Реферат на тему

**Бібліотека Java SE: Java Sound**

Обробка цифрових аудіоданих

Підготував Копиця Олексій Сергійович

механіко-математичний факультет,

3 курс, комп’ютерна математика 2

Зміст

[**1. Введення до бібліотеки Java Sound** 4](#_Toc120399993)

[**Для чого створена бібліотека Java Sound?** 4](#_Toc120399994)

[**Сфери застосування бібліотеки Java Sound?** 4](#_Toc120399995)

[**Як Java Sound пов’язана з іншими інтерфейсами?** 4](#_Toc120399996)

[**Пакети** **Java Sound** 4](#_Toc120399997)

[**2. Пакет Sampled Audio** 5](#_Toc120399998)

[**Що таке Sampled Audio?** 5](#_Toc120399999)

[**Конфігурації аудіо** 5](#_Toc120400000)

[**3. Пакет MIDI** 6](#_Toc120400001)

[**Що таке MIDI?** 6](#_Toc120400002)

[**Конфігурації MIDI** 6](#_Toc120400003)

[**4. Доступ до ресурсів аудіосистеми** 7](#_Toc120400004)

[**Клас AudioSystem** 7](#_Toc120400005)

[**Інформаційні об’єкти** 7](#_Toc120400006)

[**Отримання мікшера** 7](#_Toc120400007)

[**Отримання лінії потрібного типу** 7](#_Toc120400008)

[**Отримання лінії безпосередньо з AudioSystem** 8](#_Toc120400009)

[**Отримання лінії з мікшера** 9](#_Toc120400010)

[**5. Відтворення аудіо** 9](#_Toc120400011)

[**Використання Clip** 9](#_Toc120400012)

[**Налаштування Clip для відтворення** 9](#_Toc120400013)

[**Початок і зупинка відтворення** 10](#_Toc120400014)

[**Використання SourceDataLine** 10](#_Toc120400015)

[**Налаштування SourceDataLine для відтворення** 10](#_Toc120400016)

[**Початок і зупинка відтворення** 10](#_Toc120400017)

[**6. Запис аудіо** 12](#_Toc120400018)

[**Налаштування TargetDataLine** 12](#_Toc120400019)

[**Зчитування даних з TargetDataLine** 13](#_Toc120400020)

[**7. Обробка звуку за допомогою елементів керування** 14](#_Toc120400022)

[**Введення до засобів керування** 14](#_Toc120400023)

[**Отримання лінії, яка має бажані елементи керування** 16](#_Toc120400024)

[**Отримання елементів керування з лінії** 16](#_Toc120400026)

[**Використання елемента керування для зміни аудіосигналу** 16](#_Toc120400027)

[**Керування беззвучним станом (mute) лінії** 17](#_Toc120400028)

[**Зміна гучності лінії** 17](#_Toc120400029)

[**Вибір між різними налаштуваннями реверберації** 17](#_Toc120400030)

[**8. Використання конвертерів файлів і форматів** 18](#_Toc120400031)

[**Зчитування звукових файлів** 18](#_Toc120400032)

[**Запис звукових файлів** 20](#_Toc120400033)

[**Конвертування з одного формату файлу в інший** 21](#_Toc120400034)

[**Конвертування звуку між різними форматами даних** 22](#_Toc120400042)

**1. Введення до бібліотеки Java Sound**

**Для чого створена бібліотека Java Sound?**

Java Sound API — це низькорівневий API для керування введенням і виведенням звукових носіїв, включаючи аудіо та дані цифрового інтерфейсу музичних інструментів (MIDI). Java Sound забезпечує явний контроль над можливостями, які зазвичай необхідні для введення та виведення звуку, у рамках, що сприяє розширюваності та гнучкості.

**Сфери застосування бібліотеки Java Sound?**

Потенційні сфери застосування включають:

* Комунікаційні структури, такі як конференції
* Системи доставки контенту до кінцевих користувачів, наприклад, медіаплеєри та музика під час трансляцій контенту
* Інтерактивні прикладні програми, такі як ігри та веб-сайти, які використовують динамічний вміст
* Створення та редагування контенту
* Інструменти та утиліти

**Як Java Sound пов’язана з іншими інтерфейсами?**

Бібліотека Java Sound забезпечує найнижчий рівень підтримки звуку на платформі Java. Вона надає прикладним програмам великий контроль над звуковими операціями та є розширюваною. Наприклад, Java Sound надає механізми для встановлення, доступу та керування системними ресурсами, такими як аудіомікшери, MIDI-синтезатори, інші аудіо- або MIDI-пристрої, засоби читання та запису файлів і конвертери звукових форматів. Java Sound не включає складні звукові редактори чи графічні інструменти, але вона надає можливості, на основі яких можна створювати такі програми. Вона підкреслює низький рівень контролю, який зазвичай очікується від кінцевого користувача.

**Пакети** **Java Sound**

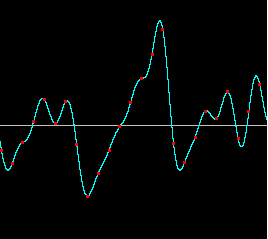
Java Sound включає підтримку як цифрового аудіо, так і даних MIDI. Ці два основні функціональні модулі надаються в окремих пакетах:

* javax.sound.sampled  Цей пакет визначає інтерфейси для захоплення, мікшування та відтворення цифрового аудіо (sampled audio).
* javax.sound.midi Цей пакет надає інтерфейси для синтезу MIDI

**2. Пакет Sampled Audio**

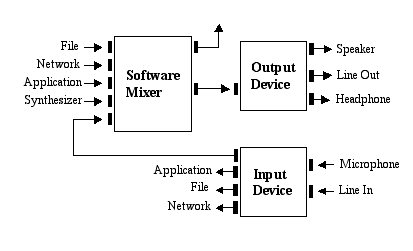
**Що таке Sampled Audio?**

Пакет javax.sound.sampled  обробляє цифрові аудіодані, які бібліотека Java Sound називає семплованим аудіо (sampled audio). Семпли (samples) — це послідовні знімки сигналу. У випадку аудіо сигналом є звукова хвиля. Мікрофон перетворює акустичний сигнал у відповідний аналоговий електричний сигнал, а аналого-цифровий перетворювач перетворює це й аналоговий сигнал у дискретизовану цифрову форму. На наступному малюнку зображено короткий момент звукового запису



