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.**

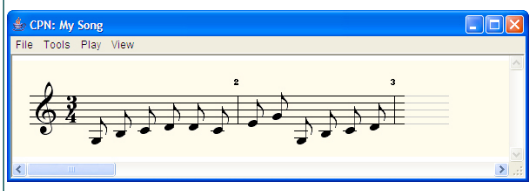
Version 3: SongNode and SongPhrase

- 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.

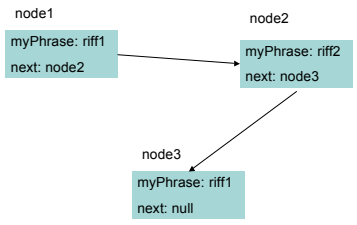
Using SongNode and SongPhrase

```
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



Declarations for SongNode

```
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

```
/**
  * 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;
 }
```

insertAfter

```
/**
  * 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()

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

```
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

 }
```

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.

SongPhrase.riff1()

```
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;
```

SongPhrase.riff2()

```
//Little Riff2
  static public Phrase riff2() {
    double[] phrasedata =

    {JMC.D4,JMC.EN,JMC.C4,JMC.EN,JMC.E4,JMC.EN,JM
    C.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;
}
```

As long as it's a phrase…

- 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 Slightly Less Random Notes

```
/*
   * 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;
  }
```

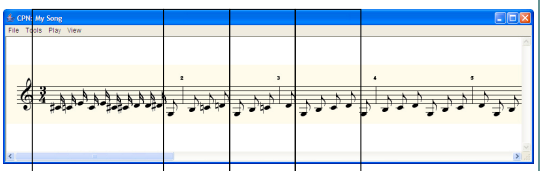
Going beyond connecting nodes

- 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.

Repeating a Phrase

```
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



node | node1 | node1 | node1 ...

Repeating

Note! What happens to this's **next**? How would you create a *looong* repeat chain of *several* types of phrases with this?

```
/**
 * 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
  }
}
```

Here's making a copy

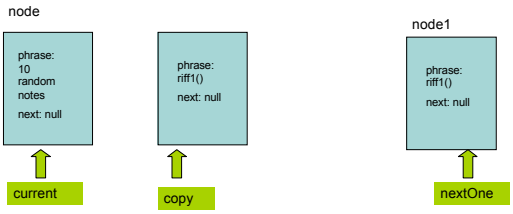
```
/**
 * copyNode returns a copy of this node
 * @return another song node with the same
  notes
 */
public SongNode copyNode(){
  SongNode returnMe = new SongNode();
  returnMe.setPhrase(this.getPhrase());
  return returnMe;
}
```

Step 1:

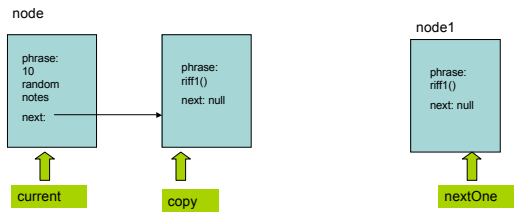
public void repeatNext(SongNode nextOne,int count) {
SongNode current = this; // Start from here
SongNode copy; // Where we keep the current copy

Step 2:

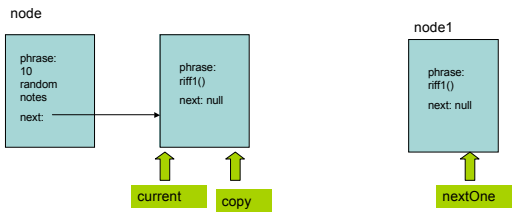
copy = nextOne.copyNode(); // Make a copy



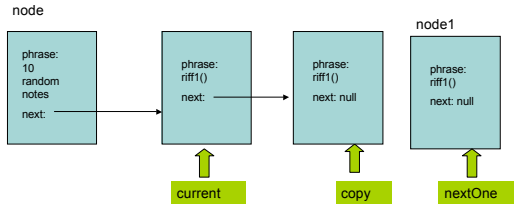
Step 3:
current.setNext(copy); // Set as next



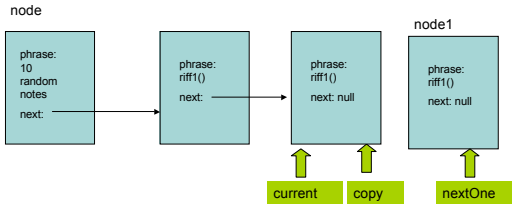
Step 4:
current = copy; // Now append to copy



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



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?

repeatNextInserting

```
/**
  * 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
   }
 }
```

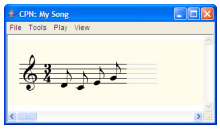
Weaving

```
/**
 * Weave the input phrase count times every skipAmount nodes
 * @param nextOne node to be copied into the list
 * @param count how many times to copy
 * @param skipAmount how many nodes to skip per weave
 */
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
    if (current.next() == null) // Did we actually get to the end early?
       break; // Leave the loop

  }
}
```

Should we break before the last insert (when we get to the end) or after?

Creating a node to weave

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

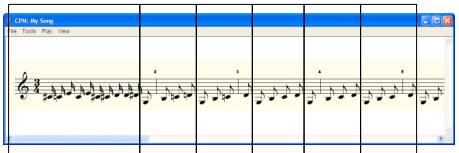


Doing a weave

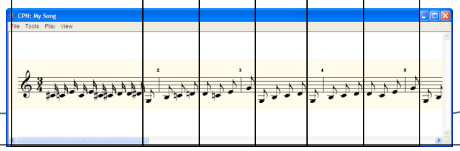
```
> node.weave(node2,4,2);
> node.showFromMeOn(JMC.PIANO);
```

Weave Results

Before:



After



Walking the Weave

```
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
```

Skip forward

```
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++;
     };
```

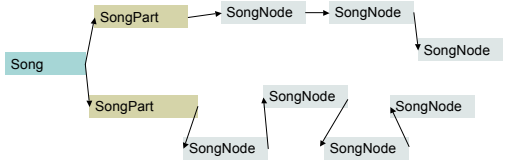
Then do an insert

```
  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…

Example Song

```
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();
  }
}
```