Structuring Music

CS1316: Representing Structure and Behavior
Story

- Using JMusic
  - With multiple Parts and Phrases
- Creating music objects for exploring composition
  - Version 1: Using an array for Notes, then scooping them up into Phrases.
  - Version 2: Using a *linked list* of song elements.
  - Version 3: General song elements and song phrases
    - Computing phrases
    - Repeating and weaving
  - Version 4: Creating a tree of song parts, each with its own instrument.
JMusic: Java Music library

- JMusic knows about WAV files and many other formats, too (e.g., QuickTime)
- We’ll use it for manipulating **MIDI**
  - Musical Instrument Digital Interface, an industry-standard interface used on electronic musical keyboards and PCs for computer control of musical instruments and devices.
- MIDI is about recording *music*, not *sound*. 
Welcome to DrJava.

```java
> import jm.music.data.*
> import jm.JMC;
> import jm.util.*;
> Note n = new Note(JMC.C4,JMC.QUARTER_NOTE);
> n

jMusic NOTE: [Pitch = 60][RhythmValue = 1.0][Dynamic = 85][Pan = 0.5][Duration = 0.9]
> JMC.C4
60
> JMC.QUARTER_NOTE
1.0
> JMC.QN
1.0
> Note n2 = new Note(64,2.0);
> n2

jMusic NOTE: [Pitch = 64][RhythmValue = 2.0][Dynamic = 85][Pan = 0.5][Duration = 1.8]
```

JMC=JMusic Constants

Makes code easier to read from a music perspective
Creating Phrases

> Phrase phr = new Phrase();
> phr.addNote(n);
> phr.addNote(n2);
> double [] notes1 = {67, 0.25, 64, 0.5, 60, 1.0}
> phr.addNoteList(notes1)
> double [] notes2 = {JMC.G4, JMC.QN, JMC.E4, JMC.EN, JMC.C4, JMC.WN}
> phr.addNoteList(notes2)

Using notes, or an array of note pieces.
Viewing Phrases

> View.notate(phr)
From Viewer: Manipulate and MIDI

- Can save or open MIDI files
- Can change key or time signature.
- Other tools allow changing other characteristics, like tempo.
Different ways of creating Phrases

> Phrase phr2 = new Phrase("Phrase 2", 4.0, JMC.FLUTE);
> phr2.addNoteList(notes2)
A Phrase that starts later

> Phrase phr2 = new Phrase("Phrase 2", 4.0, JMC.FLUTE);
> phr2.addNoteList(notes2)
> View.notate(phr2)
Adding parts into phrases (Wrong way first)

> Part part1 = new Part();
> part1.addPhrase(phr);
> part1.addPhrase(phr2);
> View.notate(part1);

Kinda lost the phrase distinctions.
Building Parts and Scores

> Part partA = new Part("Part A",JMC.PIANO,1)
> partA.addPhrase(phr);
> Part partB = new Part("Part B",JMC.SAX,2)
> partB.addPhrase(phr2);
> Score score1 = new Score("My Two Part Score");
> score1.addPart(partA);
> score1.addPart(partB);
Viewing the Score

> View.notate(score1);
import jm.music.data.*;
import jm.JMC;
import jm.util.*;
import jm.music.tools.*;

public class AmazingGraceSong {
    private Score myScore = new Score("Amazing Grace");

    public void fillMeUp() {
        myScore.setTimeSignature(3, 4);
        double[] phrase1data =
            {JMC.G4, JMC.QN, JMC.C5, JMC.HN, JMC.E5, JMC.EN, JMC.C5, JMC.EN, JMC.E5, JMC.HN, JMC.D5, JMC.QN, JMC.C5, JMC.HN, JMC.A4, JMC.QN, JMC.G4, JMC.HN, JMC.G4, JMC.EN, JMC.A4, JMC.EN, JMC.C5, JMC.HN, JMC.E5, JMC.EN, JMC.C5, JMC.EN, JMC.E5, JMC.HN, JMC.D5, JMC.EN, JMC.E5, JMC.HN, JMC.D5, JMC.EN, JMC.G5, JMC.DHN};
        Phrase myPhrase = new Phrase();
        myPhrase.addNoteList(phrase1data);
        myPhrase.addNoteList(phrase2data);
        Part aPart = new Part("Parts", JMC.FLUTE, 1);
        aPart.addPhrase(myPhrase);
        myScore.addPart(aPart);
    }

    public void showMe() {
        View.notate(myScore);
    }
}

> AmazingGraceSong song1 = new AmazingGraceSong();
> song1.fillMeUp();
> song1.showMe();
Imports and some *private* data

```java
import jm.music.data.*;
import jm.JMC;
import jm.util.*;
import jm.music.tools.*;

public class AmazingGraceSong {
    private Score myScore = new Score("Amazing Grace");
    
    // myScore is *private* instance data
```
Filling the Score

Each array is note, duration, note, duration, note, duration, etc.