**Конфігурації аудіо**

Java Sound API не передбачає певної конфігурації звукового обладнання; він розроблений, щоб дозволити установлювати різні типи аудіокомпонентів у системі та отримувати доступ через API. Java Sound підтримує такі загальні функції, як вхід і вихід із звукової карти (наприклад, для запису та відтворення звукових файлів), а також змішування кількох потоків аудіо. Ось один приклад типової аудіоархітектури:



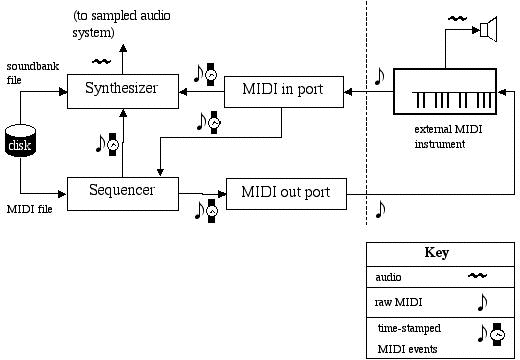
**3. Пакет MIDI**

**Що таке MIDI?**

У той час як семплований звук є прямим представленням самого звуку, дані MIDI можна розглядати як рецепт для створення звуку, особливо музичного звуку. Дані MIDI, на відміну від аудіоданих, не описують звук безпосередньо. Замість цього вони описують події, які впливають на звук, який створює синтезатор. Дані MIDI аналогічні подіям клавіатури та миші графічного інтерфейсу користувача. У випадку MIDI події можна розглядати як дії на музичній клавіатурі разом із діями на різних педалях, повзунках, перемикачах і регуляторах цього музичного інструменту. Ці події не обов’язково повинні відбуватися з апаратного музичного інструменту; їх можна моделювати програмним забезпеченням і зберігати у файлах MIDI. Програма, яка може створювати, редагувати та виконувати ці файли, називається секвенсором. Багато комп’ютерних звукових карт включають мікросхеми музичного синтезатора, керовані MIDI, на які секвенсори можуть надсилати свої MIDI-події. Синтезатори також можуть бути реалізовані повністю програмно. Синтезатори інтерпретують отримані MIDI-події та створюють аудіовихід. Зазвичай звук, синтезований із даних MIDI, є музичним (на відміну від мови, наприклад). MIDI-синтезатори також здатні генерувати різноманітні звукові ефекти.

**Конфігурації MIDI**

Наведена нижче діаграма ілюструє функціональні зв’язки між основними компонентами в можливій конфігурації MIDI на основі Java Sound API. (Як і у випадку з аудіо, Java Sound дозволяє встановлювати та з’єднувати різноманітні програмні пристрої MIDI. Показана тут система є одним із потенційних сценаріїв.) Потік даних між компонентами позначається стрілками.



**4. Доступ до ресурсів аудіосистеми**

Java Sound API використовує гнучкий підхід до налаштування системи. На комп'ютер можна встановити різні аудіопристрої (мікшери). API робить небагато припущень щодо того, які пристрої встановлено та які їхні можливості. Натомість він надає способи для системи повідомляти про доступні аудіокомпоненти та способи для нашої програми доступу до них.

**Клас AudioSystem**

Клас AudioSystem діє як інформаційний центр для аудіокомпонентів, включаючи вбудовані служби та окремо встановлені служби від сторонніх постачальників. AudioSystem служить точкою входу прикладної програми для доступу до цих встановлених ресурсів семплованих аудіо. Ми можемо запитати AudioSystem, щоб дізнатися, які види ресурсів було встановлено, а потім отримати до них доступ.

**Інформаційні об’єкти**

Кілька класів в Java Sound API надають корисну інформацію про пов’язані інтерфейси. Наприклад, Mixer.Info надає деталі про встановлений мікшер. Line.Info отримує клас конкретної лінії. Підкласи Line.Info включають Port.Info та DataLine.Info, які отримують деталі, що стосуються конкретного порту та лінії передачі даних відповідно.

**Отримання мікшера**

Зазвичай, одна з перших речей, які програма, яка використовує Java Sound API, має зробити, це отримати мікшер або принаймні одну лінію мікшера, щоб ми могли передавати звук на комп’ютер або з нього. Вашій програмі може знадобитися певний тип мікшера, або ми можемо відобразити список усіх доступних мікшерів, щоб користувач міг вибрати один. AudioSystem пропонує наступний метод:

static Mixer.Info[] getMixerInfo()

Кожен об’єкт Mixer.Info, повернутий цим методом, визначає один тип встановленого мікшера. Mixer.Info містить такі рядки для визначення типу мікшера: ім'я, версія, продавець, опис.

Коли відповідний мікшер знайдено, програма викликає наступний метод AudioSystem, щоб отримати потрібний мікшер:

static Mixer getMixer(Mixer.Info info)

**Отримання лінії потрібного типу**

Є два способи отримати лінію:

* Безпосередньо з об’єкта AudioSystem
* З мікшера, який ви вже отримали з об’єкта AudioSystem

**Отримання лінії безпосередньо з AudioSystem**

Можна використати метод AudioSystem:

static Line getLine(Line.Info info)

який аналогічний методу getMixer, описаному вище.

Line.Info є абстрактним класом, тому ви використовуєте один із його підкласів (Port.Info або DataLine.Info), щоб отримати лінію. Наступний уривок коду використовує підклас DataLine.Info для отримання та відкриття цільового рядка даних:

TargetDataLine line;

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

format); // format is an AudioFormat object

if (!AudioSystem.isLineSupported(info)) {

// обробка помилки.

}

// Отримуємо та відкриваємо лінію.

try {

line = (TargetDataLine) AudioSystem.getLine(info);

line.open(format);

} catch (LineUnavailableException ex) {

// обробка помилки.

//...

}

Цей код отримує об’єкт TargetDataLine без вказівки будь-яких атрибутів, крім його класу та аудіоформату. Ми можемо використовувати аналогічний код для отримання інших видів ліній.

