a gravação em uma trilha.
- O método **write** da classe **MidiSystem** escreve a sequência em um arquivo especificado.
- Um método alternativo para gravar MIDI sem ter que lidar com transmissores e receptores é criar eventos a partir de **ShortMessages**. Os eventos devem ser adicionados a uma **Track** de uma **Sequence**.
- A interface **Synthesizer** é uma interface **MidiDevice** que habilita o acesso à geração de som MIDI, aos instrumentos, aos recursos de canais e aos bancos de sons.
- O **SoundBank** é o contêiner para diversos **Instruments**, que dizem ao computador como emitir uma nota específica e são algoritmos programados com instruções. Notas diferentes em diversos instrumentos são reproduzidas através de um **MidiChannel** em diferentes trilhas simultaneamente, para produzir melodias sinfônicas.
- A aquisição de quaisquer recursos MIDI dispara uma **MidiUnavailableException** se o recurso não estiver disponível.
- Invoke o método **getChannels** de **Synthesizer** para obter todos os 16 canais do sintetizador. O **MidiChannel** pode emitir uma nota chamando seu método **noteOn** com o número da nota (0 a 127) e o volume como parâmetros. O método **noteOff** de **MidiChannel** desliga uma nota apenas com o número da nota como parâmetro.
- O método **getAvailableInstruments** de **Synthesizer** obtém os programas do instrumento *default* de um sintetizador. Também se pode importar mais instrumentos carregando um banco de sons personalizado através do método **loadAllInstruments** (**SoundBank**) na interface **Synthesizer**. O banco de sons normalmente tem 128 instrumentos. O método **programChange** de **MidiChannel** carrega o programa do instrumento desejado no sintetizador.
- Invoke o método **send** de um **Receiver** com uma **MidiMessage** e um *time stamp* como parâmetros, para enviar a mensagem MIDI para todos os transmissores.

