


## B4. Toneinbindung und Tonbearbeitung in Java

- B4.1 Java Sound API im Überblick 
- B4.2 Pipelines zur Audio-Verarbeitung
- B4.3 Bearbeitung von Audio-Samples
- B4.4 Umgang mit MIDI-Daten

### Literatur:

<http://java.sun.com/products/java-media/sound/>  
<http://www.jsresources.org>  
<http://www.tritonius.org>

## Java Sound API

- "Low-level" API
  - zur Steuerung der Ein- und Ausgabe von Tonmedien
  - umfasst Funktionen für digitale Audioinformation und für MIDI-Daten
  - erweiterbare Basis, keine ausgefeilten Editor-Funktionen o.ä.
- Verwandte Java-Technologien:
  - Java Media Framework (JMF)
    - » auf höherer Ebene angesiedelt
    - » einfachere Lösung für Abspielen von Tonmedien
    - » Synchronisation mit anderen Medien (v.a. Video)
- Pakete des Java Sound APIs (in Standard-Java-Installation enthalten):
  - `javax.sound.sampled`
  - `javax.sound.midi`

## Entwicklung der Sound-Unterstützung in Java

- Java 1.0 und 1.1:
  - sehr einfache Unterstützung durch die Schnittstelle `java.applet.audioclip`
- Java 2-Plattform (ab 1.2)
  - Java Sound seit Version 1.3 (noch nicht in 1.2)
  - Nur Fehlerkorrektur in Java 1.4 und 1.5

## Klassifikation von digitaler Audioverarbeitung

- Entstehungsprozess des Materials
  - Weiterverarbeitung von ursprünglich analogem Tonmaterial (*sampled*)
  - Digitale Erzeugung von Musik (*MIDI*)
- Zeitlicher Verlauf
  - Realzeit-Behandlung:
    - » Weitersenden empfangener Information in kurzer Zeit, ungefähr mit der gleichen Rate wie Daten empfangen werden (*online*)
    - » Operationen werden beendet, bevor alle Daten bekannt sind
  - Speicher-basierte Behandlung:
    - » Laden der gesamten Information (evtl. stufenweise) in den Computerspeicher
    - » Operationen werden erst begonnen, wenn alle Daten bekannt sind (*offline*)

## Digitales Audio in Java Sound

- Hauptfokus in `javax.sound.sampled`:
  - Transport von Audiodaten (Aufnahme und Wiedergabe)
  - Verwaltung von Puffern
  - Mischen von Daten
  - Steuerung: Start, Anhalten, Aussetzen
- Prinzipiell mit der Schnittstelle möglich:
  - Direkte Bearbeitung von Sample-Information (z.B. selbstgeschriebene Synthesizer- und Filter-Funktionen)
- Default-Implementierung eingeschränkt in Funktionalität:
  - z.B. Einlesen von WAV-Dateien, aber nicht von MP3- oder MPEG4-AAC-Dateien
  - Funktionalität erweiterbar durch „Service Provider Interface“
    - » Plug-Ins mit zusätzlicher Codec-Funktionalität (teilweise) verfügbar

## Audio-Formate

- Java Sound befasst sich mit formatierten Audio-Daten
- Ein Audio-Format wird durch ein Objekt der Klasse `AudioFormat` beschrieben und enthält die folgenden Angaben:
  - Codierungsverfahren (normalerweise Pulscode-Modulation (PCM))
    - » Zusammenhang zwischen Klangdruck und Signal: (PCM/linear,  $\mu$ -Gesetz, a-Gesetz)
    - » Bei PCM: Samplewerte mit/ohne Vorzeichen (*signed/unsigned*)
  - Anzahl von Kanälen (z.B. 2 für Stereo)
  - Abtastrate (samples/second)
  - Auflösung (bits/sample)
  - Paketgröße (*frame size*) (bytes)
  - Paketrate (*frame rate*) (frames/s)
  - Byte-Lesereihenfolge (*byte order*) (big-endian/little-endian) (Wo steht das höchstwertige Bit?)