Для Port можна використовувати статичні екземпляри Port.Info у такому коді:

if (AudioSystem.isLineSupported(Port.Info.MICROPHONE)) {

try {

line = (Port) AudioSystem.getLine(

Port.Info.MICROPHONE);

}

}

Використовуємо метод isLineSupported, щоб побачити, чи мікшер взагалі має рядок потрібного типу.

Згадаємо, що вихідна лінія є входом для мікшера, а саме об’єкта Port, якщо мікшер представляє пристрій аудіовходу, і об’єкт SourceDataLine або Clip, якщо мікшер представляє пристрій аудіовиведення. Подібним чином цільова лінія є виходом мікшера: об’єкт Port для мікшера аудіовиходу та об’єкт TargetDataLine для мікшера аудіовходу. Що робити, якщо мікшер взагалі не підключається до жодного зовнішнього апаратного пристрою? Наприклад, розглянемо внутрішній або лише програмний мікшер, який отримує аудіо з прикладної програми та повертає змішане аудіо назад до програми. Цей вид мікшера має об’єкти SourceDataLine або Clip для рядків введення та об’єкти TargetDataLine для вихідних рядків.

Ми також можемо використовувати наступні методи AudioSystem, щоб дізнатися більше про вихідні та цільові лінії визначеного типу, які підтримуються будь-яким встановленим мікшером:

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

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

**Отримання лінії з мікшера**

Інтерфейс мікшера включає варіанти методів доступу AudioSystem для вихідних і цільових ліній, описаних вище. Ці методи Mixer включають ті, які приймають аргументи Line.Info, як і методи AudioSystem. Однак Mixer також включає варіанти, які не приймають аргументів:

Line.Info[] getSourceLineInfo()

Line.Info[] getTargetLineInfo()

Ці методи повертають масиви всіх об’єктів Line.Info для конкретного мікшера. Отримавши масиви, ми можемо перебирати їх, викликаючи метод getLine в Mixer для отримання кожного рядка, а потім метод open в Line для резервування використання кожного рядка для нашої програми.

**5. Відтворення аудіо**

Для відтворення звуку можна використовувати два типи ліній: Clip і SourceDataLine. Головна відмінність між ними полягає в тому, що за допомогою Clip ми вказуємо всі звукові дані одночасно перед відтворенням, тоді як за допомогою SourceDataLine ми продовжуємо записувати нові буфери даних безперервно під час відтворення.

**Використання Clip**

Отримуємо Clip, як описано раніше в розділі «Отримання лінії потрібного типу» в розділі 4. «Доступ до ресурсів аудіосистеми»: створюємо об’єкт DataLine.Info з Clip.class для першого аргументу та передаємо цей DataLine.Info як аргумент для методу getLine в AudioSystem або в Mixer.

**Налаштування Clip для відтворення**

Отримання лінії просто означає, що ми отримали спосіб посилатися на неї; getLine фактично не резервує лінію для нас. Оскільки мікшер може мати обмежену кількість доступних ліній потрібного типу, може статися так, що після того, як ми викличемо getLine для отримання кліпу, інша прикладна програма підскочить і захопить кліп, перш ніж ми будемо готові розпочати відтворення. Щоб фактично використовувати кліп, нам потрібно зарезервувати його для виключного використання нашою програмою, викликавши один із таких методів Clip:

void оpеn(AudioInputStream input\_stream)

void оpen(AudioFormat format, bytе[] data, int offset, int buffer\_size)

Незважаючи на аргумент bufferSize у другому відкритому методі вище, Clip (на відміну від SourceDataLine) не містить методів для запису нових даних у буфер. Аргумент buffer\_size тут просто вказує, скільки байтового масиву завантажувати в кліп. Це не буфер, у який ми можемо згодом завантажити більше даних, як це можна зробити з буфером SourceDataLine.

Відкривши кліп, ми можемо вказати, в якій точці даних він має почати відтворення, використовуючи методи setFramePosition або setMicroSecondPosition в Clip. Інакше все почнеться спочатку. Ми також можемо налаштувати повторне відтворення за допомогою методу setLoopPoints.

**Початок і зупинка відтворення**

Коли ми будемо готові розпочати відтворення, просто викличемо метод start. Щоб зупинити або призупинити кліп, викличемо метод stop, а щоб відновити відтворення, викличемо знову start. Кліп запам’ятовує позицію медіафайлу, де було зупинено відтворення, тому немає потреби в явних методах паузи та відновлення. Якщо ми не хочемо, щоб він продовжувався з того місця, де зупинився, ми можемо "перемотати" кліп на початок (або в будь-яке інше положення, якщо на те пішло), використовуючи згадані вище методи позиціонування за кадром або мікросекундою.

Рівень гучності та стан активності кліпу (активний чи неактивний) можна контролювати, викликавши методи DataLine: getLevel і isActive відповідно. Активний кліп – це той, який наразі відтворює звук.

**Використання SourceDataLine**

Отримання SourceDataLine подібне до отримання Clip.

**Налаштування SourceDataLine для відтворення**

Відкриття SourceDataLine також подібне до відкриття Clip, оскільки метою є знову зарезервувати лінію. Однак ми використовуємо інший метод, успадкований від DataLine:

void open(AudioFormat format)

Зауважимо, що коли ми відкриваємо SourceDataLine, ми ще не пов’язуємо жодних звукових даних із лінією, на відміну від відкриття Clip. Натомість ми просто вказуємо формат аудіоданих, які хочемо відтворити. Система вибирає довжину буфера за замовчуванням.

**Початок і зупинка відтворення**

Коли SourceDataLine відкрито, ми можемо почати відтворення звуку. Ми робимо це, викликаючи метод запуску DataLine, а потім багаторазово записуючи дані в буфер відтворення рядка.

Метод start дозволяє лінії розпочати відтворення звуку, щойно в її буфері з’являться будь-які дані. Ми розміщуємо дані в буфері таким способом:

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

Зміщення в масиві (offset) виражається в байтах, як і довжина массиву (length).