Terminologia

addControllerListener, método de **Controller**

addDataSinkListener, método de **DataSink**

addLineListener, método de **Line**

addMetaEventListener, método de **Sequencer**

addTarget, método de **RTPManager**

AudioFormat, classe

AudioFormat.Encoding.PCM_SIGNED

AudioInputStream, classe

AudioSystem, classe

banda larga

Boolean.TRUE, codificação

CannotRealizeException, classe

captura

CaptureDevice, interface

CaptureDeviceInfo, classe

CaptureDeviceManger, classe

Clip

Clip.class

Clip.LOOP_CONTINUOUSLY

clipe de mídia

Clock, interface

close, método de **Controller**

close, método de **Line**

close, método de **MidiDevice**

colocar previamente em buffer

configure, método de **Processor**

configureComplete, método de **ControllerAdapter**

ConfigureCompleteEvent, classe

Controller.Prefetching

Controller.Prefetched

Controller.Realized

Controller.Realizing

Controller.Started

Controller.Stopped

ControllerAdapter, classe

ControllerEvent, classe

ControllerListener, interface

createDataSink, método de **Manager**
createDataSource, método de **Manager**
createPlayer, método de **Manager**
createProcessor, método de **Manager**
createRealizedProcessor, método de **Manager**
createSendStream, método de **RTPManager**
createTrack, método de **Sequence**
 dados empacotados
DataLine, interface
DataLine.Info, classe
DataSink, interface
DataSinkEvent, classe
DataSinkListener, interface
dataSinkUpdate, método de **DataSinkListener**
DataSource, classe
deleteTrack, método de **Sequence**
 Direct Sound
dispose, método de **RTPManager**
 dispositivo de captura
 endereço de mídia
endOfMedia, método de **ControllerAdapter**
EndofMediaEvent, classe
EndofStreamEvent, classe
FileTypeDescriptor, classe
FileTypeDescriptor.QUICKTIME
Format, classe
FormatControl, interface
 formato de saída
 frames
get, método de **Track**
getAudioInputStream, método de **AudioSystem**
getBank, método de **Patch**
getChannels, método de **Synthesizer**
getCommand, método de **ShortMessage**
getControlComponent, método de **Control**
getControlPanelComponent, método de **Player**
getData1, método de **ShortMessage**
getData2, método de **ShortMessage**
getDataOutput, método de **Processor**
getDeviceList, método de **CaptureDeviceManager**
getFormat, método de **AudioFormat**
getFormat, método de **FormatControl**
getFormatControls, método de **CaptureDevice**
getFrameLength, método de **AudioInputStream**
getFrameSize, método de **AudioFormat**
getLine, método de **AudioSystem**
getLocator, método de **CaptureDeviceInfo**
getMessage, método de **MidiEvent**
getMidiFileTypes, método de **MidiSystem**
getPatch, método de **Instrument**
getProgram, método de **Patch**
getReceiver, método de **MidiDevice**
getResolution, método de **Sequence**
getSequence, método de **MidiSystem**
getSequencer, método de **MidiSystem**
getSupportedFormats, método de **FormatControl**
getSynthesizer, método de **MidiSystem**
getTargetFormat, método de **AudioSystem**
getTick, método de **MidiEvent**
getTrackControls, método de **Processor**
getTracks, método de **Sequence**
getTransmitter, método de **MidiDevice**
getType, método de **LineEvent**
getVisualComponent, método de **Player**
initialize, método de **RTPManager**
Instrument, classe
 interface
InvalidMidiException, classe
InvalidSessionAddress, classe
isEnabled, método de **FormatControl**
isLineSupported, método de **AudioSystem**
 Java Media Framework
 Java Sound
 javax.media, pacote
 javax.media.control, pacote
 javax.media.datasink, pacote
 javax.media.format, pacote
 javax.media.protocol, pacote
 javax.media.rtp, pacote
 javax.sound.midi, pacote
 javax.sound.sampled, pacote
JOptionPane.CLOSED_OPTION
JOptionPane.DEFAULT_OPTION
JOptionPane.OK_OPTION
Line, interface
LineEvent, classe
LineEvent.Type.STOP
LineListener, interface
LineUnavailableException, classe
 loop, método de **Clip**
Manager, classe
Manger.LIGHTWEIGHT_RENDERER
MediaLocator, classe
 MIDI (Musical Instrument Digital Interface)
 MIDI, especificação
 mídia
 mídia capturada
 mídia de streaming
MidiChannel, interface
MidiEvent, classe
MidiMessage, classe
MidiSystem, classe
MidiUnavailableException, classe
 monitoração
MonitorControl, interface
 MP3
 MPEG-1
newInstance, método de **RTPManager**
NoDataSinkException, classe
NoDataSourceException, classe
NoPlayerException, classe
NoProcessorException, classe
noteOff, método de **MidiChannel**
noteOn, método de **MidiChannel**
 open, método de **Clip**

open, método de **DataSink**
open, método de **MidiDevice**
Patch, classe
pitch
Player, interface
 portas de dispositivo
 portas de rede
prefetchComplete, método de **ControllerAdapter**
PrefetchCompleteEvent, classe
 processar previamente
Processor, interface
Processor.Configured
Processor.Configuring
ProcessorModel, classe
 protocolo
QuickTime
realize, método de **Controller**
realizeComplete, método de **ControllerAdapter**
RealizeCompleteEvent, classe
Receiver, interface
recordEnable, método de **Sequencer**
removeTargets, método de **RTPManager**
 retardo de propagação
RMF (Rich Music Format)
RTP (Real-time Transport Protocol)
RTPManager, classe
SecurityException, classe
send, método de **Receiver**
SendStream, interface
Sequence, classe
Sequence.PPQ
Sequencer, interface
SessionAddress, classe
SessionAddress, classe
SessionEvent, classe
SessionListener, interface
setContentDescriptor, método de **Processor**
setFormat, método de **FormatControl**
setHint, método de **Manager**
setMediaTime, método de **Clock**
setMessage, método de **ShortMessage**
setReceiver, método de **Transmitter**
setSequence, método de **Sequencer**
ShortMessage, classe
ShortMessage.NOTE_OFF
ShortMessage.NOTE_ON
ShortMessage.PROGRAM_CHANGE
 simulação
 sincronização
 síntese
size, método de **track**
SoundBank, interface
start, método de **DataLine**
start, método de **DataSink**
start, método de **Player**
start, método de **SendStream**
start, método de **Sequencer**
startRecord, método de **Sequencer**
stop, método de **DataLine**
stop, método de **DataSink**
stop, método de **Sequencer**
stopRecord, método de **Sequencer**
 streams
Synthesizer, interface
 teleconferência
 tempo
 time stamp
Track, classe
TrackControl, interface
Transmitter, interface
 trilhas de mídia
UnsupportedAudioFileException, classe
UnsupportedFormatException, classe
 videoconferência
Video for Windows
write, método de **MidiSystem**

