Laboration 3: Musikbibliotek

Syfte

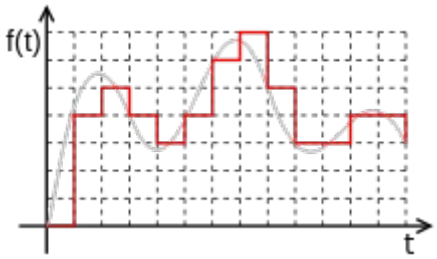
I denna laboration ska ni definiera ett litet bibliotek för att generera musik som kan avlyssnas med vanliga musikprogram. Tiden medger inte något omfattande bibliotek, så den musik vi kan åstadkomma är begränsad. Syftet med denna laboration är att få övning i att använda en-dimensionella fält.

Redovisning

Källkoden för uppgifterna skall lämnas in via Fire senast torsdag 22/10. Lämna in samtliga källkodsfiler som en komprimerad zip-fil.

Förberedelser 1. Snabbkurs i digitalt ljud

Ljud uppstår genom att en tryckvåg fortplantas genom luften; vi har alltså ett analogt fenomen. För behandling i datorer vill vi ha en digital representation av den analoga signalen. Vi betraktar följande figur:



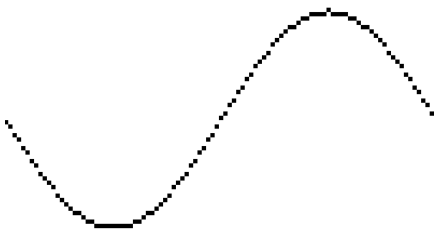
Den grå analoga signalen har *samplats* (avlästs) vid diskreta tidpunkter (de vertikala streckade linjerna). Dessutom har signalvärdena (amplituderna) vid dessa tidpunkter avrundats till närmaste heltalsvärde, representerat av de horisontella streckade linjerna. Den samplade digitala signalen är alltså i detta fall

[0, 4, 5, 4, 3, 4, 6, 7, 5, 3, 3, 4, 4, 3].

En vanlig standard för digital musik, som används för CD-skivor, har samplingsfrekvensen 44100 Hz (dvs antalet avläsningar av signalen per sekund). Noggrannheten i amplitud är typiskt 16 bitar, dvs 2^{16}=65536 olika amplituder kan anges. I vårt bibliotek ska vi sampla signaler med denna frekvens, men representera amplituder med värden av typen **double**, skalade så att amplituden alltid ligger mellan -1 och 1. Först i sista steget, i en klass som ni får given, gör vi om till 16 bitars värden för amplituden. En lämplig datatyp för musik är alltså ett fält av flyttal, dvs **double**[].

Vi kan göra två observationer:

- Det mänskliga örat kan uppfatta ljud ungefär i intervallet 20-20000 Hz. Grundfrekvensen för ettstrukna A (den ton efter vilken instrument stäms) är 440 Hz. När en sinussignal med denna frekvens samplas med samplingsfrekvens 44.1 kHz får man alltså med ca 100 värden per period, dvs den samplade signalen ger en mycket god bild av den analoga signalen. Nedan visas en period av en sinuskurva, samplad med 100 punkter.



  För högre frekvenser blir givetvis upplösningen sämre, men för hörbara signaler har man ändå minst två avläsningar per period.

- Om man sparar en digital signal i en fil med 16 bitar, dvs två bytes, per avläsning och 44100 avläsningar per sekund, så blir det 88 kbytes per sekund, dvs ca 5 Mbytes per minut. Och detta i mono; med två kanaler fördubblas filstorleken till 10 Mbytes per minut. Ljudfiler i detta format blir därför stora. Ofta använder man därför komprimerade format som MP3, vilka tappar en del ljudinformation men ger mycket mindre filer. Vi ska dock i denna laboration hålla oss till okomprimerade filer i formatet WAV.

Förberedelser 2. Tillhandahållna klasser

Skapa en ny mapp för denna laboration och ladda ner filen filesLab3.zip från kurshemsidan till denna mapp.

Bekanta er sedan med det givna programmet:

1. Öppna filen Main.java i IntelliJ. Bygga projektet och kör programmet (med hörlurar på!).

   Ni hör en ren sinussignal med frekvensen 440 Hz i två sekunder, följt av en lika lång signal en oktav högre.