## AudioInputStream

- `javax.sound.sampled.AudioInputStream`
- Abgeleitet von der abstrakten Oberklasse `java.io.InputStream`
  - Byteweise lesbare Datenströme
  - `read()`, `skip()`, `close()`
  - `markSupported()`, `mark()`, `reset()`:  
Springen zu markierter Position
- Methoden zum Öffnen einer Audiodatei:  
`javax.sound.sampled.AudioSystem`
  - `AudioInputStream getAudioInputStream(File file)`
  - Es werden ggf. mehrere Parser für unterstützte Audio-Formate eingesetzt
  - Konversion von Streams in andere Formate möglich
- Ein `AudioInputStream` hat immer ein festgelegtes `AudioFormat`
  - Das Format wird z.B. beim Öffnen einer Audio-Datei festgelegt und im Streamobjekt gespeichert.
  - `AudioFormat getFormat()`


## Beispiel: Öffnen einer Audio-Datei

```
import javax.sound.sampled.*;
...

public static void main(String[] args) {

    String strFilename = args[0];
    File soundFile = new File(strFilename);
    AudioInputStream audioInputStream = null;
    try {
        audioInputStream =
            AudioSystem.getAudioInputStream(soundFile);
    }
    catch (Exception e) {};
    AudioFormat audioFormat =
        audioInputStream.getFormat();
    System.out.println(audioFormat);
    ...
}
```

## B4. Toneinbindung und Tonbearbeitung in Java

- B4.1 Java Sound API im Überblick
- B4.2 Pipelines zur Audio-Verarbeitung 
- B4.3 Bearbeitung von Audio-Samples
- B4.4 Umgang mit MIDI-Daten

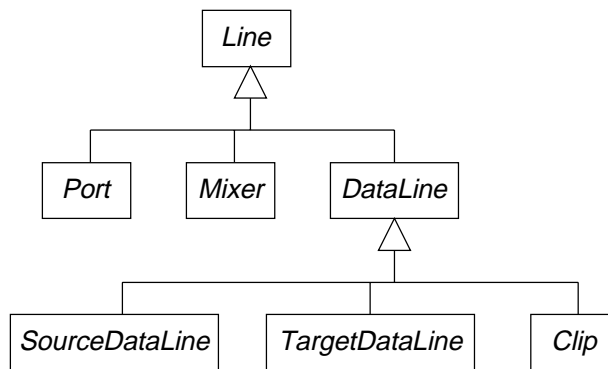
### Literatur:

<http://java.sun.com/products/java-media/sound/>  
<http://www.jsresources.org>  
<http://www.tritonius.org>

## Port und Mixer

- Ein *Port* beschreibt einen Ein- oder Ausgang für einen Strom von Audio-Daten zu einem (virtuellen oder physikalischen) Gerät
- Eine *DataLine* beschreibt einen Datenstrom, der steuerbar ist (Start/Stop).
- Ein *Mixer* ist die Java-Sound-Abstraktion für alle tonverarbeitenden Komponenten mit Ein- und Ausgabeports
  - Kann einem realen physikalischen Gerät entsprechen oder rein in Software realisiert werden
  - Wichtiger Spezialfall eines Mixer-Objekts: Mischpult-Funktion
  - Anwendungsprogramme können Daten entweder auf Eingabe-Ports abliefern oder auf Ausgabeports abholen (streaming)
  - Alternativ können Daten für Eingabeports aus Dateien gelesen werden und für Ausgabeports in Dateien geschrieben werden (audio files)
- Das Konzept der *Line* verallgemeinert Ein- und Ausgabeports und Mixer zu beliebigen Bestandteilen einer "Audio-Pipeline"