I broke it roughly into halves.

class JMC {
public double[] phrase1data = {
JMC.G4, JMC.QN,
JMC.C5, JMC.HN, JMC.E5, JMC.EN, JMC.C5, JMC.EN,
JMC.E5, JMC.HN, JMC.D5, JMC.QN,
JMC.C5, JMC.HN, JMC.A4, JMC.QN,
JMC.G4, JMC.HN, JMC.G4, JMC.EN, JMC.A4, JMC.EN,
JMC.C5, JMC.HN, JMC.E5, JMC.EN, JMC.C5, JMC.EN,
JMC.E5, JMC.HN, JMC.D5, JMC.EN, JMC.E5, JMC.EN,
JMC.G5, JMC.DHN};

double[] phrase2data = {
JMC.G5, JMC.HN, JMC.E5, JMC.EN, JMC.G5, JMC.EN,
JMC.G5, JMC.HN, JMC.E5, JMC.EN, JMC.C5, JMC.EN,
JMC.E5, JMC.HN, JMC.D5, JMC.QN,
JMC.C5, JMC.HN, JMC.A4, JMC.QN,
JMC.G4, JMC.HN, JMC.G4, JMC.EN, JMC.A4, JMC.EN,
JMC.C5, JMC.HN, JMC.E5, JMC.EN, JMC.C5, JMC.EN,
JMC.E5, JMC.HN, JMC.D5, JMC.QN,
JMC.C5, JMC.DHN};

Phrase myPhrase = new Phrase();
myPhrase.addNoteList(phrase1data);
myPhrase.addNoteList(phrase2data);
// create a new part and add the phrase to it
Part aPart = new Part("Parts",
JMC.FLUTE, 1);
aPart.addPhrase(myPhrase);
// add the part to the score
myScore.addPart(aPart);
public void showMe(){
    View.notate(myScore);
};
The Organization of JMusic Objects

Score: timeSignature, tempo, &

Part: Instrument &

Phrase: startingTime &

Note (pitch,duration) Note (pitch,duration)

Note (pitch,duration)

Phrase: startingTime &

Note (pitch,duration) Note (pitch,duration)

Note (pitch,duration)

Phrase: startingTime &

Note (pitch,duration) Note (pitch,duration)

Note (pitch,duration)

Phrase: startingTime &

Note (pitch,duration) Note (pitch,duration)
Thought Experiment

- How are they doing that?
- How can there be any number of Notes in a Phrase, Phrases in a Part, and Parts in a Score?
  - (Hint: They ain’t usin’ arrays!)
How do we *explore* composition here?

- We want to quickly and easily throw together notes in different groupings and see how they sound.
- The current JMusic structure *models* music.
  - Let’s try to create a structure that *models* thinking about music as bunches of *riffs/SongElements* that we want to combine in different ways.
Version 1: Notes in an array

- Let’s just put notes of interest (for now, just random) in an array.
- We’ll *traverse* the array to gather the notes up into a Phrase, then use View to notate the Phrase.
Using an array to structure Notes

> Note [] someNotes = new Note[100];
> for (int i = 0; i < 100; i++)
  {someNotes[i]= new Note((int) (128*Math.random()),0.25);}
> // Now, traverse the array and gather them up.
> Phrase myphrase = new Phrase()
> for (int i=0; i<100; i++)
  {myphrase.addNote(someNotes[i]);}
> View.notate(myphrase);
Critique of Version 1

- So where’s the music?
  - 100 random notes isn’t the issue.
  - It’s that we don’t think about notes as just one long strand.

- Where are the phrases/riffs/elements?
  - We just have one long line of notes.

- How do we explore patterns like this?
  - insertAfter and delete are just as hard here as in sampled sounds!
Version 2: Using a linked list of song elements

- Let’s re-think *Amazing Grace* as a collection of *elements* that we can shuffle around as we’d like.
- We can make any element follow any other element.
What’s in each element?