Ось приклад ітерації фрагментів даних, які зчитуються з потоку, записуючи один фрагмент за раз у SourceDataLine для відтворення:

line

line.start();

while (total < total\_to\_read && !stopped)}

num\_of\_bytes\_read = stream.read(myData, 0, num\_of\_bytes\_to\_read);

if (num\_of\_bytes\_read == -1) break;

total += num\_of\_bytes\_read;

line.write(myData, 0, num\_of\_bytes\_read);

}

Оскільки write закінчується до завершення відтворення всіх даних, як дізнатися, коли відтворення фактично завершено? Один із способів — викликати метод drain з DataLine після запису даних останнього буфера. Цей метод блокується, доки не буде відтворено всі дані. Коли контроль повертається до нашої програми, ми можемо звільнити рядок, якщо бажаємо, не боячись передчасно припинити відтворення будь-яких аудіо зразків:

line.write(b, offset, num\_of\_bytes\_to\_write);

line.drain();

line.stop();

line.close();

line = null;

SourceDataLine генерує подію STOP кожного разу, коли потік даних було зупинено, незалежно від того, чи була ця зупинка ініційована методом drain, stop, або flush, або через те, що кінець буфера відтворення було досягнуто до того, як прикладна програма викликала запис знову, щоб надати нові дані. Подія STOP не обов’язково означає, що було викликано метод stop, і це не обов’язково означає, що наступний виклик isRunning поверне false. Однак це означає, що isActive поверне false. (Коли було викликано метод start, метод isRunning поверне true, навіть якщо згенеровано подію STOP, і він почне повертати false лише після виклику методу  stop.) Важливо розуміти, що події START і STOP відповідають на isActive, а не на isRunning.

**6. Запис аудіо**

Типова система аудіовведення в реалізації Java Sound API складається з:

1. Вхідний порт, як-от мікрофонний або лінійний порт, який подає вхідні аудіодані в мікшер
2. Мікшер, який розміщує вхідні дані в одній або кількох цільових лініях даних
3. Одна або кілька цільових ліній даних, з яких програма може отримати дані.

Зазвичай одночасно може бути відкритий лише один вхідний порт, але також можливий мікшер аудіовходу, який змішує аудіо з кількох портів. Інший сценарій складається з мікшера, який не має портів, але отримує аудіовхід через мережу.

TargetDataLine  є прямим аналогом інтерфейсу SourceDataLine, який був описаний в розділі 5. «Відтворення аудіо». Нагадаємо, що інтерфейс SourceDataLine складається з:

* Метод write для надсилання аудіо на мікшер
* Метод available для визначення того, скільки даних можна записати в буфер без блокування

Аналогічно, TargetDataLine складається з:

* Метод read  для отримання аудіо з мікшера
* Метод available  для визначення того, скільки даних можна прочитати з буфера без блокуванн

**Налаштування TargetDataLine**

Процес отримання цільової лінії даних був описаний у Розділі 4. «Доступ до ресурсів аудіосистеми», але ми повторимо його тут для зручності:

TargetDataLine line;

DataLine.Info info = new DataLine.Info(TargetDataLine.class,   
 format); // format is an AudioFormat object

if (!AudioSystem.isLineSupported(info)) {

// обробка помилки ...

}

// Отримуємо та відкриваємо лінію..

try {  
 line = (TargetDataLine) AudioSystem.getLine(info);

line.open(format);

} catch (LineUnavailableException ex) {

// обробка помилки ...

}

Як показано в цьому прикладі, як тільки ми отримали цільову лінію даних, ми резервуємо її для використання нашою програмою, викликавши відкритий метод DataLine, точно так, як було описано у випадку вихідного рядка даних у розділі 5. «Відтворення аудіо». Версія методу open з одним параметром призводить до того, що буфер рядка має розмір за замовчуванням. Натомість ми можемо встановити розмір буфера відповідно до потреб нашої програми, викликавши версію з двома параметрами:

void open(AudioFormat format, int bufferSize)

Вибираючи розмір буфера, пам’ятаймо про компроміс між затримками, спричиненими довгими буферами, з одного боку, та ризиком розривів аудіо, якщо буфери настільки короткі, що ми не можемо отримати дані достатньо швидко, з іншого.

**Зчитування даних з TargetDataLine**

Коли лінію відкрито, вона готова до збору даних, але ще не активна. Щоб фактично розпочати запис аудіо, скористаємося методом start з DataLine. Це починає доставляти вхідні аудіодані до буфера рядка для читання нашою програмою. Наша програма має викликати start лише тоді, коли вона готова розпочати читання з рядка, в іншому випадку багато часу витрачається на заповнення буфера захоплення лише для його переповнення (тобто скидання даних).

Щоб почати отримувати дані з буфера, викличемо метод read з TargetDataLine:

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

Цей метод намагається зчитати довжину байтів даних у масив b, починаючи з позиції байта offset в масиві. Метод повертає кількість фактично прочитаних байтів.

Як і у випадку з методом write в SourceDataLine, ми можемо запросити більше даних, ніж фактично поміщається в буфері, оскільки метод блокується, доки не буде доставлено необхідну кількість даних, навіть якщо ми запитуємо багато даних у буферах.

Щоб уникнути зависання програми під час запису, ми можемо викликати метод читання в циклі, доки не отримаємо весь вхідний аудіо, як у цьому прикладі:

// Припустимо, що TargetDataLine, лінію,вже отримано і відкрито

ByteArrayOutputStream out = new ByteArrayOutputStream();

int num\_of\_bytes\_read;

byte[] data = new byte[line.getBufferSize() / 5];

// Початок захоплення аудіо.

line.start();

// Тут stopped — це глобальне логічне значення, встановлене іншим потоком.

while (!stopped) {

// Зчитуємо наступний фрагмент даних із TargetDataLine.

num\_of\_bytes\_read = line.read(data, 0, data.length);

// Збереження цієї частини даних.

out.write(data, 0, num\_of\_bytes\_read);

Звернемо увагу, що в цьому прикладі розмір масиву байтів, у який зчитуються дані, встановлено рівним одній п’ятій розміру буфера рядка. Якщо натомість ми зробимо його таким же великим, як буфер рядка, і спробуємо прочитати весь буфер, нам потрібно бути дуже точним з вибором часу, тому що дані будуть скинуті, якщо мікшеру потрібно доставити дані в рядок, поки ми читаємо з нього . Використовуючи частину розміру буфера рядка, як показано тут, наша програма буде більш успішною у спільному доступі до буфера рядка зі змішувачем.

