Rhythminator

CSE1102 Project 06, Spring 2016

Bryan Arnold

4/17/16

TA: Zigeng Wang

Section: 51

Instructor: Jeffrey A. Meunier

**Introduction:**

A Java program that works as a sound sequencer, which is something that plays a sequence of sounds in a rhythmic manner. This sequencer will use a set of pre-defined sounds that are stored in sound files. This program will not generate (or synthesize) its own sounds, nor will it vary the pitch of any of the sounds. The first part of this project will have you create the classes that make the sequencer work. The second part (the project after this one) will have you create the graphics that let the user interact with the sequencer easily. In this project you will be creating the model for this application. The model is how it works, not what it looks like (that's a view) or how the view connects to the model (that's a controller).

**Output:**

There is no actual output that you can see for this project. The output consists of noises being played in a sequence of beats. For this project, the user will hear 16 total beats of various sounds repeated twice for them. These sounds will be played simultaneously in a certain sequence and can be replayed multiple times for the same order of sounds. This is the only output of the program and there are a total of 6 unique sounds played in the sequence that create a rhythm, hence the name Rhythminator.

NOTE: Javadoc was not able to run on my computer, whether it be eclipse or my computer itself, so there is no Javadoc file including in the project itself, although comments are done in Javadoc. Sound files are also not included in submission.

**Source Code:**

**IPublisher Interface**

**package** message;

/\*\*

\* ISubscriber interface

\* Creates the subscription

\* methods for subscribers for later implementation

\* for other classes.

\* **@author** Bryan

\*/

**public** **interface** IPublisher{

/\*\*

\* Subscribe Method: Abstract method for later

\* implementation, allows a subscriber to subscribe

\* to a certain publisher (sequence, broadcaster, etc.)

\* **@param** ISubscriber subscriber. A subscriber

\* is necessary in order to subscribe to a

\* publisher.

\*/

**public** **void** subscribe(ISubscriber subscriber);

/\*\*

\* Unsubscribe Method: Abstract method for later

\* implementation, allows a subscriber to unsibscribe

\* from the certain publisher they are subscribed to.

\* **@param** ISubscriber subscriber. The subscriber

\* who wishes to be unsubscribed from a certain

\* type of publisher must be known in order to

\* unsubscribe them.

\*/

**public** **void** unsubscribe(ISubscriber subscriber);

}

**ISubscriber Interface**

**package** message;

/\*\*

\* ISubscriber interface

\* Creates the notify method for implementation

\* for all classes that need it. (subscribers made)

\* **@author** Bryan Arnold

\*/

**public** **interface** ISubscriber{

/\*\*

\* Notify Method: Abstract method for

\* later implementation. Notifies the subscribers

\* with message once subscribes or for other notifcation

\* reasons.

\* **@param** Message msg. A message must

\* be used in order for a notification

\* to the subscribers to be sent.

\*/

**public** **void** notify(Message msg);

}

**Message Class**

**package** message;

/\*\*

\* This class creates a message notification

\* that is sent out to the subscribers of a certain

\* publisher. Every message comes from a publisher

\* **@author** Bryan Arnold

\*/

**public** **class** Message{

**private** IPublisher \_publisher;

/\*\*

\* Constructor to give a message its respective publisher.

\* **@param** IPublisher publisher. Every message

\* has to come from a certain publisher,

\* so the message needs a publisher origination.

\*/

**public** Message(IPublisher publisher){

\_publisher = publisher;

}

/\*\*

\* **@return** the publisher of the message instance.

\*/

**public** IPublisher getPublisher(){

**return** \_publisher;

}

}

**BroadCaster Class**

**package** message;

/\*\*

\* Broadcasts a message to multiple subscribers.

\* **@author** Bryan Arnold

\*/

**public** **class** Broadcaster **implements** ISubscriber, IPublisher {

**private** ISubscriber[] \_subscribers;

/\*\* Creates an instance of a Broadcaster that can hold up to some number of

\* subscribers.

\* **@param** : int limit. The maximum number of subscribers that this

\* Broadcaster can hold.

\*/

**public** Broadcaster(**int** limit){

**this**.\_subscribers = **new** ISubscriber[limit];

}

/\*\* Subscribe Method: adds a subscriber to the list of subscribers

\* for a specific Broadcaster.

\* **@param**: ISubscriber subscriber.The subscriber who wishes to

\* subscribe to the Broadcaster. A subscriber is needed

\* in order to subscribe to a Broadcaster (no duh).