Exercícios de auto-revisão

- 22.1** Preencha as lacunas em cada uma das frases seguintes:
- A classe _____ fornece acesso a muitos recursos de JMF.
 - Além de endereços de arquivos de mídia armazenados no computador local, o _____ também pode especificar o endereço de dispositivos de captura e sessões de RTP.
 - A classe _____ fornece acesso a recursos do sistema para amostra de áudio, enquanto a classe _____ fornece acesso a recursos do sistema para MIDI.
 - O evento do tipo _____ indica que um **Controller** já estabeleceu comunicação com a fonte de mídia.
 - O método **createRealizedProcessor** recebe um _____ como argumento.
 - Em ordem, os estados de **Processor** são: *Unrealized*, _____, _____, _____, _____, _____ e *Started*.
 - A constante _____ especifica que o **Processor** deve enviar mídia para a saída no formato *QuickTime*.
 - Para fazer *streaming* de mídia, podemos usar um _____ ou um _____.

- i) Os objetos _____ configuram os formatos de *streams* para dispositivos de captura.
- j) Invocar o método _____ de **Clip** com a constante _____ como argumento repete um arquivo de amostra de áudio continuamente.
- k) O _____ MIDI contém múltiplas trilhas, as quais contêm uma sequência de _____ MIDI, e cada uma deles encapsula uma _____ MIDI.

22.2 Diga se cada uma das seguintes afirmações é *verdadeira* ou *falsa*. Se for *falsa*, explique por quê.

- a) O método **setHint** de **Manager** pode ser usado para especificar que o componente visual de um clipe de mídia deve ser gerado com componentes GUI peso leve.
- b) O **ControllerListener** trata de eventos gerados por um **DataSink**.
- c) Somente objetos que implementam a interface **Processor** podem reproduzir mídia.
- d) O **Player** não pode acessar mídia de dispositivos de captura; o **Processor** deve ser usado para esta finalidade.
- e) O **Clip** reproduz **Sequences** de MIDI.
- f) A reprodução de MIDI pára automaticamente quando o **Sequencer** atinge o fim de uma sequência MIDI.
- g) O **RTPManager** pode fazer *streaming* de um arquivo de mídia inteiro, independentemente do número de trilhas no arquivo.
- h) O método **createPlayer** dispara uma **NoDataSourceException** se ele for incapaz de localizar a fonte de dados de mídia especificada.

Respostas aos exercícios de auto-revisão

22.1 a) **Manager**. b) **MediaLocator**. c) **AudioSystem**, **MidiSystem**. d) **RealizeCompleteEvent**. e) **ProcessorModel**. f) *Configuring*, *Configured*, *Realizing*, *Realized*, *Prefetching*, *Prefetched*. g) **FileTypeDescriptor**. **QUICKTIME**. h) **DataSink**, **RTPManager**. i) **FormatControl**. j) **loop**, **Clip.LOOP_CONTINUOUSLY**. k) **Sequence**, eventos, mensagem.

22.2 As respostas do exercício de auto-revisão 3.2 são as seguintes:

- a) Verdadeira.
- b) Falsa. O **DataSinkListener** trata de **DataSinkEvents** gerados por um **DataSink**.
- c) Falsa. Os objetos que implementam **Player** ou **Processor** podem reproduzir mídia.
- d) Falsa. Tanto um **Processor** quanto um **Player** podem acessar mídia de dispositivos de captura.
- e) Falsa. O **Sequencer** reproduz sequências MIDI.
- f) Verdadeira.
- g) Falsa. Cada **RTPManager** pode fazer *streaming* de apenas uma trilha.
- h) Falsa. O método **createPlayer** dispara uma **NoPlayerException** se ele não for capaz de localizar a fonte de dados de mídia especificada.

Exercícios

22.3 Os clipes de áudio *Wave* são comumente usados para reproduzir os sons que alertam o usuário sobre um problema em um programa. Normalmente, tais sons são acompanhados por diálogos de mensagem de erro. Modifique o exemplo **DivideByZeroTest** da Fig. 14.1 para reproduzir um som de mensagem de erro (além de exibir um diálogo de mensagem de erro) se o usuário digitar um inteiro inválido ou tentar dividir por zero. Carregue previamente um clipe de som compatível, usando uma linha de **Clip** como demonstrado na Fig. 22.5. A linha de **Clip** precisa suportar só o formato do clipe de som escolhido. Deve haver um método separado que invoca a reprodução do clipe. Quando o programa detecta uma exceção, ele deve chamar este método para reproduzir o som da mensagem de erro. Após cada reprodução do clipe, o programa precisa “reenrolar” o clipe invocando o método **setFramePosition** de **Clip** com a posição da *frame* como argumento, de modo que o clipe possa ser repetido a partir de sua posição inicial.