2. Titta på filen Main.java:

```
import java.io.File;
public class Main {
    public static double[] sine(double freq, double duration) {
        int n = (int) (duration*SoundDevice.SAMPLING_RATE);
        double[] a = new double[n];
        double dx = 2*Math.PI*freq / SoundDevice.SAMPLING_RATE;
        for (int i = 0; i < n; i = i + 1) {
            a[i] = Math.sin(i * dx);
        }
        return a;
    }//sine
    public static void main(String[] args) {
      SoundDevice device = new SoundDevice();
      Song song = new Song(5);
      song.add(sine(440, 2));
      song.add(sine(880, 2));
      song.play(device);
      song.save(device.getFormat(), new File("twotones.wav"));
   }//main
} //Main
```

   Vi börjar med att titta på metoden main.

   Först skapas två objekt, en SoundDevice och en Song.

   - Klassen SoundDevice får ni given. Denna klass gör det möjligt att komma åt datorns ljudkort.
   - Klassen Song, som ni också får given, används för att avbilda en sång. Klassen innehåller metoden add för att lägga till toner, metoden play för att spela sången, samt metoden save för att spara sången på en fil.

   Vi läser vidare i main. Två gånger lägger vi till toner genom att anropa song.add. Tonerna som läggs till är resultat av metoden sine (som definieras ovanför main). Metoden sine producerar en sinussignal med given frekvens och längd. Därefter spelas sången (vilket ni hörde) genom att anropa song.play. Med anropet song.save sparas slutligen sången på en fil med namnet "twotones.wav". (Filen kan öppnas och spelas upp med datorns CD-spelare (om en sådan finns). Gör detta och kolla att ni höra de två tonerna.)

   Vi är nu i en vanlig situation. Vi har fått ett program, provkört det, tittat på det och förstår en del av vad som pågår. Men en del är fortfarande konstigt. Vad betyder till exempel 5 i **new** Song(5)?

3. I underkatalogen doc finns dokumentation av de två klasserna Song och SoundDevice. Studera denna dokumentation och källkoden för de båda klasserna.

   Genom att läsa dokumentationen om Song bör ni kunna förstå vad femman betyder.

   För laborationen behöver vi inte förstå hur klassen SoundDevice är uppbyggd. Det räcker att se att konstanten SAMPLING_RATE = 44100 definieras i denna klass.

4. Det återstår att förstå funktionen sine som skapar och fyller ett fält med avläsningar av en ren sinusfunktion. Antalet värden i fältet är antalet avläsningar per sekund (SAMPLING_RATE) gånger tonens längd i sekunder (duration).

   Den funktion vi ska sampla är $s(t) = \sin(2\pi f \cdot t)$, där f är frekvensen (som heter freq i metoden). Ska vi fylla fältet a med SAMPLING_RATE avläsningar av denna funktion så bör elementet med nummer i vara s(i/SAMPLING_RATE), eller $\sin(i \cdot dx)$, där dx definieras som i koden ovan.

Uppgift 1: Ljudet av en sträng

Nu är det dags att skapa ett musikaliskt intressantare ljud än en ren sinuston. Er uppgift är att definiera en metod

public static double[] pluck(**double** freq, **double** duration)

som genererar en ton med den givna frekvensen och varaktigheten och som liknar ljudet då man knäpper på en sträng på ett stränginstrument. För detta finns en berömd metod: Karplus-Strongs algoritm, uppfunnen för cirka 30 år sedan. Notera att metoden pluck har samma parametrar som metoden sine, dvs tonens frekvens och varaktighet. Trots det kommer den att låta annorlunda.

Börja med att skapa en ny klass MusicUtils (i en ny fil MusicUtils.java). Flytta dit sine från Main. När ni flyttat bort `sine` från klassen Main måste ni ändra användningen av den i metoden main; ni måste nu skriva MusicUtils.sine för att java-kompilatorn ska hitta metoden. Gör det, kompilera om och kolla att ni fortfarande kan köra Main och höra de två sinustonerna.

Ni ska nu lägga till pluck till klassen MusicUtils. Karplus-Strongs algoritm fungerar på följande sätt:

Först måste ni deklarera och skapa ett fält av flyttal av lämplig storlek på precis samma sätt som i funktionen sine.