## Schnittstellen-Hierarchie für Audio-Pipelines



Gepufferte Wiedergabe  
einer Audioquelle

Gepufferte Aufnahme  
einer Audioquelle

Wiedergabe einer  
Audioquelle, die  
komplett in den Speicher  
vorgeladen wird

## Audio-Ressourcenverwaltung

- Typischerweise ist Audioverarbeitung an Systemressourcen gebunden
  - Installierte Geräte
  - Pufferspeicher in Systembereichen
- Erzeugung eines Line-Objekts ausschließlich über zentrale Verwaltung:
  - `javax.sound.sampled.AudioSystem`
- Anfragefunktionen, welche Ressourcen vorhanden sind
  - Es gibt immer einen Standard-Mixer, der auf das eingebaute Hardware-Audio-System abgebildet wird
- Erzeugung eines speziellen Line-Objekts
- Zur Beschreibung von Line-Objekten:
  - `javax.sound.sampled.Line.Info` mit Unterklassen:
    - » `DataLine.Info`, `Port.Info`
  - `javax.sound.sampled.Mixer.Info`

## Beispiel: Abfrage der Audio-Systemressourcen

```
import javax.sound.sampled.*;

public class LineInfoQuery {

    public static void main (String[] args) {
        System.out.println("Mixers available:");
        Mixer.Info[] mixers = AudioSystem.getMixerInfo();
        for (int i = 0; i < mixers.length; i++) {
            System.out.println(mixers[i]);
        };
        System.out.println("System default mixer:");
        Mixer stdMixer = AudioSystem.getMixer(null);
        System.out.println(stdMixer.getMixerInfo());
        System.out.println("Supported source line types
            in default mixer:");
        Line.Info[] lineTypes = stdMixer.getSourceLineInfo();
        for (int i = 0; i < lineTypes.length; i++) {
            System.out.println(lineTypes[i]);
        };
        ...
    }
}
```

## Erzeugung eines Line-Objekts

- Anfrage an AudioSystem, unter Angabe von LineInfo-Information für das gewünschte Line-Objekt
  - welcher Typ:
    - » *SourceDataLine* (Wiedergabe)
    - » *Clip* (wiederholte Wiedergabe)
    - » *TargetDataLine* (Aufnahme)
  - welches Audioformat
- Beispiel (Ausgabe von Audioinformation):

```
AudioFormat audioFormat =
    audioInputStream.getFormat();
DataLine.Info info = new DataLine.Info
    (SourceDataLine.getClass(), audioFormat)
AudioSystem.getLine(info);
```

## Operationen zur Nutzung eines Line-Objekts

- Bisher nur die Verfügbarkeit des gewünschten Typs von `Line` geprüft.
- Nächste Schritte:
  - Reservierung der Ressource mit `open()`
    - » Reservieren des benötigten Pufferbereichs
    - » kann bei gleichzeitigen Anforderungen anderer Programme an Ressourcenmangel scheitern
    - » bei `sourceDataLine` Angabe eines Audioformats möglich
  - Bei `DataLine` (d.h. steuerbarem Strom):
    - » verschiedene Zustände:  
running = true/false (Übertragung ein/aus)
    - » Bei Erzeugung: running = false
    - » Starten durch `start()`

Die meisten folgenden Beispiele basieren auf [www.jsresources.org](http://www.jsresources.org) !

## Beispiel: Öffnen einer Audio-Line zur Wiedergabe

```
public static void main(String[] args) {
    String strFilename = args[0];
    File soundFile = new File(strFilename);
    AudioInputStream audioInputStream = null;
    try {
        audioInputStream =
            AudioSystem.getAudioInputStream(soundFile);
    }
    catch (Exception e) {};

    AudioFormat audioFormat = audioInputStream.getFormat();
    SourceDataLine line = null;
    DataLine.Info info =
        new DataLine.Info(SourceDataLine.class, audioFormat);
    try {
        line = (SourceDataLine) AudioSystem.getLine(info);
        line.open();
    }
    catch (Exception e) {};
    line.start();
    ...
}
```