\*/

@Override

**public** **void** subscribe(ISubscriber subscriber) {

**for**(**int** i = 0; i < \_subscribers.length; i++){

**if**(\_subscribers[i] == **null**){

\_subscribers[i] = subscriber;

**break**;

}

}

}

/\*\* Unsubscribe Method: removes a subscriber from the

\* list of subscribers for the Broadcaster. However this

\* method is not used for the project so it is left blank,

\* but is needed to implement the IPublisher class.

\* **@param**: ISubscriber subscriber. The subscriber who wishes

\* to unsubscribe from the Broadcaster. A subscriber must

\* be known in order to remove them from the subscriber list.

\*/

@Override

**public** **void** unsubscribe(ISubscriber subscriber) {

}

/\*\* Notify Method: notifies the subscribers in the

\* list of subscribers of a notification once

\* it is received.

\* **@param**: Message msg. The notification the method

\* receives that will be sent to all subscribers of

\* the Broadcaster in a message. The message is needed

\* in order to pass it on to the subscribers.

\*/

@Override

**public** **void** notify(Message msg) {

**for**(ISubscriber subscriber : \_subscribers){

**if**(subscriber != **null**){

subscriber.notify(msg);

}

}

}

}

**Sequencer Class**

**package** message;

/\*\*

\* Sends a message to multiple subscribers in order of the subscriber list.

\* It conforms to the ISubscriber and IPublisher interfaces

\* **@author** Bryan Arnold

\*/

**public** **class** Sequencer **implements** ISubscriber, IPublisher {

**private** **int** spot;

**private** ISubscriber[] \_subscribers;

/\*\* Creates an instance of a Sequence that can hold up to some number of

\* subscribers.

\* **@param** : int limit. The maximum number of subscribers that this

\* Sequence can hold.

\*/

**public** Sequencer(**int** limit){

**this**.\_subscribers = **new** ISubscriber[limit];

}

/\*\* Subscribe Method: adds a subscriber to the list of subscribers

\* for the list of subscribers in the specific sequence.

\* **@param**: ISubscriber subscriber.The subscriber who wishes to

\* subscribe to the Broadcaster. A subscriber is needed

\* in order to subscribe to the list of subscribers

\* in the sequence (no duh).

\*/

@Override

**public** **void** subscribe(ISubscriber subscriber) {

**for**(**int** i = 0; i < \_subscribers.length; i++){

**if**(\_subscribers[i] == **null**){

\_subscribers[i] = subscriber;

**break**;

}

}

}

/\*\* Unsubscribe Method: removes a subscriber from the

\* list of subscribers in the Sequence. However this

\* method is not used for the project so it is left blank,

\* but is needed to implement the IPublisher class.

\* **@param**: ISubscriber subscriber. The subscriber who wishes

\* to unsubscribe from the list of subscribers

\* in the sequence. A subscriber must

\* be known in order to remove them from the subscriber list.

\*/

@Override

**public** **void** unsubscribe(ISubscriber subscriber) {

}

/\*\* Notify Method: notifies the subscribers in the

\* list of subscribers one at a time in a

\* sequence.

\* **@param**: Message msg. The notification the method

\* receives that will be sent to a subscriber

\* and then passed on one at a time in a sequence. The message is needed

\* in order to pass it on to the subscribers.

\*/

@Override

**public** **void** notify(Message msg) {

**if**(spot == \_subscribers.length){

spot = 0;

}

**if** (\_subscribers[spot] != **null**){

\_subscribers[spot].notify(msg);

}

spot = spot + 1;

}

}

**Clock Class (provided to us)**

**package** model;

**import** java.util.ArrayList;

**import** java.util.Timer;

**import** java.util.TimerTask;

**import** message.IPublisher;

**import** message.ISubscriber;

**import** message.Message;

/\*\*

\* A Clock instance is a free-running asynchronous timer. It conforms to the

\* IPublisher interface.

\* **@author** jeff, credit goes to him for this class

\*/

**public** **class** Clock **extends** TimerTask **implements** IPublisher{

**private** ArrayList<ISubscriber> \_subscribers = **new** ArrayList<ISubscriber>();

**private** Timer \_timer = **new** Timer();

**private** **long** \_delay = 500;

**private** **int** \_numTicksLeft = 0;

**private** Message \_msg = **new** Message(**this**); // create one message to keep re-using

**public** Clock(){

// nothing to see here

}

/\*\*

\* This method is called automatically by the timer at each clock tick.

\* If you're thinking about calling this method, don't. Call tick() instead.

