Software Development (CS2500)

Lecture 36: **Making Music with `MIDI`**

M.R.C. van Dongen

January 21, 2011

Contents

1 Outline

Now that we can handle exceptions we're ready to make some music. We shall start with the `MIDI` basics. We use the basics to implement a simple one-sound application. Next we shall study `MidiEvents`. We shall use the `MidiEvents` to enhance our application.

The content of this lecture is not examinable. It is mainly based on the last part of Chapter 10 and is only presented so we can keep up with the case study in the book.

2 Making Sound

In Lecture 30 we studied how to create a `Sequencer` object, which was needed to create and play a sound stream. This section provides some more information about `Sequencer` object and how they're used to make sounds.

The sequencer creates and plays the music. Playing the `Sequence` is done using the `play( )` call.

- The `Sequencer` plays the `Sequence`.
- Every `Sequence` has a `Track`.
- Every `Track` is a sequence of timestamped `Messages`.
- A timestamped `Message` is called a `MidiEvent`.
- Each `Message` is a simple instruction: use a different instrument, start playing a given note, stop playing the note, ...

Creating the `Sequence` is done by creating the `Track` and adding `MidiEvents` to the `Track`.
The following is what you do in `Java`.

1. Create a `Sequencer` and `open( )` it. As it turns out this is quite complicated and the code which is provided is not correct. The following section provides code that creates the `Sequencer`. There's no point in trying to understand it as it uses some `Java` constructs which we haven't studied yet. Just use the code. After the next coming 5 lectures you should understand the `Java` constructions that are used in the code.
2. Create a new `Sequence`.

```Java
Sequence seq = new Sequence( Sequence.PPQ, 4 );
```

3. Create a new `Track` from the `Sequence`.

```Java
Track track = seq.createTrack( );
```

4. Fill the `Track` with `MidiEvents`.

```Java
track.add( midiEvent );
```

5. Give the `Sequence` to the `Sequencer` and play it.

```Java
sequencer.setSequence( seq );
sequencer.play( );
```

3 Our First Music App

This section lists a simple program that outputs a single note on Channel 1 of Track 1.

3.1 Class Overview

The following provides an overview of the main class. JavaDoc is added where necessary. However, to keep examples simple some of the JavaDoc tags have been omitted.

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

/**
 * Create single note with MIDI interface.
 */
public class MiniMusicApp {
    // MidiEvent Message types.
    private static final int ON = ShortMessage.NOTE_ON;
    private static final int OFF = ShortMessage.NOTE_OFF;
    // Meta event.
    private static final int END_OF_TRACK = 47;

    private final Sequencer sequencer;

    public static void main( String[] args ) {
        MiniMusicApp application = new MiniMusicApp( );
        application.play( );
    }
    ...
}
```

3.2 Creating the `Sequencer`

The following is the class constructor. As announced before, the construction is complicated and there's no need to understand it. The main reason for listing it is providing a complete example.

```Java
private MiniMusicApp( ) {
    Sequencer seq = null;
    try {
        seq = MidiSystem.getSequencer( );
    } catch (Exception exception) {
        handle( exception );
    }
    sequencer = seq;
    sequencer.addMetaEventListener(
    new MetaEventListener( ) {
        public void meta(MetaMessage event) {
            if (event.getType() == END_OF_TRACK) {
                sequencer.stop( );
                sequencer.close( );
                System.exit( 0 );
            }
        }
    } );
}
```

The first seven lines in the constructor demonstrate an idiom which useful when a value is assigned to a `final` variable and getting the value may throw an exception. The purpose of the seven lines is to assign a value to `sequencer`. Simply writing `sequencer = MidiSystem.getSequencer( )` in the `try`-black won't work because `Java` will notice that the assignment may fail if an exception is thrown and `Java` insists that *some* value is assigned to the variable. Temporarily assigning `null` to the variable before the `try`-block isn't possible because `sequencer` is final. To overcome the problem, we (1) use a temporary variable — in this case `seq` — which we initialise to `null`, (2) try assign `seq` the result of the call `MidiSystem.getSequencer( )`, which may throw an exception, and (3) assign `seq` to `sequencer`.

The method `handle( )` handles the exception. It is just as in the previous lecture.

```Java
private static void handle( Exception exception ) {
    String cause = exception.getMessage( );
    if (cause != null) {
        System.err.println( cause );
    }
    exception.printStackTrace( );
    System.exit( 1 );
}
```

3.3 Playing the Note

The following is the method that plays the note. The code that creates the `Sequence`, adds the `MidiEvents` to it, adds the `Sequence` to the sequencer, and plays the `Sequencer` is in the next listing.

```Java
/**
 * Plays note number 44 with given velocity
 * at timestamp 1-16 on channel 1.
 */
private void play( ) {
    final int CHANNEL  =   1;
    final int NOTE     =  44;
    final int VELOCITY = 100;
    final int START    =   1;
    final int STOP     =  16;
    try {
        ⟨creates and plays the sequence⟩
    } catch(Exception ex) {
        handle( ex );
    }
}
```

The following is the code for ⟨creates and plays the sequence⟩. The instructions follow the template at the end of the previous section. The method `addMidiEvent( )` is presented in the next section. By passing the class constant `Sequence.PPQ` to the `Sequence` constructor, we tell it that we want to create a `Sequence` wit a fixed number of *Pulses Per Quarter note*. The magic number 4 in the code is the number of pulses per quarter note.