22.4 Incorpore recursos de reprodução de arquivos MIDI, como demonstrado na classe **MidiData** (Fig. 22.7), à demonstração **ClipPlayer**. A classe **ClipPlayer** deve ter métodos separados para obter os dados da sequência MIDI e reproduzir a sequência com um sequenciador.

22.5 A demonstração **SimplePlayer** (Fig. 22.7) demonstrou os recursos de JMF para reprodução de mídia (vídeos, mídia capturada) com a interface **Player**. Usando a demonstração **SimplePlayer** como diretriz, desenvolva um aplicativo de karaokê no qual uma parte do programa reproduz um arquivo de música/vídeo (de preferência sem voz) en-

quanto outra parte do programa simultaneamente captura a voz do usuário. O programa deve iniciar a reprodução e a captura assim que obtiver a mídia. É importante que o programa permita ao usuário controlar tanto a captura quanto a música, de modo que as GUIs de controle de cada mídia devem ser exibidas. Quando o arquivo de mídia termina de ser reproduzido, a captura de voz deve terminar e o programa deve posicionar a música novamente no início. O programa deve fechar todos os recursos relacionados com o **Player** quando o usuário termina o programa.

22.6 Modifique a solução do Exercício 3.5 implementando o programa com a interface **Processor**. Crie um **Processor** que está pronto para exibir a mídia sem especificações de formato ou saída. A captura de voz não deve terminar e a mídia não deve voltar para o início quando a reprodução da mídia termina. Libere recursos relacionados com o **Processor** quando o usuário abrir um novo arquivo ou fechar o programa. Todos os outros detalhes do programa permanecem os mesmos, como especificado no Exercício 3.5.

22.7 Usando como referência o processo de salvamento de arquivo demonstrado na classe **CapturePlayer** (Fig. 22.2), modifique a solução do Exercício 22.6 salvando os dois *streams* de áudio em dois arquivos *QuickTime* separados. Especifique que as trilhas de mídia devem estar no formato de codificação **AudioFormat**. **IMA4**. O programa deve exibir diálogos de salvamento de arquivo para cada salvamento de arquivo de áudio e uma mensagem quando o salvamento for completado ou parar. Devem existir gravadores de dados separados para cada *stream* de áudio. O programa deve fechar os gravadores de dados quando não houver mais dados para processar ou quando o usuário terminar o programa. O método **getSourceDataSink** de **DataSinkEvent** está disponível para obter o **DataSink** que está gerando o **DataSinkEvent**. Use **MonitorControls** para monitorar os dois *streams* de áudio, de modo que não haja necessidade de exibir controles de vídeo ou controles *default* do usuário. O método **setEnabled** de **MonitorControl** está disponível para habilitar o monitoramento dos *streams* de áudio. Exiba um dos **MonitorControls** em uma caixa de diálogo. Assegure-se de fechar os recursos do **Processor** quando não houver mais dados no arquivo de mídia ou quando o usuário abrir outro arquivo ou terminar o programa.

22.8 Modifique a classe **MidiSynthesizer** (Fig. 22.9) para um aplicativo em que o usuário pode tocar notas de violino pressionando as teclas do teclado do computador. Use o código virtual das teclas para especificar o número da nota que o sintetizador deve tocar. O número do programa de violino é 40. Use o primeiro e o nono canais para emitir as notas ao mesmo tempo. O nono canal pode gerar uma versão diferente das notas do violino.

Seção especial: projetos desafiadores de multimídia

Os exercícios precedentes são vinculados ao texto e projetados para testar a compreensão do usuário dos conceitos fundamentais de JMF e Java Sound. Esta seção inclui uma coleção de projetos avançados de multimídia. O leitor deve achar estes problemas desafiadores, mas ainda assim divertidos. Os problemas variam consideravelmente em grau de dificuldade. Alguns exigem uma hora ou duas para codificação do programa e implementação. Outros são úteis para trabalhos práticos que podem exigir duas ou três semanas de estudo e implementação. Alguns são projetos desafiadores para um semestre. [Nota: não são fornecidas as soluções para estes exercícios.]