\* (Yes, it's like having two doorways that lead into the same room. This

\* doorway is not for you.)

\*/

@Override

**public** **void** run(){

tick();

}

/\*\*

\* Sets the delay of this clock. By default, the clock is set to tick

\* once every 500 mS (which is every 1/2 second).

\* **@param** delay The number of milliseconds to delay between ticks.

\*/

**public** **void** setDelay(**long** delay){

\_delay = delay;

}

/\*\*

\* Starts the clock running for an unbounded number of ticks.

\* To stop the clock, call the stop() method.

\*/

**public** **void** start(){

**this**.start(-1);

}

/\*\*

\* Starts the clock running for a specific number of ticks.

\* **@param** numTicks

\*/

**public** **void** start(**int** numTicks){

\_numTicksLeft = numTicks;

\_timer.schedule(**this**, 0, \_delay);

}

/\*\*

\* Stops a running clock.

\*/

**public** **void** stop(){

\_timer.cancel();

}

/\*\*

\* Adds a subscriber to the list of subscribers for this clock.

\* Each time the clock ticks, all subscribers will be notified of the clock tick.

\* **@param** subscriber

\*/

**public** **void** subscribe(ISubscriber subscriber){

\_subscribers.add(subscriber);

}

/\*\*

\* This method is called by the run() method.

\* You may call this method to test or simulate a single clock tick.

\*/

**public** **void** tick(){

**if**(\_numTicksLeft == 0)

**this**.stop();

**else**{

// you can un-comment this next line if you want to see when the clock ticks

//System.out.println("Clock.tick(" + \_numTicksLeft + ")");

**for**(ISubscriber subscriber : \_subscribers)

subscriber.notify(\_msg);

**if**(\_numTicksLeft > 0)

\_numTicksLeft--;

}

}

/\*\*

\* Removes a subscriber from the list of subscribers for this clock.

\* **@param** subscriber

\*/

**public** **void** unsubscribe(ISubscriber subscriber){

\_subscribers.remove(subscriber);

}

}

**Sound Class (provided to us)**

**package** model;

**import** java.io.File;

**import** java.io.IOException;

**import** java.util.ArrayList;

**import** java.util.Arrays;

**import** java.util.Hashtable;

**import** java.util.Set;

**import** javax.sound.sampled.AudioInputStream;

**import** javax.sound.sampled.AudioSystem;

**import** javax.sound.sampled.Clip;

**import** javax.sound.sampled.LineUnavailableException;

**import** javax.sound.sampled.UnsupportedAudioFileException;

/\*\*

\* This class represents sounds that can be played. The sounds are stored

\* in sound files on the disk, but get loaded into memory automatically.

\* **@author** jeff, credit goes to him for the class

\*/

**public** **class** Sound{

**private** String \_name;

**private** Clip \_clip;

**public** **static** **final** String ***SOUND\_DIR*** = "sounds";

**private** **static** Hashtable<String, File> *\_soundFiles* = **new** Hashtable<String, File>();

**private** **static** String[] *\_extensions* = {".aiff", ".wav"};

/\*\*

\* Creates a new sound from a sound file on disk. The sound file must be present

\* in the sounds directory.

\* **@param** soundName The base of the file name from which to create the sound.

\*/

**public** Sound(String soundName){

**this**(soundName, *\_soundFiles*.get(soundName));

}

/\*\*

\* Creates a new sound from a sound file. The file can be located anywhere on disk.

\* **@param** soundName What to call the sound.

\* **@param** soundFile The file that refers to the sound file on disk.

\*/

**public** Sound(String soundName, File soundFile){

**if**(soundFile == **null**)

**throw** **new** RuntimeException("there is no sound file having name " + soundName);

\_name = soundFile.getName();

**try**{

\_clip = AudioSystem.*getClip*();

AudioInputStream ais = AudioSystem.*getAudioInputStream*(soundFile);

\_clip.open(ais);

}

**catch**(LineUnavailableException exn){

**throw** **new** RuntimeException("Unable to open the line for audio output. Is another program using it?");

}

**catch**(UnsupportedAudioFileException exn){

System.***out***.println("Unsupported audio format for that file: " + soundFile.toString());

**throw** **new** RuntimeException("Unsupported audio format for that file: " + soundFile.toString());

}

**catch**(IOException exn){

System.***out***.println("Unable to open file: " + soundFile.toString());

**throw** **new** RuntimeException("Unable to open file: " + soundFile.toString());

}

}

/\*\*

\*

\* **@return** Array of all sound files that were found in the sound directory.