Sedan ska ni fylla ni fältets element med värden. Detta görs i två steg:

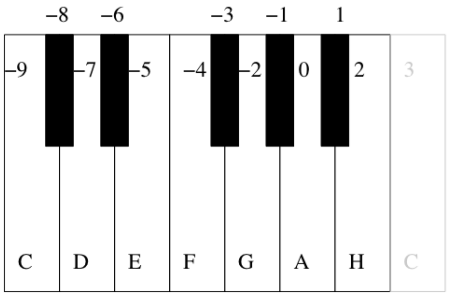
1. Låt `p` vara antalet avläsningar per period, dvs samplingsfrekvensen 44100 dividerat med frekvensen freq för den önskade tonen. Fyll de första p elementen med slumptal i intervallet [-1.0, 1.0[. För att få slumptal använder instansmetoden nextDouble() i klassen Random . Om ni tittar i API'n, ser ni att metoden nextDouble() ger ett ett värde i intervallet [0.0, 1.0[; hur gör man om det till ett tal mellan -1 och 1?
2. Övriga element i fältet, elementen efter de p första elementen, beräknas på följande sätt: elementet med index *i* är summan av elementen med index *i-p* och *i-(p-1)*, multiplicerad med en dämpkonstant *k*. Ett lämpligt värde på *k* är 0.498, men ni får gärna experimentera med olika värden (som ger olika ljud). Intressanta ljud fås bara för *k*-värden litet mindre än 0.5.

Testa er metod genom att ändra i main så att ni anropar pluck i stället för sine. Ändra också namnet på filen där ni sparar sången. Ni bör fortfarande höra två toner, men nu ska de låta som när man knäpper på en sträng.

Vi kan nu se att koden i sine och pluck har samma struktur: vi deklarerar och skapar ett fält, fyller det med innehåll och returnerar det. Innehållet blir olika i de två fallen och vi får därför olika ljud när vi spelar upp filen.

Uppgift 2: Ett bättre sätt att ange toner

Hittills har vi angett toner genom att ge frekvens och varaktighet. Vi vill nu närma oss hur toner anges normalt i musik, dvs med noter i en skala. Vi kan inte gå igenom musikteori här, utan nöjer oss med att konstatera att på till exempel en pianoklaviatur finns tolv tangenter per oktav, sju vita och fem svarta:



Vi skall inte ange toner med de vanliga bokstäverna CDEFGAH, eftersom det kräver förtecken för de svarta tonerna och dessutom något sätt att ange oktav. Istället numrerar vi tonerna med heltal som på klaviaturen ovan, där ettstrukna A ges nummer 0 (vi väljer A som utgångspunkt, eftersom den, med frekvensen 440 Hz, oftast är referenspunkt för musikaliska skalor). Ettstrukna C är alltså -9, tvåstrukna C är 3, trestrukna C är 15, osv. Detta har också fördelen att vi lätt får en direkt formel för frekvensen för en given ton: ton nummer *k* har frekvensen $440*2^{k/12}$ Hz. Ettstrukna C har alltså frekvensen $440*2^{-9/12}$=261.63 Hz.

Er uppgift är nu att utöka MusicUtils med ytterligare en metod

public static double[] note(**int** pitch, **double** duration)

som genererar en ton med givet nummer (anges med parametern pitch) och varaktighet (anges med parametern duration). Observera: Det enda man behöver göra är att räkna ut frekvensen från tonens nummer, anropa `pluck` för att få ett fält och returnera detta fält. I note ska man varken skapa eller fylla fältet; det sköter pluck om.

När ni gjort detta kan ni testa resultatet genom att ändra i main så att programmet spelar till exempel början på Gubben Noak:

```
song.add(MusicUtils.note(-9, 0.4));
song.add(MusicUtils.note(-9, 0.4));
song.add(MusicUtils.note(-9, 0.4));
song.add(MusicUtils.note(-5, 0.4));
song.add(MusicUtils.note(-7, 0.4));
song.add(MusicUtils.note(-7, 0.4));
song.add(MusicUtils.note(-7, 0.4));
song.add(MusicUtils.note(-4, 0.4));
song.add(MusicUtils.note(-5, 0.4));
song.add(MusicUtils.note(-5, 0.4));
song.add(MusicUtils.note(-7, 0.4));
song.add(MusicUtils.note(-7, 0.4));
song.add(MusicUtils.note(-9, 1));
song.play(device);
```

Valet av 0.4 sekunder per ton här är en smaksak; ni kan välja annat tempo om ni så önskar. Vi ser också att det blir mycket omständligt att beskriva musik på detta sätt. Men innan vi ger oss på att åtgärda detta ska vi förbättra ljudet ytterligare.