```Java
sequencer.open( );
Sequence seq = new Sequence( Sequence.PPQ, 4 );
Track track  = seq.createTrack( );

// Add MidiEvent to track: start playing the note at time = START.
addMidiEvent( track, ON,  CHANNEL, NOTE, VELOCITY, START );
// Add MidiEvent to track: stop  playing the note at time =  STOP.
addMidiEvent( track, OFF, CHANNEL, NOTE, VELOCITY,  STOP );

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

4 MidiEvents

This section lists the code that adds a `MidiEvent` to a given `Track`. It is recalled that a `MidiEvent` consists of a `MidiMessage` at a given time. It is created as follows:

1. Create a new `MidiMessage`
2. Put the instructions in the message.

3. Create a `MidiEvent` using `MidiMessage` and time.

4. Add the `MidiEvent` to the track.

So what is a `MidiMessage`? Basically it is a very simple instruction. The the following are possible examples which have `MidiMessage` equivalents.

- Start playing a given note with a given velocity on a given channel.
- Stop playing a given note with a given velocity on a given channel.
- Change the instrument of a given channel.

Here a *note* is a number in the range 0–127. The smaller the note, the lower the note. A *channel* is a logical notion which corresponds to a musician. Different channels can be played at the same time, so this allows you to construct a whole orchestra. The *velocity* of a note determines how hard it's played. Velocity plays a role when starting and stopping a note.

The following is the code. In the code a `ShortMessage` is created which is used as a `MidiMessage`: `ShortMessage` is a subclass of `MidiMessage`.

Java

```
/**
 * Create a new MidiEvent and add it to a given Track.
 * @param track    The given track.
 * @param type     The type of the MidiEvent's message.
 * @param channel  The channel of the MidiEvent's message.
 * @param note     The note of the MidiEvent's message.
 * @param velocity The velocity of the MidiEvent's message.
 * @param tick     The tick of the MidiEvent.
 */
private static
void addMidiEvent( Track track, int type, int channel,
                   int note, int velocity, int tick )
                 throws InvalidMidiDataException {
    // Create the MidiMessage.
    ShortMessage message = new ShortMessage( );
    // Set message details.
    message.setMessage( type, channel, note, velocity );
    // Create the event and add it to the track.
    MidiEvent event = new MidiEvent( message, tick );
    track.add( event );
}
```

5 Additional Features

In this section we shall add some additional features to our basic application. Specifically, we shall add a feature that allows us to play a different note. Next we shall print out the `Tracks` and `MidiEvents` of a

given Sequencer.

5.1 Playing a Different Tune

As you may remember from the previous section, a MidiMessage may be used to change the note of a given channel. The following demonstrates how this is done.

```Java
private static final int NEW_INSTRUMENT;
NEW_INSTRUMENT = ShortMessage.PROGRAM_CHANGE;
...
try {
    ShortMessage message = new ShortMessage( );
    message.setMessage( NEW_INSTRUMENT, channel, instrument, 0 );
    MidiEvent changeInstrument = new MidiEvent( message, tick );
} catch (InvalidMidiDataException exception) {
    handle( exception );
}
```

Here instrument can be any number in the range 0–127.

5.2 Showing the `MidiEvents`

Given a Sequence we can also explore what's in the sequence.

`getTicksLength( ):` Returns the number of ticks in the Sequence.

`getMicrosecondLength( ):` Returns the number of microseconds in the Sequence.

`getDivisionType( ):` Returns the timing division type for this Sequence. (We shall ignore this, but this is the first argument we passed to the constructor of the Sequence.)

`getdTracks( ):` Returns an array with the Tracks on the Sequence. In our application we only had one track.

Each Track on the Sequence's Tracks corresponds to a sequence of MidiEvents. Let track be a Track. then `track.size( )` returns the number of MidiEvents on the Track. furthermore, `track.get( i )` returns its ith MidiEvent

Once we have the MidiEvent there are several methods which can be used to get its details.

```Java
MidiEvent event = track.get( trackNumber );
long tick = event.getTick( );
MidiMessage message = event.getMessage( );
// Turn the message into some interesting information.
String info = convertMessageToString( message );
System.out.println( " tick " + tick + ", " + info );
```

The following is a simple method to turn a `MidiMessage` into a descriptive `String`. By no means is it intended to be used for a general purpose application: we assume the only three events are 'start a note', 'stop a note', and 'change the instrument'.

```Java
private static
String convertMessageToString( MidiMessage message ) {
    byte[] bytes = message.getMessage( );
    String result;
    result = "END OF TRACK"; // Not true in general, but it is here.
    if (message instanceof ShortMessage) {
        ShortMessage msg= (ShortMessage)message;
        int channel = msg.getChannel( );
        int command = msg.getCommand( );
        int first   = msg.getData1( );
        result = "channel " + channel + ". ";
        result = result + convertCommandNumberToString( command );
    }
    return result;
}
```

```Java
private static
String convertCommandNumberToString( int command ) {
    String result = "";
    if (command == ShortMessage.PROGRAM_CHANGE) {
        result = result + "change instrument to " + first;
    } else if (command == ShortMessage.NOTE_ON) {
        result = result + "note " + first + " on"
                        + ", velocity = " + msg.getData2( );
    } else if (command == ShortMessage.NOTE_OFF) {
        result = result + "note " + first + " off"
                        + ", velocity = " + msg.getData2( );
    }
    return result;
}
```

6 For Friday

study Chapter 10. ry implement a program that plays several different instruments at different times. When doing this, don't use multiple tracks. Keep things simple: much can go wrong.

7 Acknowledgements

Thanks to Dave Murphy (d.murphy@cs.ucc.ie) for a useful discussion. Some of the material of this lecture is based on Dan Becker's Java MIDI programming tutorial, which may be found at http://www.ibm.com/developerworks/library/it/it-0801art38/.