AmazingGraceSongElement

It **KNOWS**: it’s Part and what comes next

It **CAN DO**: filling itself from the first or second phrase (with a given start time and instrument), setting the next one, getting the next one, and showing (notating) myself and all others.
Welcome to DrJava.
> import jm.JMC;
> AmazingGraceSongElement2 part1 = new 
   AmazingGraceSongElement2();
> part1.setPhrase(part1.phrase1(),0.0,JMC.FLUTE);
> AmazingGraceSongElement2 part2 = new 
   AmazingGraceSongElement2();
> part1.getEndTime()
22.0
> part2.setPhrase(part2.phrase2(),22.0,JMC.PIANO);
> part1.setNext(part2);
> part1.showFromMeOn();
Part1.showFromMeOn()
What’s going on here?

AmazingGraceSongElement part1

myPart: Filled with phrase1(flute))
next: part2

AmazingGraceSongElement part2

myPart: Filled with phrase2(piano)
next: null
Adding a third part

> AmazingGraceSongElement2 part3 = new AmazingGraceSongElement2();
> part3.setPhrase(part3.phrase1(), 0.0, JMC.TRUMPET);
> part1.setNext(part3);
> part3.setNext(part2);
> part1.showFromMeOn();
part1.showFromMeOn();
Now has three parts
What's going on here?

- **AmazingGraceSongElement part1**
  - myPart: Filled with phrase1 (flute)
  - next: part3

- **AmazingGraceSongElement part2**
  - myPart: Filled with phrase2 (piano)
  - next: null

- **AmazingGraceSongElement part3**
  - myPart: Filled with phrase1 (trumpet)
  - next: part2
Introducing the *Linked List*

- A linked list is information broken into smaller pieces, where each piece knows the next piece, but none other.
Another example of a linked list

- Non-linear video editing (like in iMovie)
  - You have a collection of video clips (information)
  - You drag them into a timeline.
    - Each clip still doesn’t know all clips, but it knows the next one.
Why use linked lists versus arrays?