22.9 Modifique a demonstração **ClipPlayer** (Fig. 22.5 e Fig. 22.6) para fornecer uma caixa de marcação de repetição que permita ao usuário repetir a reprodução do arquivo de amostra de áudio.

22.10 Modifique a demonstração **RTPServer** (Fig. 22.3) para permitir a transmissão das partes de áudio dos arquivos de mídia para dois clientes. A classe de teste do aplicativo deve ter o dobro do número de caixas de diálogo de pedido de IP e número da porta. O programa pode verificar os formatos de áudio fazendo com que os formatos de controle de trilha da mídia sejam instâncias de **AudioFormat** (classe **AudioFormat**).

22.11 Muitos *sites* da Web são capazes de reproduzir clipes de vídeo. O programa **SimplePlayer** (Fig. 22.1) pode ser um *applet*. Simplifique o programa **SimplePlayer** para um *applet* que reproduz um clipe de mídia previamente carregado em uma página Web. Insira a marca deste *applet* em seu arquivo HTML:

```
<applet code = NomeDoApplet.class width = # height = # >
  <param name = arquivo value = "exemplo.mov" >
</applet>
```

22.12 Modifique a classe **MidiRecord** (Fig. 22.8) e a classe **MidiData** (Fig. 22.7) para criar uma nova classe de aplicativo que duplica arquivos MIDI gravando a sequência em um novo arquivo. Reproduza a sequência usando **MidiData** e use seu **Transmitter** para transmitir informações MIDI para o **Receiver** de **MidiRecord**.

22.13 Modifique a solução do Exercício 22.6 para criar um aplicativo de karaokê com *streaming* no qual o aplicativo faz *streaming* somente da parte de vídeo de um vídeo de música e o *stream* de som é substituído por um *stream* de captura de voz. [Nota: se o formato de vídeo contiver somente uma trilha tanto para áudio como para vídeo, o aplicativo não pode optar por fazer *streaming* apenas da parte de vídeo da trilha.]

22.14 Modifique a classe **MidiData** (Fig. 22.7) para carregar todas as trilhas de um arquivo MIDI e revise a classe **MidiDemo** (Fig. 22.10) para habilitar o usuário a selecionar a reprodução de cada trilha exibida em um painel seletor **JList**. Permita que o usuário repita a sequência para sempre.

22.15 Implemente um reprodutor de MP3 com uma janela com uma lista de arquivos usando **Vectors** e uma **JList**.

22.16 Modifique a classe **MidiRecord** (Fig. 22.8) e a classe **MidiDemo** (Fig. 22.10) para permitir que o usuário grave MIDI em trilhas armazenadas em um **Vector**. A reprodução do MIDI gravado deve reproduzir todas as trilhas MIDI simultaneamente.

22.17 Atualmente, o programa **MidiDemo** (Seção 22.7) grava música sintetizada com o primeiro instrumento disponível (i.e., *Grand Piano*). Modifique a classe **MidiDemo** de modo que a música seja gravada com um instrumento selecionado pelo usuário e permita mudar o instrumento durante a gravação. Também permita ao usuário importar seus próprios bancos de som. Faça mudanças nas classes **MidiSynthesizer**, **MidiData** e **MidiRecord** conforme necessário. (Dica: o parâmetro de comando para mudar o instrumento é **ShortMessage.PROGRAM_CHANGE**).

22.18 Modifique a demonstração **SimplePlayer** (Fig. 22.1) para suportar vários reprodutores de mídia. Apresente cada clipe de mídia em seu próprio **JInternalFrame**. O programa precisa criar **Players** separados para cada clipe de mídia e deve registrar um **ControllerListener** para cada **Player**. O método **getSourceController** de **ControllerEvent** está disponível para obter o controlador que gerou o **ControllerEvent**. Implemente o programa com uma estrutura dinâmica de dados, como **Vector**, para armazenar os vários **Players**.