\*/

**public** **static** String[] getAllSoundFileNames(){

Set<String> nameSet = *\_soundFiles*.keySet();

String[] names = **new** String[nameSet.size()];

nameSet.toArray(names);

Arrays.*sort*(names);

**return** names;

}

/\*\*

\*

\* **@return** The name of this sound instance.

\*/

**public** String getName(){

**return** \_name;

}

/\*\*

\* Plays the sound. Make sure your speakers are turned up.

\*/

**public** **void** play(){

\_clip.setFramePosition(0); // necessary for re-playing an already-played clip

\_clip.start();

}

/\*\*

\* Scans the default sound directory for sound files.

\*/

**public** **static** **void** scanSoundDir(){

*scanSoundDir*(***SOUND\_DIR***);

}

/\*\*

\* Scans the specified sound directory for sound files.

\* **@param** dirName

\*/

**public** **static** **void** scanSoundDir(String dirName){

ArrayList<String> soundFileNames = **new** ArrayList<String>();

File directory = **new** File(dirName);

File[] files = directory.listFiles();

**for**(File file : files){

String fileName = file.getName();

**for**(String ext : *\_extensions*){

**if**(fileName.endsWith(ext)){

soundFileNames.add(fileName);

String shortName = file.getName();

shortName = shortName.substring(0, shortName.lastIndexOf('.'));

*\_soundFiles*.put(shortName, file);

**break**;

}

}

}

**int** count = *\_soundFiles*.size();

System.***out***.println("Sound.scanSoundDir class found " + count + " sound files in directory '" + ***SOUND\_DIR*** + "'");

*\_soundFiles*.put("(none)", **new** File(""));

}

@Override

**public** String toString(){

**return** "Sound(" + \_name + ")";

}

}

**SoundPlayer Class**

**package** model;

**import** message.ISubscriber;

**import** message.Message;

/\*\*

\* When a subscriber receives a notification, this class

\* plays a sound for the subscriber who is being notified.

\* **@author** Bryan Arnold

\*/

**public** **class** SoundPlayer **implements** ISubscriber {

**private** Sound sound;

/\*\*

\* Constructor to give the sound a name.

\* **@param** name: The name of the sound

\* so that the sound name is known.

\*/

**public** SoundPlayer(String name){

**this**.sound = **new** Sound(name);

}

/\*\*

\* Constructor to give the new sound

\* instance an actual sound.

\* **@param** sound: The sound itself is then

\* applied to the sound that will be used

\* to notify the subscriber.

\*/

**public** SoundPlayer(Sound sound){

**this**.sound = sound;

}

/\*\*

\* Notify Method: plays a sound

\* when a subscriber is notified.

\* **@param** msg: A message being sent to

\* the subscriber must exist for the sound

\* to play once the message is received.

\*/

@Override

**public** **void** notify(Message msg) {

**this**.sound.play();

}

}

**Chord Class**

**package** model;

**import** message.Broadcaster;

/\*\*

\* This class takes certain sounds instances and

\* combines them into one chord. All the sound instances

\* in the chord are played simultaneously. Subclass

\* of the Broadcaster class and utilizes its constructor

\* and subscribe methods.

\* **@author** Bryan Arnold

\*/

**public** **class** Chord **extends** Broadcaster {

/\*\*

\* Constructor to give a chord instance its respective

\* sounds.

\* **@param** Sound[] sounds. Since a chord is a

\* collection of a few sounds, multiple sound

\* instances must be used to create a chord. Each

\* sound is put into the chord and subscribed.

\*/

**public** Chord(Sound[] sounds) {

**super**(sounds.length);

**for**(Sound sounds1 : sounds){

SoundPlayer sp = **new** SoundPlayer(sounds1);

**this**.subscribe(sp);

}

}

}

**SoundBank Class**

**package** model;

/\*\*

\* This class creates a SoundBank instance

\* that collects sounds to become chord instances

\* that the input desires. So, if the user

\* wants specific sounds of the sound collection,

\* this class does just that.

\* **@author** Bryan

\*/

**public** **class** SoundBank {

**private** Sound[] \_sounds;

/\*\*

\* Constructor to create SoundBank instances, or a collection

\* of sounds to be crafted into chords.

\* **@param** Sound[] sounds. A collection of the

\* sounds desired is required in order to be stored

\* into a SoundBank.

\*/

**public** SoundBank(Sound[] sounds){

**this**.\_sounds = sounds;

}

/\*\*

\* Chord Method: creates a chord with the desired

\* sounds that the user wishes from the collection of sounds.

