# Medientechnik

Übung

# Heute

- Java Sound:
  - Grundlagen
  - Soundfiles abspielen
  - MIDI

# 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

# Digitale 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)*
  - 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)

# 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, µ-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)
  - etc.

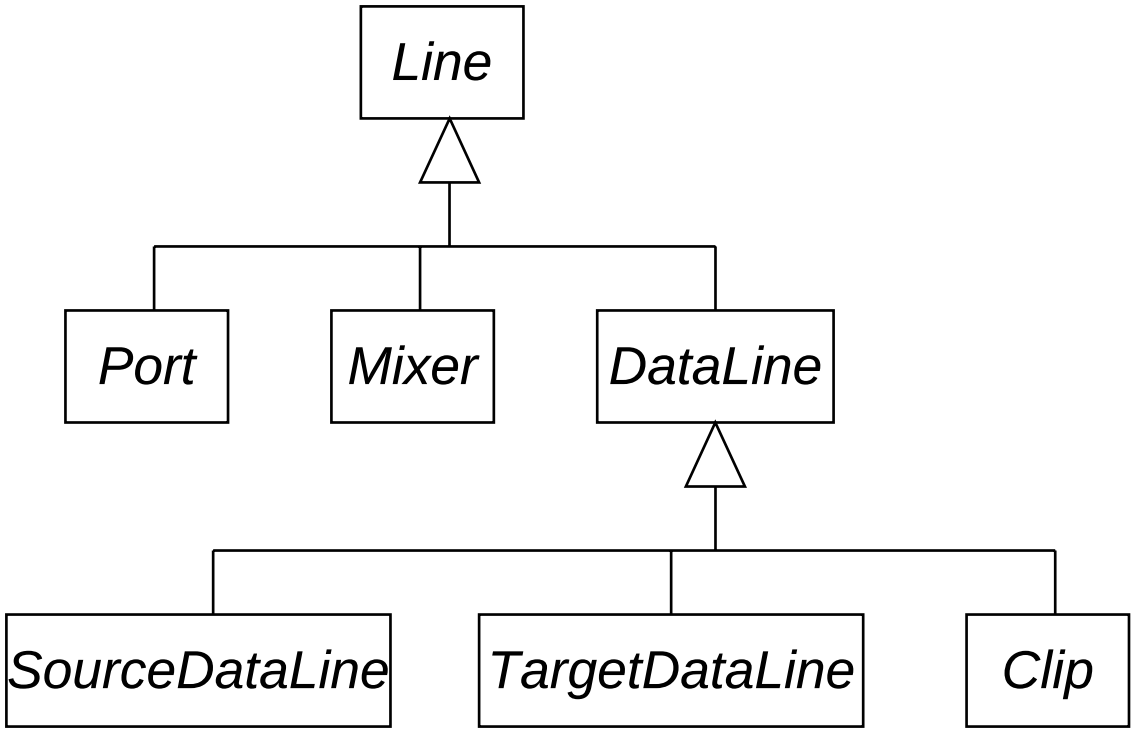
# AudioInputStream

- **`javax.sound.sampled.AudioInputStream`**
  - 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)`** oder **`getAudioInputStream(URL url)`**
  - Es werden ggf. mehrere Parser für unterstützte Audio-Formate eingesetzt
- Ein AudioInputStream hat immer ein festgelegtes **`AudioFormat`**
  - Das Format wird z.B. beim Öffnen einer Audio-Datei festgelegt und im Streamobjekt gespeichert.
  - **`AudioFormat getFormat()`**

# Line, 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
  - 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`**

# Beispielcode

/home/proj/mi_mt_ss08/jsound/src

Bitte alle java-Dateien in euer home-Verzeichnis kopieren!

Beispiele aus
www.jsresources.org

# Codebeispiel: Abspielen einer Audiodatei

*SimpleWavePlayer.java* (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);
```

# 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()`**

# Codebeispiel: Abspielen einer Audiodatei

*SimpleWavePlayer.java* (II)

# 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);
  }
...
```

Puffer

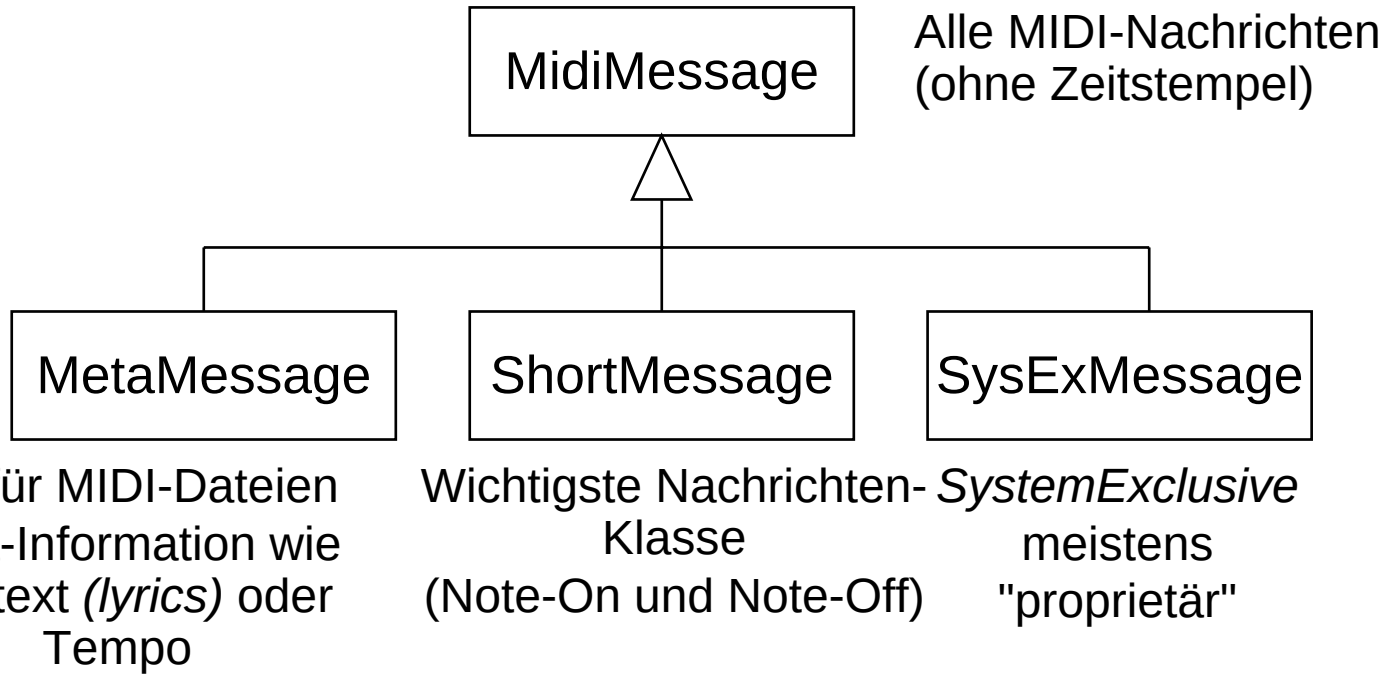
Startindex

Endindex

SourceDataLine.write()

# 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 **`MidiEvent`**s
  - Kann 16 Kanäle addressieren
- **`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 **`AudioSystem`**
- **`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.

# Codebeispiel: Abspielen einer MIDI-Datei

*SimpleMidiPlayer.java*

# Codebeispiel: Abspielen einer MIDI-Note

*SynthNote.java*