Метод read з TargetDataLine приймає три аргументи: масив байтів, зміщення в масиві та кількість байтів вхідних даних, які ми хочемо прочитати. У цьому прикладі третій аргумент — це просто довжина нашого байтового масиву. Метод read повертає кількість байтів, які фактично були зчитані у наш масив.

**7. Обробка звуку за допомогою елементів керування**

Існує два способи обробки сигналу:

* Ми можемо використовувати будь-яку обробку, яка підтримується мікшером або його компонентними лініями, викликаючи об’єкти керування, а потім встановлюючи елементи керування за бажанням користувача. Типові елементи керування, які підтримуються мікшерами та лініями, включають елементи керування посиленням, панорамуванням і реверберацією.
* Якщо тип обробки, який нам потрібен, не забезпечується мікшером або його лініями, наша програма може працювати безпосередньо з аудіобайтами, маніпулюючи ними за бажанням.

**Введення до засобів керування**

Мікшер може мати різні види керування обробкою сигналу на деяких або всіх його лініях. Наприклад, мікшер, який використовується для захоплення аудіо, може мати вхідний порт із регулятором посилення та цільові лінії даних із регуляторами посилення та панорамування. Мікшер, який використовується для відтворення аудіо, може мати елементи керування частотою дискретизації на своїх лініях вихідних даних. У кожному випадку доступ до елементів керування здійснюється за допомогою методів інтерфейсу Line.

Оскільки інтерфейс Mixer розширює Line, сам мікшер може мати власний набір елементів керування. Вони можуть служити основними елементами керування, що впливають на всі вихідні або цільові лінії мікшера. Наприклад, мікшер може мати основний регулятор посилення, значення якого в децибелах додається до значень окремих регуляторів посилення на його цільових лініях.

Якщо мікшер або будь-яка з його ліній мають елементи керування, ми можемо відобразити елементи керування через графічні об’єкти в інтерфейсі користувача нашої програми, щоб користувач міг регулювати звукові характеристики за бажанням. Самі елементи керування не є графічними, вони просто дозволяють отримати та змінити їхні налаштування. Нам вирішувати, які графічні представлення (повзунки, кнопки тощо), якщо такі є, використовувати у нашій програмі.

Усі елементи керування реалізовані як конкретні підкласи абстрактного класу Control. Багато типових елементів керування обробкою аудіо можна описати абстрактними підкласами Control на основі типу даних (наприклад, boolean, enumerated або float). Boolean елементи керування, наприклад, представляють елементи керування двійковим станом, такі як елементи керування ввімкненням/вимкненням для вимкнення звуку або реверберації. Float елементи керування, з іншого боку, добре підходять для представлення безперервно змінних елементів керування, таких як панорамування, баланс або гучність.

Бібліотека Java Sound визначає такі абстрактні підкласи Control:

* BooleanControls — представляє елемент керування за допомогою станів (true або false). Наприклад, перемикачі відключення звуку, соло та ввімкнення/вимкнення.
* FloatControl —модель даних, яка забезпечує контроль над діапазоном значень з плаваючою крапкою. Наприклад, об’єм і панорамування є FloatControl, якими можна керувати за допомогою повзунка.
* EnumControl — пропонує вибір із набору об’єктів. Наприклад, ми можемо пов’язати набір кнопок в інтерфейсі користувача з EnumControl для вибору одного з кількох попередньо встановлених параметрів реверберації.
* CompoundControl — надає доступ до колекції пов’язаних елементів, кожен з яких сам є екземпляром підкласу Control. CompoundControl представляють багатоконтрольні модулі, такі як графічні еквалайзери (графічний еквалайзер, як правило, зображується набором повзунків, кожен з яких впливає на FloatControl).

У наведеній нижче таблиці показано кожен підклас Control, його відповідний підклас Control.Type і статичні екземпляри, які вказують на конкретні види елементів керування:

|  |  |  |
| --- | --- | --- |
| Control | Control.Type | Control.Type екземпляри |
| BooleanControl | BooleanControl.Type | MUTE – переводить стан лінії в беззвучний (mute)  APPLY\_REVERB - eвімкнення/вимкнення реверберації |
| CompoundControl | CompoundControl.Type | - |
| EnumControl | EnumControl.Type | REVERB - доступ до налаштувань реверберації (кожен є екземпляром ReverbType) |
| FloatControl | FloatControl.Type | AUX\_RETURN – допоміжне зворотне посилення на лінії  AUX\_SEND – підсилення допоміжного відправлення на лінії  BALANCE – баланс гучності ліворуч-праворуч  MASTER\_GAIN – загальний приріст на лінії  PAN – позиція ліво-право  REVERB\_RETURN – підсилення пост-реверберації на лінії  REVERB\_SEND – посилення попередньої реверберації на лінії  SAMPLE\_RATE – частота дискретизації відтворення  VOLUME – обсяг на лінії |

**Отримання лінії, яка має бажані елементи керування**

У багатьох випадках програма просто відображатиме будь-які елементи керування, які підтримуються відповідною лінією. Але що, якщо важливо знайти лінію, яка має певні елементи керування? У такому випадку ми можемо використати Line.Info, щоб отримати лінію з потрібними характеристиками, як описано в розділі «Отримання лінії потрібного типу» у розділі 4. «Доступ до ресурсів аудіосистеми».

Наприклад, припустімо, що ми надаємо перевагу вхідному порту, який дозволяє користувачеві встановлювати гучність вхідного звуку. Наступний уривок коду показує, як можна запитати мікшер за замовчуванням, щоб визначити, чи має він потрібний порт і елемент керування:

Port lineIn;

FloatControl volCtrl;

try {

mixer = AudioSystem.getMixer(null);

lineIn = (Port)mixer.getLine(Port.Info.LINE\_IN);

lineIn.open();

volCtrl = (FloatControl) lineIn.getControl(  
 FloatControl.Type.VOLUME);

// Припускаючи, що виклик getControl завершився успішно,

// тепер у нас є регулятор VOLUME LINE\_IN.