22.19 Modifique a solução do Exercício 3.7 para salvar os dois *streams* de mídia em um arquivo e reproduzir o *stream* combinado. Use o método **createMergingDataSource** de **Manager**, que recebe um *array* de objetos **DataSource**, para salvar tanto o *stream* de captura quanto o *stream* de música em um só *stream*, cujo tipo de conteúdo será **MIXED**. O programa deve obter as **DataSources** de saída dos **Processors** como os objetos **DataSource** a serem intercalados. O programa também deve obter uma **DataSource** duplicata (da **DataSource** intercalada) para criar o **Player** para aquela **DataSource**. Para fazer isto, use o método **createCloneableDataSource** de **Manager** para criar uma **DataSource Cloneable** com a **DataSource** intercalada como argumento. Duplica a **DataSource** para o **Player** invocando o método **createClone** da interface **Cloneable** sobre a **DataSource** (semelhante a se obter os **FormatControls** de uma **DataSource** de **CaptureDevice** na demonstração **CapturePlayer** (Fig. 22.2)).

22.20 Um programa pode gravar MIDI, sem o uso de transmissores e receptores, criando **MidiMessages**, colocando-as em **MidiEvents** e adicionando estes eventos a uma **Track**. Além do argumento **MidiMessage**, um programa deve especificar um *time stamp* para criar o **MidiEvent**, expresso em *ticks* (isto é, milissegundos do tipo **long**), de modo que o programa deve obter a hora atual do sistema em milissegundos, que pode ser obtida do método **currentTimeMillis** de **System**. O método **add** de **Track** está disponível para adicionar eventos a uma trilha. Crie uma mesa acústica (por exemplo, tambores, pratos, etc.) na qual o usuário pode selecionar uma sequência de instrumentos a tocar. Permita que o usuário salve a sequência MIDI gravada em um arquivo.

22.21 Crie um *kit* de teleconferência *peer-to-peer* que habilita os usuários a conversar e ouvir um ao outro. Para ouvir um ao outro, cada usuário deve abrir uma sessão de RTP para o *stream* de captura. O programa pode abrir um *stream* de RTP com um **MediaLocator** especificando o endereço de sessão RTP. A seguir, o programa pode usar o **MediaLocator** para criar um **Player** para o *stream* de RTP. O programa pode enviar a captura de voz como demonstrado na demonstração **RTPManager** (Fig. 22.3). Para enviar a captura de voz para mais de duas pessoas, use **RTPManagers** para cada sessão separada e chame seu método **addTarget** para adicionar o endereço de sessão de cada receptor como o destino do *stream* de captura. Isto é chamado de sessões *multicast-unicast*.

22.22 Usando o acionador de “tocador de piano” (a Seção 22.7 discute o acionador) da classe **MidiDemo** (Fig. 22.10) e as funções de animação de imagens do Capítulo 18, escreva um programa que exibe uma bola que salta cuja altura máxima é especificada pelos números de notas (de uma **ShortMessage** em um **MidiEvent**) de uma trilha carregada em um arquivo MIDI. O programa deve usar os pulsos de duração do **MidiEvent** como duração do salto.

22.23 A maioria dos vídeos de música de karaokê está no formato MPEG, para o qual o JMF fornece uma interface **MpegAudioControl** para controlar os canais de áudio. Para vídeos MPEG em mais de uma língua e karaokê, os canais direcionam o áudio principal para um de dois *streams* de áudio (por exemplo, *stream* de áudio dublado em inglês) ou os dois *streams* de áudio (por exemplo, o canal de música e o de voz ligados em vídeos de karaokê). Como recurso adicional para vídeos MPEG em sua solução do Exercício 3.6, obtenha e exiba o componente GUI de controle de um **MpegAudioControl**, se houver, ou exiba uma GUI personalizada com botão de rádio seletor, para deixar o usuário selecionar o leiaute de canal do áudio MPEG. A interface **MpegAudioControl** fornece o método **setChannelLayout** para configurar o leiaute do canal de áudio para vídeos MPEG, com o leiaute do canal como argumento (como referência, veja o uso de **MonitorControl** na Fig. 22.2).

22.24 Crie um jogo-da-velha com multimídia que reproduza sons de amostra de áudio quando o jogador faz um movimento válido ou faz um movimento inválido. Use um **Vector** para armazenar **AudioInputStreams** carregados previamente que representam cada clipe de áudio. Use uma linha de **Clip** para reproduzir sons em resposta às interações do usuário com o jogo. Reproduza música de fundo MIDI continuamente enquanto o jogo está em andamento. Use a interface **Player** de JMF para reproduzir um vídeo quando um jogador vence.