- Just two reasons now, more later:
  1. Can grow to *any* size (well, as long as memory permits)
     - Just create a new element and poke it into the list.
  2. *MUCH* easier to insert!
     - Look at how easily we put part3 between part1 and part2.
Implementing AmazingGraceSongElement2

```java
import jm.music.data.*;
import jm.JMC;
import jm.util.*;
import jm.music.tools.*;

public class AmazingGraceSongElement2 {
    // Every element knows its next element and its part (of the score)
    private AmazingGraceSongElement2 next;
    private Part myPart;

    It's considered good form to make your object's data **private** unless you **need** to make it **public**.
}
Our Constructor

// When we make a new element, the next part is empty, and ours is a blank new part
public AmazingGraceSongElement2(){
    this.next = null;
    this.myPart = new Part();
}

What setPhrase does

// setPhrase takes a phrase and makes it the one for this element
// at the desired start time with the given instrument
public void setPhrase(Phrase myPhrase, double startTime, int instrument) {
    // Phrases get returned from phrase1() and phrase2() with
default (0.0) startTime
    // We can set it here with whatever setPhrase gets as input
    myPhrase.setStartTime(startTime);
    this.myPart.addPhrase(myPhrase);
    this.myPart.setInstrument(instrument);
}

Don’t get hung up on these details—this is just
manipulating the JMusic
classes so that we can store
the information we want.
The Phrases

static public Phrase phrase1() {
    double[] phrase1data =
    {JMC.G4, JMC.QN,
     JMC.C5, JMC.HN, JMC.E5, JMC.EN,
     JMC.C5, JMC.EN,
     JMC.E5, JMC.EN,
     JMC.C5, JMC.HN, JMC.D5, JMC.QN,
     JMC.C5, JMC.EN,
     JMC.E5, JMC.EN,
     JMC.C5, JMC.HN, JMC.D5, JMC.QN,
     JMC.C5, JMC.EN,
     JMC.E5, JMC.EN,
     JMC.C5, JMC.HN, JMC.D5, JMC.QN,
     JMC.C5, JMC.EN,
     JMC.E5, JMC.EN,
     JMC.C5, JMC.HN, JMC.D5, JMC.QN};
    Phrase myPhrase = new Phrase();
    myPhrase.addNoteList(phrase1data);
    return myPhrase;
}

static public Phrase phrase2() {
    double[] phrase2data =
    {JMC.G5, JMC.HN, JMC.E5, JMC.EN, JMC.G5, JMC.EN,
     JMC.G5, JMC.HN, JMC.E5, JMC.EN, JMC.C5, JMC.EN,
     JMC.E5, JMC.HN, JMC.D5, JMC.QN,
     JMC.C5, JMC.EN, JMC.E5, JMC.EN,
     JMC.E5, JMC.HN, JMC.D5, JMC.QN,
     JMC.C5, JMC.EN, JMC.E5, JMC.EN,
     JMC.E5, JMC.HN, JMC.D5, JMC.QN,
     JMC.C5, JMC.DHN};
    Phrase myPhrase = new Phrase();
    myPhrase.addNoteList(phrase2data);
    return myPhrase;
}

Static? This means that we can actually access them without any instances. Is that useful here? Well, not yet…
Handling the linked list

// Here are the two methods needed to make a linked list of elements
public void setNext(AmazingGraceSongElement2 nextOne){
    this.next = nextOne;
}

public AmazingGraceSongElement2 next(){
    return this.next;
}
Controlling access: An accessor method

// We could just access myPart directly
// but we can CONTROL access by using
// (called an accessor)
private Part part(){
    return this.myPart;
}
A little object manipulation

// Why do we need this?
// If we want one piece to start after another, we need
// to know when the last one ends.
// Notice: It's the phrase that knows the end time.
// We have to ask the part for its phrase (assuming only
// one)
// to get the end time.
public double getEndTime(){
    return this.myPart.getPhrase(0).getEndTime();
}
public void showFromMeOn(){
    // Make the score that we'll assemble the elements into
    // We'll set it up with the time signature and tempo we like
    Score myScore = new Score("Amazing Grace");
    myScore.setTimeSignature(3,4);
    myScore.setTempo(120.0);

    // Each element will be in its own channel
    int channelCount = 1;

    // Start from this element (this)
    AmazingGraceSongElement2 current = this;
    // While we're not through...
    while (current != null)
    {
        // Set the channel, increment the channel, then add it in.
        current.setChannel(channelCount);
        channelCount = channelCount + 1;
        myScore.addPart(current.part());

        // Now, move on to the next element
        current = current.next();
    }

    // At the end, let's see it!
    View.notate(myScore);
}

This is called *traversing* the linked list.
The Key Part

// Start from this element (this)
AmazingGraceSongElement2 current = this;
// While we're not through...
while (current != null)
{
    // Set the channel, increment the channel, then add it in.
    //BLAH BLAH BLAH (Ignore this part for now)
    // Now, move on to the next element
    current = current.next();
}

// At the end, let's see it!
View.notate(myScore);
Step 1:

// Start from this element (this)
AmazingGraceSongElement2 current = this;

AmazingGraceSongElement part1
myPart: Filled with phrase1 (flute)
next: part3

AmazingGraceSongElement part2
myPart: Filled with phrase2 (piano)
next: null

AmazingGraceSongElement part3
myPart: Filled with phrase1 (trumpet)
next: part2
Step 2:
// While we're not through...
while (current != null)
{
    //BLAH BLAH BLAH – PROCESS THIS PART

    AmazingGraceSongElement part1
    myPart: Filled with phrase1 (flute)
    next: part3

    current

    AmazingGraceSongElement part2
    myPart: Filled with phrase2 (piano)
    next: null

    AmazingGraceSongElement part3
    myPart: Filled with phrase1 (trumpet)
    next: part2
Step 3:
// Now, move on to the next element
    current = current.next();
};
Step 4:
// While we're not through...
while (current != null)
{
    // BLAH BLAH BLAH – PROCESS THIS PART
    AmazingGraceSongElement part1
    myPart: Filled with phrase1 (flute)
    next: part3

    AmazingGraceSongElement part2
    myPart: Filled with phrase2 (piano)
    next: null

    AmazingGraceSongElement part3
    myPart: Filled with phrase1 (trumpet)
    next: part2

    current
Step 5:
// Now, move on to the next element
current = current.next();
};
Step 6:
// While we're not through...
while (current != null)
{
  //BLAH BLAH BLAH – PROCESS THIS PART

AmazingGraceSongElement part1
myPart: Filled with phrase1 (flute)
next: part3

AmazingGraceSongElement part2
myPart: Filled with phrase2 (piano)
next: null

AmazingGraceSongElement part3
myPart: Filled with phrase1 (trumpet)
next: part2

current
Step 7:
// Now, move on to the next element
current = current.next();
};
Step 8:
// While we're not through...
while (current != null)

- **AmazingGraceSongElement part1**
  - myPart: Filled with phrase1 (flute)
  - next: part3

- **AmazingGraceSongElement part2**
  - myPart: Filled with phrase2 (piano)
  - next: null

- **AmazingGraceSongElement part3**
  - myPart: Filled with phrase1 (trumpet)
  - next: part2

**current**

**STOP THE LOOP!**
Traversing arrays vs. lists

//TRaversing A LIST
  // Start from this element (this)
  AmazingGraceSongElement2
  current = this;
  // While we're not through...
  while (current != null)
  {
    // Set the channel, increment the channel, then add it in.
  //BlaH BlaH BlaH (Ignore this part for now)
    // Now, move on to the next element
    current = current.next();
  }

  // Now, traverse the array and gather them up.
  Phrase myphrase = new Phrase();
  for (int i=0; i<100; i++)
  {
    myphrase.addNote(someNotes[i]);
  }
Inserting into lists

// Here are the two methods needed to make a linked list of elements
public void
setNext(AmazingGraceSongElement2 nextOne)
{
    this.next = nextOne;
}

public
AmazingGraceSongElement2 next()
{
    return this.next;
}

> part1.setNext(part3);
> part3.setNext(part2);
> part1.showFromMeOn();
Inserting into arrays