} catch (Exception e) {

System.out.println("Failed trying to find LINE\_IN"

+ " VOLUME control: exception = " + e);

}

if (volCtrl != null)

// ...

**Отримання елементів керування з лінії**

Припустимо, що наша програма відтворює звук. Ми використовуємо SourceDataLine. Ми можемо отримати доступ до елементів керування лінією, викликавши метод Line:

Control[] getControls()

Потім для кожного з елементів керування в цьому отриманому масиві ми використовуємо такий метод Control, щоб отримати тип елемента керування:

Control.Type getType()

Знаючи конкретний екземпляр Control.Type, наша програма може відобразити відповідний елемент інтерфейсу користувача.

**Використання елемента керування для зміни аудіосигналу**

Приклади того, що можна регулювати в аудіосигналі за допомогою елементів керування:

* Керування беззвучним станом (mute) лінії
* Зміна гучності лінії
* Вибір між різними налаштуваннями реверберації

**Керування беззвучним станом (mute) лінії**

Керування беззвучним станом будь-якого рядка полягає в простому виклику наступного методу BooleanControl:

void setValue(boolean value)

Щоб заглушити сигнал, який проходить через лінію, програма викликає наведений вище метод, вказуючи true як параметр. Щоб вимкнути приглушення, дозволяючи сигналу проходити через лінію, програма викликає метод із параметром false.

**Зміна гучності лінії**

Припустімо, що наша програма пов’язує певний графічний повзунок із регулятором гучності певного рядка. Значення регулятора гучності (наприклад, FloatControl.Type. VOLUME) установлюється за допомогою такого методу FloatControl:

void setValue(float newValue)

Виявивши, що користувач перемістив повзунок, програма отримує поточне значення повзунка та передає його як параметр newValue вищезазначеному методу. Це змінює гучність сигналу, що проходить через лінію.

**Вибір між різними налаштуваннями реверберації**

Припустимо, що в нашій програмі є мікшер із лінією, яка має елемент керування типу EnumControl.Type.REVERB. Виклик методу EnumControl:

java.lang.Objects[] getValues()

Cтворюється масив об’єктів ReverbType. Якщо потрібно, доступ до конкретних налаштувань параметрів кожного з цих об’єктів можна отримати за допомогою таких методів ReverbType:

int getDecayTime()

int getEarlyReflectionDelay()

float getEarlyReflectionIntensity()

int getLateReflectionDelay()

float getLateReflectionIntensity()

Наприклад, якщо програмі потрібен лише один параметр реверберації, який звучить як в печері, вона може перебирати об’єкти ReverbType, доки не знайде той, для якого getDecayTime повертає значення більше 2000.

Зазвичай програма створює елемент інтерфейсу користувача, наприклад перемикач, для кожного з об’єктів ReverbType у масиві, виведеному методом getValues. Коли користувач натискає один з цих перемикачів, програма викликає метод EnumControl

void setValue(java.lang.Object value)

де value встановлено ReverbType, що відповідає щойно натиснутій кнопці. Аудіосигнал, надісланий через лінію, потім буде реверберуватися відповідно до налаштувань параметрів, які становлять поточний ReverbType елемента керування.

**8. Використання конвертерів файлів і форматів**

**Зчитування звукових файлів**

Клас AudioSystem надає два типи служб читання файлів:

* Інформація про формат аудіоданих, що зберігаються у звуковому файлі
* Потік відформатованих аудіоданих, які можна прочитати зі звукового файлу

Перший з них надається трьома варіантами методу getAudioFileFormat:

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

Як згадувалося вище, повернутий об’єкт AudioFileFormat повідомляє тип файлу, довжину даних у файлі, кодування, порядок байтів, кількість каналів, частоту дискретизації та кількість бітів на вибірку.

Другий тип функцій читання файлів забезпечується цими методами AudioSystem:

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

Ці методи надають нам об’єкт (AudioInputStream), який дозволяє читати аудіодані файлу за допомогою одного з методів читання AudioInputStream.

Припустимо, що ми пишемо програму для редагування звуку, яка дозволяє користувачеві завантажувати звукові дані з файлу, відображати відповідну форму хвилі або спектрограму, редагувати звук, відтворювати відредаговані дані та зберігати результат у новому файлі. Або, можливо, наша програма прочитає дані, що зберігаються у файлі, застосує певну обробку сигналу (наприклад, алгоритм, який уповільнює звук, не змінюючи його висоту), а потім відтворить оброблене аудіо. У будь-якому випадку нам потрібно отримати доступ до даних, що містяться в аудіофайлі. Якщо припустити, що наша програма надає користувачеві певні засоби для вибору або визначення вхідного звукового файлу, читання аудіоданих цього файлу включатиме три кроки:

1. Отримання об’єкту AudioInputStream із файлу.
2. Створення байтового масиву, у якому ми будемо зберігати послідовні фрагменти даних із файлу.
3. Багаторазове зчитування байтів із вхідного аудіопотоку в масив. На кожній ітерації робити щось з байтами в масиві (наприклад, ми можемо відтворювати їх, фільтрувати, аналізувати, відображати або записувати в інший файл).

Наведений нижче приклад коду описує ці кроки.

int totalFramesRead = 0;

File fileIn = new File(somePathName);

try {

AudioInputStream audioInputStream =

AudioSystem.getAudioInputStream(fileIn);

int bytesPerFrame =

audioInputStream.getFormat().getFrameSize();

//Встановити розмір буфера в 1024 кадри.

int num\_of\_bytes = 1024 \* bytesPerFrame;

byte[] audioBytes = new byte[num\_of\_bytes];

try {

int num\_of\_bytes\_read = 0;

int num\_of\_frames\_read = 0;

// Спроба прочитати num\_of\_bytes байт із файлу.

while ((num\_of\_bytes\_read =

audioInputStream.read(audioBytes)) != -1) {

// Обчислюємо кількість фактично прочитаних кадрів.

num\_of\_frames\_read = num\_of\_bytes\_read / bytesPerFrame;

totalFramesRead += num\_of\_frames\_read;

}

} catch (Exception ex) {

// обробка помилки...