## Beispiel: Audiowiedergabe aus Datei

```
...
byte[] abData = new byte[EXTERNAL_BUFFER_SIZE]; //128k

int nBytesRead = 0;
while (nBytesRead != -1) {
    try {
        nBytesRead =
            audioInputStream.read(abData, 0, abData.length);
    }
    catch (Exception e) {};
    if (nBytesRead >= 0)
        line.write(abData, 0, nBytesRead);
}
...
SourceDataLine.write()
```

The diagram illustrates the parameters of the `read` method call: `abData` is the buffer (Puffer), `0` is the start index (Startindex), and `abData.length` is the end index (Endindex). The `write` method call in the `if` statement is also annotated with a box labeled `SourceDataLine.write()`.

## Beispiel: Aufnahmen von Audio-Daten (1)

```
public static void main(String[] args) {

    String strFilename = args[0];
    File outputFile = new File(strFilename);
    AudioFormat audioFormat = new AudioFormat(
        AudioFormat.Encoding.PCM_SIGNED, 44100.0F,
        16, 2, 4, 44100.0F, false);
    DataLine.Info info = new DataLine.Info(
        TargetDataLine.class, audioFormat);
    TargetDataLine targetDataLine = null;
    try {
        targetDataLine =
            (TargetDataLine) AudioSystem.getLine(info);
        targetDataLine.open(audioFormat);
    }
    catch (Exception e) {}
    AudioFileFormat.Type targetType =
        AudioFileFormat.Type.WAVE;
    ...
}
```

## Beispiel: Aufnahmen von Audio-Daten (2)

```
...
SimpleAudioRecorder1 recorder = new SimpleAudioRecorder1(
    targetDataLine, targetType, outputFile);
System.out.println(
    "Press ENTER to start the recording.");
try {
    System.in.read();
}
catch (Exception e) {}
recorder.start();
System.out.println("Recording...");
System.out.println("Press ENTER to stop the recording.");
try {
    System.in.read();
}
catch (Exception e) {}
recorder.stopRecording();
System.out.println("Recording stopped.");
System.exit(0);
}
```

## Nebenläufigkeit in Java: Threads

- Häufig laufen in der Medientechnik mehrere Verarbeitungsketten nebeneinander ab
  - Beispiel: Aufnahme eines Audiosignals || Warten auf Ende-Befehl
- Java-Threads:
  - "leichtgewichtige Prozesse" – innerhalb eines Betriebssystemprozesses
  - Jeder Thread implementiert die Schnittstelle `java.lang.Runnable`
  - Einzige Methode in `Runnable`:

```
public void run()
```
  - Standard-Implementierung von `Runnable`:

```
java.lang.Thread implements Runnable
```
  - Starten eines Threads - d.h. der `run()`-Methode:

```
t.start()
```

## Beispiel: Ein nebenläufiger Java-Audio-Recorder

```
public class SimpleAudioRecorder1 extends Thread {
    private TargetDataLine      m_line;
    private AudioFileFormat.Type m_targetType;
    private AudioInputStream    m_audioInputStream;
    private File                m_outputFile;
    public SimpleAudioRecorder1(TargetDataLine line,
        AudioFileFormat.Type targetType, File file) {
        m_line = line;
        m_audioInputStream = new AudioInputStream(line);
        m_targetType = targetType;
        m_outputFile = file;
    }
    ...
    public void run() {
        try {
            AudioSystem.write(
                m_audioInputStream, m_targetType, m_outputFile);
        }
        catch (Exception e) {}
    } ...
}
```