```java
public void insertAfter(Sound inSound, int start) {
    SoundSample current = null;
    // Find how long insound is
    int amtToCopy = inSound.getLength();
    int endOfThis = this.getLength() - 1;
    // If too long, copy only as much as will fit
    if (start + amtToCopy > endOfThis) {
        amtToCopy = endOfThis - start - 1;
    }

    // ** First, clear out room.
    // Copy from endOfThis- amtToCopy up to endOfThis
    for (int i = endOfThis - amtToCopy; i > start; i--) {
        current = this.getSample(i);
        current.setValue(this.getSampleValueAt(i + amtToCopy));
    }

    // ** Second, copy in inSound up to amtToCopy
    for (int target = start, source = 0; source < amtToCopy; target++, source++) {
        current = this.getSample(target);
        current.setValue(inSound.getSampleValueAt(source));
    }
}
```

> Sound test2 = new Sound("D:/cs1316/MediaSources/thisisatest.wav");
> test.insertAfter(test2, 40000)
> test.play()
More on Arrays vs. Lists

- **Arrays**
  - Much easier to traverse
  - Very fast to access a specific \( n^{\text{th}} \) element
  - But really a pain to insert and delete.
    - Hard to write the code
    - Can take a long time if it’s a big array

- **Lists**
  - More complex to traverse
  - Slower to access a specific element
  - Very easy to insert (and later we’ll see, delete)
    - Simple code
    - Takes no time at all to run
Critique of Version 2

- Lovely *structuring* of data, but just how much can one do with two parts of *Amazing Grace*?
  - We need the ability to have a library of phrases
- But what does the ordering mean? What if we had gone part1->part2->part3 instead?
  - What *should* the order *encode*?
  - Right now, it encodes *nothing*.
- When we’re exploring music, do we really want to worry about instruments and start times for every phrase?
Version 3: SongNode and SongPhrase

- SongNode instances will hold pieces (phrases) from SongPhrase.
- SongNode instances will be the nodes in the linked list
  - Each one will know its next.
- Ordering will encode the order in the Part.
  - Each one will get appended after the last.
Using SongNode and SongPhrase

Welcome to DrJava.
> import jm.JMC;
> SongNode node1 = new SongNode();
> node1.setPhrase(SongPhrase.riff1());
> SongNode node2 = new SongNode();
> node2.setPhrase(SongPhrase.riff2());
> SongNode node3 = new SongNode();
> node3.setPhrase(SongPhrase.riff1());
> node1.setNext(node2);
> node2.setNext(node3);
> node1.showFromMeOn(JMC.SAX);
All three SongNodes in one Part
How to think about it

node1
  myPhrase: riff1
  next: node2

node2
  myPhrase: riff2
  next: node3

node3
  myPhrase: riff1
  next: null
import jm.music.data.*;
import jm.JMC;
import jm.util.*;
import jm.music.tools.*;

public class SongNode {
    /**
     * the next SongNode in the list
     */
    private SongNode next;

    /**
     * the Phrase containing the notes and durations associated with this
     * node
     */
    private Phrase myPhrase;

    SongNode’s know their Phrase and the next node in the list
Constructor for SongNode

/**
 * When we make a new element, the next part is empty, and ours is a blank new part
 */

public SongNode()
{
    this.next = null;
    this.myPhrase = new Phrase();
}
Setting the phrase

/**
 * setPhrase takes a Phrase and makes it the one for this node
 * @param thisPhrase the phrase for this node
 */

public void setPhrase(Phrase thisPhrase){
    this.myPhrase = thisPhrase;
}
Linked list methods