}

} catch (Exception e) {

// обробка помилки...

}

По-перше, try створює об’єкт AudioInputStream через виклик методу AudioSystem.getAudioInputStream(File). Цей метод прозоро виконує всі тестування, необхідні для визначення того, чи є вказаний файл насправді звуковим файлом типу, який підтримується бібліотекою Java Sound. Якщо файл, який перевіряється (у цьому прикладі fileIn), не є звуковим файлом або є звуковим файлом якогось непідтримуваного типу, виникає виняткова ситуація UnsupportedAudioFileException. Така поведінка зручна тим, що прикладному програмісту не потрібно турбуватися ні про тестування атрибутів файлів, ні про дотримання будь-яких домовленостей про іменування файлів. Замість цього метод getAudioInputStream виконує весь низькорівневий аналіз і перевірку, необхідні для перевірки вхідного файлу.

Потім try створює масив байтів, audioBytes, довільної фіксованої довжини. Ми переконаємося, що його довжина в байтах дорівнює цілій кількості кадрів, щоб ми не прочитали лише частину кадру або, що ще гірше, лише частину вибірки. Цей масив байтів слугуватиме буфером для тимчасового зберігання фрагмента аудіоданих, які зчитуються з потоку. Якби ми знали, що читатимемо лише дуже короткі звукові файли, ми могли б зробити цей масив такої ж довжини, що й дані у файлі, шляхом отримання довжини в байтах із довжини у кадрах, яку повертає метод getFrameLength в AudioInputStream (насправді, натомість ми, мабуть, просто використали б об’єкт Clip). Але щоб уникнути нестачі пам’яті в загальному випадку, ми натомість читаємо файл порціями, по одному буферу за раз.

try містить цикл while, у якому ми зчитуємо аудіодані з AudioInputStream у масив байтів. Ми повинні додати код у цей цикл для обробки аудіоданих у цьому масиві будь-яким способом, який відповідає потребам нашої програми. Якщо ми застосовуємо певний вид обробки сигналу до даних, нам, ймовірно, потрібно буде викликати метод AudioFormat з AudioInputStream далі, щоб дізнатися кількість бітів на вибірку тощо.

**Запис звукових файлів**

Наступний метод AudioSystem створює файл диска певного типу. Файл міститиме аудіодані, які містяться у вказаному AudioInputStream:

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

Зауважте, що другий аргумент має бути одним із типів файлів, які підтримуються системою (наприклад, AU, AIFF або WAV), інакше метод write поверне виключення IllegalArgumentException. Щоб уникнути цього, ми можемо перевірити, чи може певний AudioInputStream бути записаний у певний тип файлу, викликавши цей метод AudioSystem:

static boolean isFileTypeSupported

(AudioFileFormat.Type fileType, AudioInputStream stream)

який поверне значення true, лише якщо певна комбінація підтримується.

Загалом, ми можемо дізнатися, які типи файлів може писати система, викликавши один із цих методів AudioSystem:

static AudioFileFormat.Type[] getAudioFileTypes()

static AudioFileFormat.Type[]

getAudioFileTypes(AudioInputStream stream)

Перший з них повертає всі типи файлів, які система може записати, а другий повертає лише ті, які система може записати з заданого вхідного аудіопотоку.

Наступний уривок демонструє одну техніку створення вихідного файлу з AudioInputStream за допомогою згаданого вище методу запису.

File fileOut = new File(someNewPathName);

AudioFileFormat.Type fileType = fileFormat.getType();

if (AudioSystem.isFileTypeSupported(fileType,

audioInputStream)) {

AudioSystem.write(audioInputStream, fileType, fileOut);

}

Перший оператор зверху створює новий File об’єкт,  fileOut, із вказаним користувачем або програмою шляхом. Другий оператор отримує тип файлу з попередньо існуючого об’єкта AudioFileFormat під назвою fileFormat, який міг бути отриманий з іншого звукового файлу, такого як той, який був прочитаний у розділі «Читання звукових файлів» цієї глави. (Натомість ми можемо надати будь-який підтримуваний тип файлу, який хочемо, замість того, щоб отримувати тип файлу з іншого місця. Наприклад, ми можемо видалити другий оператор і замінити інші два входження fileType у коді вище на AudioFileFormat.Type.WAVE.)

Третій оператор перевіряє, чи можна записати файл зазначеного типу з бажаного AudioInputStream. Як і формат файлу, цей потік міг бути отриманим із попередньо прочитаного звукового файлу. (Якщо так, можливо, ми певним чином обробили або змінили його дані, оскільки в іншому випадку є простіші способи просто скопіювати файл.) Або, можливо, потік містить байти, щойно отримані з мікрофонного входу.

Нарешті, потік, тип файлу та вихідний файл передаються в метод AudioSystem.write для досягнення мети запису файлу.

**Конвертування з одного формату файлу в інший**

У цьому розділі розглядаються основи перетворення типів аудіофайлів у Java Sound. Знову ми представляємо гіпотетичну програму, метою якої, цього разу, є читання аудіоданих із довільного вхідного файлу та запис їх у файл типу AIFF. Звичайно, вхідний файл має бути такого типу, який система здатна читати, а вихідний файл має бути такого типу, який система здатна записувати. (У цьому прикладі ми припускаємо, що система здатна записувати файли AIFF.) У прикладі програми не виконується перетворення формату даних. Якщо формат даних вхідного файлу не може бути представлений як файл AIFF, програма просто повідомить користувача про цю проблему. З іншого боку, якщо вхідний звуковий файл уже є файлом AIFF, програма сповіщає користувача про те, що його не потрібно конвертувати.

Наступна функція реалізує щойно описану логіку:

public void ConvertFileToAIFF(String inputPath,

String outputPath) {

AudioFileFormat inFileFormat;

File inFile;

File outFile;

try {

inFile = new File(inputPath);

outFile = new File(outputPath);

} catch (NullPointerException ex) {

System.out.println("Error: один з параметрів ConvertFileToAIFF нуль!");

return;

}

try {

// тип файлу запиту

inFileFormat = AudioSystem.getAudioFileFormat(inFile);

if (inFileFormat.getType() != AudioFileFormat.Type.AIFF)

{

// inFile не є AIFF, тому давайте конвертуємо його.