## Starten und Anhalten der Aufnahme

```
public class SimpleAudioRecorder1 extends Thread {
    private TargetDataLine      m_line;
    private AudioFileFormat.Type m_targetType;
    private AudioInputStream    m_audioInputStream;
    private File                m_outputFile;
    ...
    public void start() {
        m_line.start(); // Startet die Aufnahme im Audio-System
                       // (TargetDataLine)
        super.start(); // Startet den Datentransfer-Thread
                       // d.h. self.run()
    }
    public void stopRecording() {
        m_line.stop();
        m_line.close();
    }
}
```

## Grundkonzepte der Audio-Verarbeitung

- Hardware und systemnahe Software einbinden
  - Aufrufe an Verwaltungsobjekte (wie AudioSystem)
- Generische Architektur
  - Durch Programmcode “steckbare” Bausteine
  - “Pipes and Filters”-Muster = Pipeline-Architektur
- Nebenläufigkeit und Echtzeitbehandlung sind unerlässlich
- Datenstruktur Puffer
  - Letztlich ist der Sound ein Byte-Array...

## B4. Toneinbindung und Tonbearbeitung in Java

- B4.1 Java Sound API im Überblick
- B4.2 Pipelines zur Audio-Verarbeitung
- B4.3 Bearbeitung von Audio-Samples 
- B4.4 Umgang mit MIDI-Daten


### Literatur:

<http://java.sun.com/products/java-media/sound/>  
<http://www.jsresources.org>

## Beispiel: Direkte Bearbeitung von Samples

```
public class SineOscillator extends AudioInputStream {
    public SineOscillator
        (float fSignalFrequency, float fAmplitude,
         AudioFormat audioFormat, long lLength) {
        super(new ByteArrayInputStream(new byte[0]),
              new AudioFormat(AudioFormat.Encoding.PCM_SIGNED,...), lLength);
        ...
        m_abData = new byte[nBufferLength];
        for (int nFrame = 0; nFrame < nPeriodLengthInFrames; nFrame++) {
            float fPeriodPosition =
                (float) nFrame / (float) nPeriodLengthInFrames;
            float fValue =
                (float) Math.sin(fPeriodPosition * 2.0 * Math.PI);
            int nValue = Math.round(fValue * fAmplitude);
            int nBaseAddr = (nFrame) * getFormat().getFrameSize();
            m_abData[nBaseAddr + 0] = (byte) (nValue & 0xFF);
            m_abData[nBaseAddr + 1] = (byte) ((nValue >>> 8) & 0xFF);
            m_abData[nBaseAddr + 2] = (byte) (nValue & 0xFF);
            m_abData[nBaseAddr + 3] = (byte) ((nValue >>> 8) & 0xFF);
        }
        m_nBufferPosition = 0;
    }
}
```

## B4. Toneinbindung und Tonbearbeitung in Java

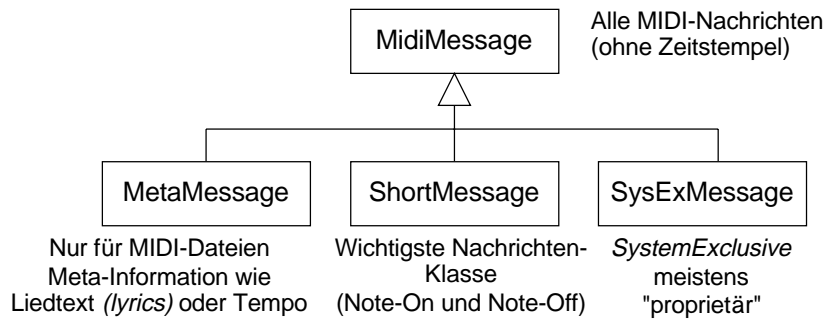
- B4.1 Java Sound API im Überblick
- B4.2 Pipelines zur Audio-Verarbeitung
- B4.3 Bearbeitung von Audio-Samples
- B4.4 Umgang mit MIDI-Daten 