\* **@param** boolean[] soundFlags. This method checks which indexes of

\* the array are true, as those are the spots in the collection of

\* sounds that the user wishes to put into a chord. Thus, true and false

\* collection would be needed reflecting the positions of the desired sounds.

\* **@return** a new Chord instance. This new chord is the indexes of the collection

\* of sounds that were the same element position as the true soundFlags. This

\* chord is now a collection of the desired sounds to be played at once from

\* the sound collection.

\*/

**public** Chord chord(**boolean**[] soundFlags){

**int** counter = 0;

**for**(**int** i = 0; i < soundFlags.length; i++){

**if**(soundFlags[i] == **true**){

counter = counter + 1;

}

}

**int**[] positions = **new** **int**[counter];

Sound[] newSounds = **new** Sound[counter];

**int** n = 0;

**for**(**int** i = 0; i < **this**.\_sounds.length; i++){

**if**(n == soundFlags.length){

**break**;

}

**if**(soundFlags[n] == **false**){

i--;

}

**if**(soundFlags[n] == **true**){

positions[i] = n;

newSounds[i] = **this**.\_sounds[positions[i]];

}

n++;

}

**return** **new** Chord(newSounds);

}

/\*\*

\* Chord method: creates a chord with the desired

\* sounds that the user wishes from the collection of sounds.

\* **@param** int[] soundNums. Instead of boolean values, this method

\* takes in the desired index locations of the desired sounds in the

\* collection. A collection of index locations, integers, would be needed.

\* **@return** a new Chord instance. This new chord instance is the collection

\* of sounds with respect to the given index locations from the parameters. This

\* chord is now a collection of the desired sounds to be played at once from

\* the sound collection.

\*/

**public** Chord chord(**int**[] soundNums){

Sound[] newSounds = **new** Sound[soundNums.length];

**int** n = 0;

**for**(**int** i = 0; i < **this**.\_sounds.length; i++){

**if**(n == soundNums.length){

**break**;

}

**if**(soundNums[n] == i){

newSounds[n] = **this**.\_sounds[i];

n++;

}

}

**return** **new** Chord(newSounds);

}

}

**Main Class**

**import** model.Clock;

**import** message.Sequencer;

**import** model.Sound;

**import** model.SoundBank;

/\*\*

\* Main Class

\* This class implements most of the features

\* of the other classes to their fruition as well

\* as a testing class.

\* **@author** Bryan

\*/

**public** **class** Main {

/\*\*

\* Main Method, this uses the classes created

\* and makes sequence of sixteen beats that

\* are repeated twice.

\* **@param** args

\*/

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

Sound.*scanSoundDir*();

Sound s1 = **new** Sound("Tom1");

Sound s2 = **new** Sound("Fizz");

Sound s3 = **new** Sound("NintendoCore1");

Sound s4 = **new** Sound("NintendoCore2");

Sound s5 = **new** Sound("Snare4");

Sound s6 = **new** Sound("KickDrum5");

SoundBank sb = **new** SoundBank(**new** Sound[]{s1, s2, s3, s4, s5, s6});

Sequencer seq = **new** Sequencer(16);

seq.subscribe(sb.chord(**new** **int**[]{2, 5})); // 1

seq.subscribe(sb.chord(**new** **int**[]{5})); // 2

seq.subscribe(sb.chord(**new** **int**[]{4})); // 3

seq.subscribe(sb.chord(**new** **int**[]{1})); // 4

seq.subscribe(sb.chord(**new** **int**[]{2, 4})); // 5

seq.subscribe(sb.chord(**new** **int**[]{0})); // 6

seq.subscribe(sb.chord(**new** **int**[]{4})); // 7

seq.subscribe(sb.chord(**new** **int**[]{0, 5})); // 8

seq.subscribe(sb.chord(**new** **int**[]{3, 5})); // 9

seq.subscribe(sb.chord(**new** **int**[]{4})); // 10

seq.subscribe(sb.chord(**new** **int**[]{1})); // 11

seq.subscribe(sb.chord(**new** **int**[]{4})); // 12

seq.subscribe(sb.chord(**new** **int**[]{0, 3, 4})); // 13

seq.subscribe(sb.chord(**new** **int**[]{0})); // 14

seq.subscribe(sb.chord(**new** **int**[]{4})); // 15

seq.subscribe(sb.chord(**new** **int**[]{0, 1})); // 16

Clock c = **new** Clock();

c.setDelay(250);

c.subscribe(seq);

c.start(32);

}

}