22.25 O pacote MIDI de Java Sound pode acessar dispositivos MIDI de *software* e de *hardware*. Se você tem um teclado MIDI que o computador pode detectar ou que possa ser ligado a uma porta MIDI IN de uma placa de som, o Java Sound pode acessar aquele teclado. Use sintetizadores, receptores, transmissores e seqüências para permitir que o usuário grave MIDI sintetizado através do teclado eletrônico. O método **getMidiDeviceInfo** de **MidiSystem** está disponível para obter as informações sobre todos os dispositivos MIDI detectáveis (*array* de objetos **MidiDevice.Info**). Use o método **getMidiDevice** de **MidiSystem** com um argumento **MidiDevice.Info** para obter o recurso do dispositivo MIDI especificado.

22.26 Melhore o programa **MidiDemo** para permitir mais configurações, como tempo, controle e repetição. **Sequencers** podem implementar **MetaEventListeners** para tratar de **MetaMessages** e **ControlEventListeners** para tratar comandos **ShortMessage.CONTROL_CHANGE** (consulte a especificação de MIDI para os tipos de mudanças e os diversos **MetaEvents**). A interface **Sequencer** também oferece muitos métodos de configuração e controle que afetam a reprodução no seqüenciador.

22.27 Melhore o programa **RTPServer** (Fig. 22.3 e Fig. 22.4) para um servidor de distribuição de vídeo em que uma entrada de vídeo ao vivo de um dispositivo de captura de vídeo (por exemplo, uma placa VFW TV, câmeras digitais) é transmitida para todos na rede. Use o endereço IP terminando com **.255** (i.e., para criar um **SessionAddress**) para transmitir para todos na sub-rede da rede. Um programa servidor deve ter acesso ao estado da transmissão do *stream*, aos controles de erros e ao gerenciamento de retardo. O pacote **javax.media.rtp** fornece muitas interfaces para tratar da configuração de *streams* e estatísticas. O pacote **javax.media.rtcp** permite acessar relatórios sobre a sessão de RTP. O pacote **javax.media.rtp.event** contém muitos eventos gerados durante uma sessão de RTP que podem ser usados para executar melhorias em RTP naqueles estágios da sessão. O pacote **javax.media.control** fornece diversas interfaces de controle úteis em sessões de RTP.

22.28 Melhore a solução do aplicativo reprodutor de mídia do Exercício 3.18 adicionando recursos de edição ao programa. Primeiro adicione uma caixa de marcação para repetição. Embora um **Processor** seja mais adequado para tarefas de controle do que um **Player**, também se pode usar as duas interfaces com um **Player** para a **DataSource** de saída do **Processor**. Os recursos de controle devem incluir formatação de trilha, ajuste de controle de *frames* (interface **FrameRateControl**), controle de *buffer* (interface **BufferControl**) e controle de qualidade (interface **QualityControl**). Inclua uma opção de programa que permita salvar clipes de mídia dados os ajustes destes controles. Para dispositivos de captura, existe uma interface **PortControl** disponível para controlar suas portas de dispositivos. Estas interfaces estão no pacote **javax.media.control**. No pacote **javax.media**, existem outras interfaces, como **Codec** e **Effect**, que permitem a geração e o processamento adicionais da mídia para formatos de mídia específicos. Permita ao usuário importar novos *codecs*. Implemente também um recurso de edição que habilite o usuário a extrair certas partes de um clipe de mídia (*Dica*: ajuste a posição da mídia de um clipe de mídia e obtenha a saída do **Processor** nas posições marcadas).

22.29 O pacote **javax.media.sound** oferece muitos recursos de áudio do sistema. Use este pacote para criar um programa de captura de som que permita salvar o *stream* de captura nos formatos, as taxas de *bits*, as frequências e codificações desejadas.

22.30 Crie um estúdio de visualização que exiba barras gráficas que se sincronizem com a reprodução de amostra de áudio.

22.31 Estenda o suporte à reprodução de MP3 para a classe **ClipPlayer** (Fig. 22.5) usando as classes do pacote **javax.media.sound.spi**. O processo de codificação de MP3 usa o *algoritmo de Huffman*.

22.32 (*Contador de histórias*) Grave em áudio um grande número de substantivos, verbos, artigos, preposições, etc. Depois utilize geração de números aleatórios para formar frases e fazer seu programa pronunciar as frases.

22.33 (*Projeto: sistema de autoria multimídia*) Desenvolva um sistema de autoria multimídia de uso geral. O programa deve permitir que o usuário componha apresentações em multimídia que consistam em texto, áudio, imagens, animações e até vídeo. O programa deixa o usuário compor uma apresentação que consiste em qualquer um desses elementos de multimídia que são selecionados de um catálogo que o programa exhibe. Forneça controles para permitir que o usuário personalize dinamicamente a apresentação enquanto ela é feita.