```java
/**
 * Creates a link between the current node and the input node
 * @param nextOne the node to link to
 */
public void setNext(SongNode nextOne){
    this.next = nextOne;
}
/**
 * Provides public access to the next node.
 * @return a SongNode instance (or null)
 */
public SongNode next(){
    return this.next;
}
```
/**
 * Insert the input SongNode AFTER this node,
 * and make whatever node comes NEXT become the next of the input node.
 * @param nextOne SongNode to insert after this one
 */
public void insertAfter(SongNode nextOne)
{
    SongNode oldNext = this.next(); // Save its next
    this.setNext(nextOne); // Insert the copy
    nextOne.setNext(oldNext); // Make the copy point on to the rest
}
Using and tracing insertAfter()

```java
public void insertAfter(SongNode nextOne)
{
    SongNode oldNext = this.next(); // Save its next
    this.setNext(nextOne); // Insert the copy
    nextOne.setNext(oldNext); // Make the copy point on to the rest
}
```

> SongNode nodeA = new SongNode();
> SongNode nodeB = new SongNode();
> nodeA.setNext(nodeB);
> SongNode nodeC = new SongNode()
> nodeA.insertAfter(nodeC);

Traversing
the list

/**
* Collect all the notes from this node on
* in an part (then a score) and open it up for viewing.
* @param instrument MIDI instrument (program) to be used in playing this list
*/
public void showFromMeOn(int instrument)
{
    // Make the Score that we'll assemble the elements into
    // We'll set it up with a default time signature and tempo we like
    // (Should probably make it possible to change these -- maybe with inputs?)
    Score myScore = new Score("My Song");
    myScore.setTimeSignature(3,4);
    myScore.setTempo(120.0);

    // Make the Part that we'll assemble things into
    Part myPart = new Part(instrument);

    // Make a new Phrase that will contain the notes from all the phrases
    Phrase collector = new Phrase();

    // Start from this element (this)
    SongNode current = this;
    // While we're not through...
    while (current != null)
    {
        collector.addNoteList(current.getNotes());

        // Now, move on to the next element
        current = current.next();
    }

    // Now, construct the part and the score.
    myPart.addPhrase(collector);
    myScore.addPart(myPart);

    // At the end, let's see it!
    View.notate(myScore);
    }
}
The Core of the Traversal

// Make a new Phrase that will contain the notes from all the phrases
Phrase collector = new Phrase();

// Start from this element (this)
SongNode current = this;
// While we're not through...
while (current != null)
{
    collector.addNoteList(current.getNotes());

    // Now, move on to the next element
    current = current.next();
}
Then return what you collected

// Now, construct the part and the score.
myPart.addPhrase(collector);
myScore.addPart(myPart);

// At the end, let's see it!
View.notate(myScore);

}
getNotes() just pulls the notes back out

/**
 * Accessor for the notes inside the node's phrase
 * @return array of notes and durations inside the phrase
 */
private Note [] getNotes()
{
    return this.myPhrase.getNoteArray();
}
**SongPhrase**

- SongPhrase is a collection of *static* methods.
- We don’t ever need an instance of SongPhrase.
- Instead, we use it to store methods that return phrases.
  - It’s not very object-oriented, but it’s useful here.
import jm.music.data.*;
import jm.JMC;
import jm.util.*;
import jm.music.tools.*;

public class SongPhrase {
    //Little Riff1
    static public Phrase riff1() {
        double[] phrasedata =
            {JMC.G3, JMC.EN, JMC.B3, JMC.EN, JMC.C4, JMC.EN, JMC.D4, JMC.EN};

        Phrase myPhrase = new Phrase();
        myPhrase.addNoteList(phrasedata);
        return myPhrase;
    }
}
//Little Riff2
static public Phrase riff2() {
    double[] phrasedata =
    {JMC.D4, JMC.EN, JMC.C4, JMC.EN, JMC.E4, JMC.EN, JMC.G4, JMC.EN};

    Phrase myPhrase = new Phrase();
    myPhrase.addNoteList(phrasedata);
    return myPhrase;
}
Computing a phrase

//Larger Riff1
static public Phrase pattern1() {
    double[] riff1data = 
    {JMC.G3, JMC.EN, JMC.B3, JMC.EN, JMC.C4, JMC.EN, JMC.D4, JMC.EN};
    double[] riff2data = 
    {JMC.D4, JMC.EN, JMC.C4, JMC.EN, JMC.E4, JMC.EN, JMC.G4, JMC.EN};

    Phrase myPhrase = new Phrase();
    // 3 of riff1, 1 of riff2, and repeat all of it 3 times
    for (int counter1 = 1; counter1 <= 3; counter1++)
        {for (int counter2 = 1; counter2 <= 3; counter2++)
            myPhrase.addNoteList(riff1data);
            myPhrase.addNoteList(riff2data);
        }
    return myPhrase;
}
The way that we use SongNote and SongPhrase, any method that returns a phrase is perfectly valid SongPhrase method.
10 Random Notes
(Could be less random...)

/*
 * 10 random notes