### Literatur:

<http://java.sun.com/products/java-media/sound/>  
<http://www.jsresources.org>

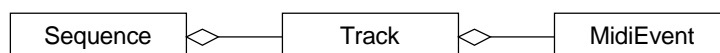
## MIDI-Audio in Java Sound

- Paket `javax.sound.midi`



## MIDI-Ereignisse in Java Sound

- `javax.sound.midi.MidiEvent`
  - enthält `MidiMessage`
  - und Zeitstempel
- `javax.sound.midi.Track`
  - Strom von `MidiEvents`
  - Kann 16 Kanäle adressieren
- `javax.sound.midi.Sequence`
  - Repräsentiert musikalische Einheit (z.B. Musikstück)
  - enthält Information über Zeitbasis
  - enthält einen oder beliebig viele MIDI-Tracks



## MIDI-Ressourcenverwaltung

- Analog zu Audio-Verarbeitungsressourcen
- `javax.sound.midi.MidiSystem`
  - Zentrale Verwaltung aller MIDI-bezogenen Systemressourcen
- Typen von MIDI-Systemressourcen:
  - **Sequencer**: Gerät/Objekt zur Aufnahme und Wiedergabe von MIDI-Ereignis-Sequenzen; benutzt Transmitter und Receiver
  - **Transmitter**: Schnittstellenobjekt zum Senden von MIDI-Ereignis-Sequenzen
  - **Receiver**: Schnittstellenobjekt zum Empfangen von MIDI-Ereignis-Sequenzen
  - **Synthesizer**: Objekt zur Klangerzeugung
- Die eigentlichen MIDI-Nachrichten sind in der Klasse `MidiChannel` codiert.

## Beispiel: MIDI-Player in Java (1)

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

public class SimpleMidiPlayer1 {

    private static Sequencer sm_sequencer = null;
    private static Synthesizer sm_synthesizer = null;

    public static void main(String[] args) {

        String strFilename = args[0];
        File midiFile = new File(strFilename);
        Sequence sequence = null;
        try {
            sequence = MidiSystem.getSequence(midiFile);
            sm_sequencer = MidiSystem.getSequencer();
        }
        catch (Exception e) {};
        ...
    }
}
```

## Beispiel: MIDI-Player in Java (2)

```
...
try {
    sm_sequencer.open();
    sm_sequencer.setSequence(sequence);
}
catch (Exception e) {}
if (! (sm_sequencer instanceof Synthesizer)) {
    try {
        sm_synthesizer = MidiSystem.getSynthesizer();
        sm_synthesizer.open();
        Receiver synthReceiver =
            sm_synthesizer.getReceiver();
        Transmitter seqTransmitter =
            sm_sequencer.getTransmitter();
        seqTransmitter.setReceiver(synthReceiver);
    }
    catch (Exception e) {}
}
sm_sequencer.start();
}
```

## MIDI-Synthesizer: Abspielen einer Note (1)

```
import javax.sound.midi.*;
public class SynthNote1 {
    public static void main(String[] args) {
        int    nNoteNumber = 0;        // MIDI key number
        int    nVelocity = 0;
        int    nDuration = 0;
        nNoteNumber = Integer.parseInt(args[0]);
        nNoteNumber = Math.min(127, Math.max(0, nNoteNumber));
        nVelocity = Integer.parseInt(args[1]);
        nVelocity = Math.min(127, Math.max(0, nVelocity));
        nDuration = Integer.parseInt(args[2]);
        nDuration = Math.max(0, nDuration);
        Synthesizer synth = null;
        try {
            synth = MidiSystem.getSynthesizer();
            synth.open();
        }
        catch (Exception e) {};
        ....
    }
}
```



## MIDI-Synthesizer: Abspielen einer Note (2)

```
...
MidiChannel[] channels = synth.getChannels();
channels[0].noteOn(nNoteNumber, nVelocity);
try {
    Thread.sleep(nDuration);
}
catch (InterruptedException e) {}
channels[0].noteOff(nNoteNumber);
System.exit(0);
}
}
```