AudioInputStream inFileAIS =

AudioSystem.getAudioInputStream(inFile);

inFileAIS.reset(); // перемотування назад

if (AudioSystem.isFileTypeSupported(

AudioFileFormat.Type.AIFF, inFileAIS)) {

// inFileAIS можна конвертувати в AIFF.

// Запишемо AudioInputStream у вихідний файл.

AudioSystem.write(inFileAIS,

AudioFileFormat.Type.AIFF, outFile);

System.out.println("Успішно створено AIFF файл, "

+ outFile.getPath() + ", з "

+ inFileFormat.getType() + " файлу, " +

inFile.getPath() + ".");

inFileAIS.close();

return;

} else

System.out.println("Warning: AIFF конвертування "

+ inFile.getPath()

+ " наразі не підтримується AudioSystem.");

} else

System.out.println("Вхідний файл " + inFile.getPath() +

" - AIFF." + " Конвертація непотрібна.");

} catch (UnsupportedAudioFileException e) {

System.out.println("Error: " + inFile.getPath()

+ " на підтримуваний тип аудіофайлу!");

return;

} catch (IOException e) {

System.out.println("Error: помилка при спробі зчитування "

+ inFile.getPath() + "!");

return;

}

}

Як згадувалося, мета цієї прикладної функції, ConvertFileToAIFF, полягає в тому, щоб запитати вхідний файл, щоб визначити, чи є він звуковим файлом AIFF, і, якщо це не так, спробувати перетворити його на такий, створюючи нову копію зі вказаним шляхом за другим аргументом. Зауважимо, що новий файл імітує формат аудіоданих оригінального вхідного файлу.

Існує кілька методів бібліотеки Java Sound, які використовуються підпрограмою та є вирішальними для перетворення звукових типів файлів. Усі ці виклики методів можна знайти у другому try вище, вони включають наступне:

* AudioSystem.getAudioFileFormat: використовується тут, щоб визначити, чи є вхідний файл уже типом AIFF. Якщо так, функція швидко повертає результат, інакше спроба перетворення продовжується.
* AudioSystem.isFileTypeSupported: вказує, чи може система записати файл указаного типу, який містить аудіодані з указаного AudioInputStream. У нашому прикладі цей метод повертає true, якщо вказаний вхідний аудіофайл можна перетворити на формат аудіофайлу AIFF. Якщо AudioFileFormat.Type.AIFF не підтримується, ConvertFileToAIFF видає попередження про те, що вхідний файл неможливо перетворити, а потім завершується.
* AudioSystem.write: тут використовується для запису аудіоданих із AudioInputStream inFileAIS у вихідний файл outFile.

Другий із цих методів, isFileTypeSupported, допомагає перед записом визначити, чи можна певний вхідний звуковий файл перетворити на певний вихідний звуковий файл.

**Конвертування звуку між різними форматами даних**

Припустимо, що ми хочемо вказати формат даних вихідного файлу, а також тип файлу. Наприклад, можливо, ми зберігаємо багато довгих файлів для використання в Інтернеті, і нас турбує обсяг простору диску та час завантаження, які потрібні нашим файлам. Ми можемо створити менші файли AIFF, які містять дані з нижчою роздільною здатністю, наприклад, дані з 8-бітним розміром вибірки, частотою дискретизації 8 кГц і одним каналом.

Основний метод для перетворення аудіоданих є в класі AudioSystem. Цей метод є варіантом getAudioInputStream:

AudioInputStream getAudioInputStream(AudioFormat

format, AudioInputStream stream)

Ця функція повертає AudioInputStream, який є результатом перетворення потоку AudioInputStream, використовуючи вказаний  AudioFormat, format. Якщо перетворення не підтримується AudioSystem, ця функція створює виключення IllegalArgumentExceptio.

Щоб уникнути цього, ми можемо спочатку перевірити, чи може система виконати необхідне перетворення, викликавши цей метод AudioSystem:

boolean isConversionSupported(AudioFormat targetFormat,

AudioFormat sourceFormat)

У цьому випадку ми б передали stream.getFormat() як другий аргумент.

Щоб створити певний об’єкт AudioFormat, ми використовуємо один із двох конструкторів AudioFormat, показаних нижче:

AudioFormat(float rate\_of\_sample, int size\_of\_sample\_in\_bits,

int num\_of\_channels, boolean signed, boolean bigEndian)

який створює AudioFormat із лінійним кодуванням PCM і заданими параметрами, або

AudioFormat(AudioFormat.Encoding encoding\_type,

float rate\_of\_sample, int size\_of\_sample\_in\_bits, int num\_of\_channels,

int size\_of\_frame, float rate\_of\_frame, boolean bigEndian)

який також створює  AudioFormat, але дозволяє вказати кодування, розмір кадру та частоту кадрів на додаток до інших параметрів.

Тепер, потрібно виконати бажане конвертування формату аудіоданих із низькою роздільною здатністю. По-перше, ми б створили об’єкт AudioFormat, що описує бажаний вихідний формат аудіоданих. Наступного оператора буде достатньо, і його можна вставити у верхній частині функції:

AudioInputStream lowResAIS;

if (AudioSystem.isConversionSupported(outDataFormat,

inFileAIS.getFormat())) {

lowResAIS = AudioSystem.getAudioInputStream

(outDataFormat, inFileAIS);

}

Таким чином, на цьому етапі процесу ми створили б новий AudioInputStream, отриманий у результаті перетворення оригінального вхідного файлу (у його формі AudioInputStream) у бажаний формат аудіоданих із низькою роздільною здатністю, як визначено outDataFormat.

Останнім кроком для створення бажаного звукового файлу AIFF із низькою роздільною здатністю буде заміна параметра AudioInputStream у виклику AudioSystem.write (тобто першого параметра) нашим перетвореним потоком lowResAIS таким чином:

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

Ці кілька модифікацій нашої попередньої функції створюють щось, що перетворює як аудіодані, так і формат файлу будь-якого вказаного вхідного файлу, звичайно, припускаючи, що система підтримує перетворення.