 ***/
static public Phrase random() {
    Phrase ranPhrase = new Phrase();
    Note n = null;
    
    for (int i=0; i < 10; i++) {
        n = new Note((int) (128*Math.random()),0.1);
        ranPhrase.addNote(n);
    }
    return ranPhrase;
}
/*
 * 10 random notes above middle C
 ***/
static public Phrase randomAboveC() {
    Phrase ranPhrase = new Phrase();
    Note n = null;

    for (int i=0; i < 10; i++) {
        n = new Note((int) (60+(5*Math.random())),0.25);
        ranPhrase.addNote(n);
    }
    return ranPhrase;
}
Going beyond connecting nodes

- So far, we’ve just created nodes and connected them up.
- What else can we do?
- Well, music is about repetition and interleaving of themes.
  - Let’s create those abilities for SongNodes.
Welcome to DrJava.
> SongNode node = new SongNode();
> node.setPhrase(SongPhrase.randomAboveC());
> SongNode node1 = new SongNode();
> node1.setPhrase(SongPhrase.riff1());
> node.repeatNext(node1,10);
> import jm.JMC;
> node.showFromMeOn(JMC.PIANO);
What it looks like

<table>
<thead>
<tr>
<th>node</th>
<th>node1</th>
<th>node1</th>
<th>node1</th>
<th>...</th>
</tr>
</thead>
</table>

[Image of a music notation software interface with a staff and notes]
/**
 * Repeat the input phrase for the number of times specified.
 * It always appends to the current node, NOT insert.
 * @param nextOne node to be copied in to list
 * @param count number of times to copy it in.
 */

public void repeatNext(SongNode nextOne, int count) {
    SongNode current = this; // Start from here
    SongNode copy; // Where we keep the current copy

    for (int i=1; i <= count; i++) {
        copy = nextOne.copyNode(); // Make a copy
        current.setNext(copy); // Set as next
        current = copy; // Now append to copy
    }
}
/**
 * copyNode returns a copy of this node
 * @return another song node with the same notes
 */

public SongNode copyNode(){
    SongNode returnMe = new SongNode();
    returnMe.setPhrase(this.getPhrase());
    return returnMe;
}
public void repeatNext(SongNode nextOne, int count) {
    SongNode current = this; // Start from here
    SongNode copy; // Where we keep the current copy

    // Step through the list
    while (current != null) {
        copy = current; // Copy the current node
        for (int i = 0; i < count; i++) {
            current = current.next; // Move to the next node
            if (current == null) { // If we reach the end of the list
                current = copy; // Start from the copy
            }
        }
    }
}
Step 2:
copy = nextOne.copyNode(); // Make a copy
Step 3:
current.setNext(copy); // Set as next

node

phrase:
10 random notes
next:
copy

node1

phrase:
riff1()
next: null

current

nextOne
Step 4:
current = copy; // Now append to copy

node

phrase: 10 random notes
next:

current

node

phrase: riff1()
next: null

copy

node1

phrase: riff1()
next: null

nextOne
Step 5 & 6:
copy = nextOne.copyNode(); // Make a copy
current.setNext(copy); // Set as next
Step 7 (and so on):
    current = copy; // Now append to copy

node

    phrase:
    10 random notes
    next:

node1

    phrase:
    riff1()
    next:

    phrase:
    riff1()
    next: null

current    copy    nextOne
What happens if the node already points to something?

- Consider `repeatNext` and how it inserts: It simply sets the next value.
- What if the node *already had a next*?
- `repeatNext` will *erase* whatever *used* to come next.
- How can we fix it?
/**
 * Repeat the input phrase for the number of times specified.
 * But do an insertion, to save the rest of the list.
 * @param nextOne node to be copied into the list
 * @param count number of times to copy it in.
 ***/
public void repeatNextInserting(SongNode nextOne, int count){
    SongNode current = this; // Start from here
    SongNode copy; // Where we keep the current copy

    for (int i=1; i <= count; i++)
    {
        copy = nextOne.copyNode(); // Make a copy
        current.insertAfter(copy); // INSERT after current
        current = copy; // Now append to copy
    }
}
public void weave(SongNode nextOne, int count, int skipAmount) {
  SongNode current = this; // Start from here
  SongNode copy; // Where we keep the one to be weaved in
  SongNode oldNext; // Need this to insert properly
  int skipped; // Number skipped currently

  for (int i=1; i <= count; i++) {
    copy = nextOne.copyNode(); // Make a copy

    // Skip skipAmount nodes
    skipped = 1;
    while ((current.next() != null) && (skipped < skipAmount)) {
      current = current.next();
      skipped++;
    }

    oldNext = current.next(); // Save its next
    current.insertAfter(copy); // Insert the copy after this one
    current = oldNext; // Continue on with the rest
  }
}
Creating a node to weave