Uppgift 3: Att blanda ljud

Ett betydligt intressantare ljud fås om vi spelar flera toner samtidigt. Som ett första steg ska ni skriva en metod

public static double[] average(**double**[] t1, **double**[] t2)

som genererar ett medelvärde av tonerna t1 och t2. Vi förutsätter också att de två tonerna har samma varaktighet så att fälten är lika långa. Metoden average ska helt enkelt skapa ett nytt fält och för varje index ta medelvärdet av motsvarande värden i t1 och t2.

Nästa steg är att ni skall skriva en metod

public static double[] harmonic(**int** pitch, **double** duration)

som skapar en ton med givet nummer och varaktighet genom att blanda *tre* toner som skapats med note, nämligen tonerna med nummer pitch, pitch-12 och pitch+12 (dvs också de två tonerna en oktav under och över den avsedda). Förslagsvis blandas först de två sistnämnda tonerna med metoden average och sedan blandas resultatet med den första tonen, återigen med metoden average.

Slutligen kan ni testa resultatet genom att i main ersätta alla note med harmonic.

När ni kommit så här långt har ni definierat en biblioteksklass MusicUtils som innehåller fem funktioner; sine som ni fick given samt pluck, note, average och harmonic som ni definierat själva.

Uppgift 4: Att separera sånger från programmet

I main ovan var beskrivningen av Gubben Noak en del av programmet. Mycket bättre är att spara beskrivningen av sången i en fil noak.txt som börjar

```
-9  0.4
-9  0.4
-9  0.4
-5  0.4
-7  0.4
-7  0.4
-7  0.4
-4  0.4
-5  0.4
-5  0.4
-7  0.4
-7  0.4
-9  0.4
```

och sedan låta programmet läsa denna fil och med hjälp av informationen på varje rad skapa en ton som läggs till sången. Ännu bättre är kanske att ändra varaktigheten för tonerna i filen till 0.25, svarande mot en kvartston, och sedan separat ange tempot.

Ändra nu Main så att en sångbeskrivning läses från en fil genom att ge filnamnet som argument till main-metoden. Koppla sedan filen till ett Scanner-objekt. Detta görs på följande vis

```
public static void main(String[] args) throws FileNotFoundException {
        Scanner sc = new Scanner(new File(args[0]));
        . . .
}//main
```

Klassen Scanner har en konstruktor som göra att Scanner-objektet läser data från en fil. En fil handhas som ett objekt av klassen File och klassen File har en konstruktor vars parameter är namnet på den fysiska filen.

Vid anropet av **new** File(arg[0]) inträffar en exception av typen FileNotFoundException om den angivna fysiska filen inte finns. FileNotFoundException tillhör en typ av exceptions som måste tas hand om eller kastas vidare. Här kastar vi FileNotFoundException vidare, därav står det **throws** FileNotFoundException i metodhuvudet till main-metoden.

I er katalog finns också en fil elise.txt som ni kan använda för testning, om ni inte vill skriva en egen fil för någon musik ni föredrar. Denna fil utnyttjar ovan antydda förbättring; varaktigheten 0.125 betyder en åttondelston. Om man låter detta vara synonymt med varaktigheten hos tonen i sekunder så blir det alldeles för högt tempo. Om en åttondel varar i 0.125 sekunder så hinner man 240 fjärdedelsnoter på en minut; 148 fjärdedelar per minut är mer lagom. En möjlighet är att även ange tempot som argument till main-metoden. Gör det!

Slutkommentarer

Förhoppningsvis har ni nu lärt er en del programmering med fält och en del om digital audio. Avslutningsvis skall påpekas att vi i ett avseende gjort det enkelt för oss: vi har slösat med minne. Vi skapar många och långa fält också för korta ljudsekvenser. Det går att göra det vi gjort, och mer, med mycket mindre minneskonsumtion. För ett allvarligt menat musikprogram måste man vara mer ekonomisk, men för vårt syfte här lämpar sig en mer frikostig attityd till minnesförbrukning bättre.