22.34 (*Video games*) Os vídeo games tornaram-se incrivelmente populares. Desenvolva seu próprio programa Java de vídeo game. Faça uma competição com seus colegas para desenvolver o melhor vídeo game original.

22.35 (*Demo de física: bola que salta*) Desenvolva um programa animado que mostre uma bola que salta. Atribua uma velocidade horizontal constante à bola. Permita que o usuário especifique o coeficiente de restituição, por exemplo, um coeficiente de restituição de 75% significa que, depois que a bola bate no chão, a ela retorna somente 75% da sua altura atingida antes de ser rebatida. A demonstração deve levar em conta o efeito da gravidade – isso fará com que a bola trace um caminho parabólico. Capture um som “boing” (como uma mola saltando) e reproduza o som toda vez que a bola atinge o chão.

22.36 (*Demo de física: cinética*) Se você estudou física, implemente um programa em Java que demonstrará conceitos como energia, inércia, quantidade de movimento (ou *momentum*), velocidade, aceleração, atrito, coeficiente de restituição, gravidade e outros. Crie efeitos visuais e utilize áudio para ênfase e realismo.

22.37 (*Projeto: simulador de vôo*) Desenvolva seu próprio programa simulador de vôo em Java. Este é um projeto muito desafiador. Também é um excelente candidato para uma competição com seus colegas.

22.38 (*Torres de Hanói*) Escreva uma versão animada do problema das Torres de Hanói que apresentamos no Exercício 6.37. À medida que cada disco é retirado de um pino ou encaixado sobre outro, reproduza um som de atrito de metal. Quando cada disco pousar sobre a pilha, emita um som de choque de metal. Reproduza alguma música de fundo apropriada.

22.39 (*A lebre e a tartaruga*) Desenvolva uma versão multimídia da simulação da lebre e da tartaruga que apresentamos no Exercício 7.41. Você poderia gravar voz de um apresentador descrevendo a corrida, “Os competidores estão na linha de partida.”, “Foi dada a largada!”, “A lebre dispara na frente.”, “A tartaruga está avançando com determinação.”, etc. À medida que a corrida prossegue, reproduza os áudios gravados apropriados. Reproduza sons para simular a corrida dos animais e não esqueça dos aplausos da torcida! Faça uma animação dos animais correndo para cima da montanha escorregadia.

22.40 (*Percorrendo o passeio do cavalo*) Desenvolva versões baseadas em multimídia dos programas do passeio do cavalo que você escreveu nos Exercícios 7.22 e 7.23.

22.41 (*Máquina de fliperama*) Eis outro problema de competição. Desenvolva um programa Java que simula uma máquina de fliperama que você próprio projetou. Realize uma competição com seus colegas para desenvolver a melhor máquina de fliperama multimídia original. Utilize todos os truques possíveis de multimídia que você possa imaginar para “incrementar” seu jogo de fliperama. Tente manter os mecanismos do jogo semelhantes àqueles dos jogos reais de fliperama.

22.42 (*Roleta*) Estude as regras para o jogo de roleta e implemente uma versão do jogo baseada em multimídia. Crie uma roleta giratória animada. Utilize áudio para simular o som da bola pulando os vários compartimentos que correspondem a cada um dos números. Utilize um áudio para simular o som da bola caindo em sua posição final. Enquanto a roleta está girando, permita que vários jogadores façam apostas. Quando a bola cair na posição final, você deve atualizar os saldos de cada um dos jogadores na banca com ganhos ou perdas adequadas.

22.43 (*Jogo de Craps*) Simule um jogo de *craps* completo. Utilize uma representação gráfica de uma mesa de jogo de *craps*. Permita que vários jogadores façam suas apostas. Utilize uma animação do jogador que está lançando os dados e mostre os dados animados rolando até parar. Utilize áudio para simular um pouco da conversa em volta da mesa do jogo de *craps*. Após cada lançamento, o sistema deve atualizar os saldos de cada um dos jogadores na banca de acordo com as apostas que eles fizeram.

22.44 (*Código Morse*) Modifique a solução do Exercício 10.26 para gerar como saída o código Morse com cliques de áudio. Utilize dois cliques de áudio diferentes para os caracteres de ponto e traço no código Morse.