```
> SongNode node2 = new SongNode();
> node2.setPhrase(SongPhrase.riff2());
> node2.showFromMeOn(JMC.PIANO);
```
Doing a weave

> node.weave(node2,4,2);
> node.showFromMeOn(JMC.PIANO);
Weave Results

Before:

After
public void weave(SongNode nextOne, int count, int skipAmount) {
    SongNode current = this; // Start from here
    SongNode copy; // Where we keep the one to be weaved in
    SongNode oldNext; // Need this to insert properly
    int skipped; // Number skipped currently
for (int i=1; i <= count; i++)
{
    copy = nextOne.copyNode(); // Make a copy

    //Skip skipAmount nodes
    skipped = 1;
    while ((current.next() != null) && (skipped < skipAmount))
    {
        current = current.next();
        skipped++;
    }
};
Then do an insert

if (current.next() == null) // Did we actually get to the end early?
    break; // Leave the loop

oldNext = current.next(); // Save its next
current.insertAfter(copy); // Insert the copy after this one
current = oldNext; // Continue on with the rest
Version 4: Creating a tree of song parts, each with its own instrument

- SongNode and SongPhrase offer us enormous flexibility in exploring musical patterns.
- But it’s only one part!
- We’ve lost the ability of having different parts starting at different time!
- Let’s get that back.
The Structure We’re Creating

Starting to look like a tree…
import jm.music.data.*;
import jm.JMC;
import jm.util.*;
import jm.JMC;

public class MyFirstSong {
    public static void main(String[] args) {
        Song songroot = new Song();

        SongNode node1 = new SongNode();
        SongNode riff3 = new SongNode();
        riff3.setPhrase(SongPhrase.riff3());
        node1.repeatNext(riff3, 16);
        SongNode riff1 = new SongNode();
        riff1.setPhrase(SongPhrase.riff1());
        node1.weave(riff1, 7, 1);
        SongPart part1 = new SongPart(JMC.PIANO, node1);
        songroot.setFirst(part1);

        SongNode node2 = new SongNode();
        SongNode riff4 = new SongNode();
        riff4.setPhrase(SongPhrase.riff4());
        node2.repeatNext(riff4, 20);
        node2.weave(riff1, 4, 5);
        SongPart part2 = new SongPart(JMC.STEEL_DRUMS, node2);
        songroot.setSecond(part2);
        songroot.show();
    }
}