# 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.*

## Creating Notes

```
Welcome to DrJava.
> 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

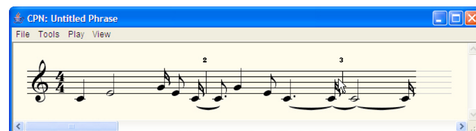
## Creating Phrases

```
> 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.

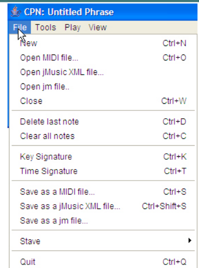
## Viewing Phrases

```
> 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)
```

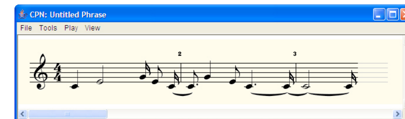
## A Phrase that starts later

```
> 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.

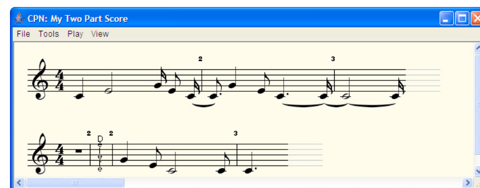



## Building Parts and Scores

```
> 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);
```



## Amazing Grace

```
> AmazingGraceSong song1 =
new AmazingGraceSong();
> song1.fillMeUp();
> song1.showMe();
```

```
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.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);
  };

}
```

## Imports and some *private* data

```
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.

```
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.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);

};
```

## Showing the Score

```
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)

Part: Instrument &

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

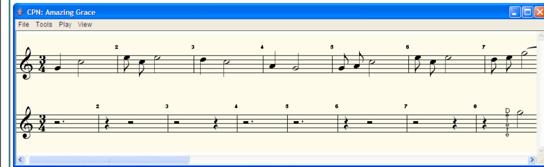
It **KNOWS:** it's Part and what comes *next*

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.

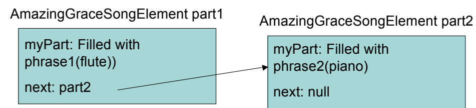
## What that would look like to use it

```
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()



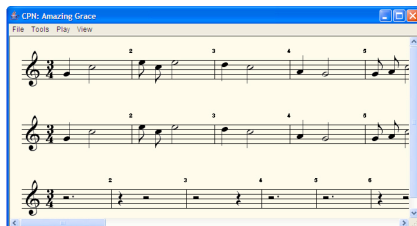
## What's going on here?



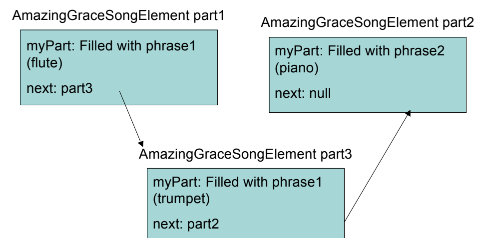

## Adding a third part

```
> 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

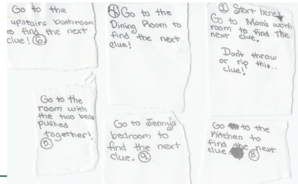


## What's going on here?



## 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

```
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 <u>need</u> 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.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};

    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.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(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 a method
 // (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();
 }
```

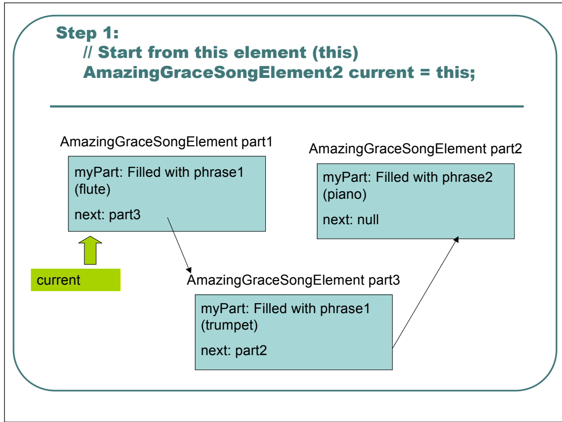
## showFromMeOn()

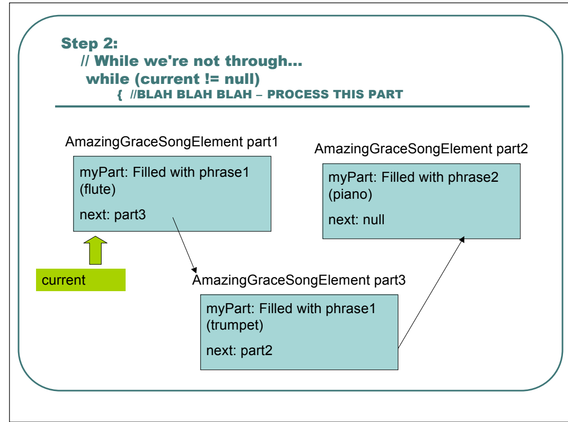
This is called *traversing* the linked list.

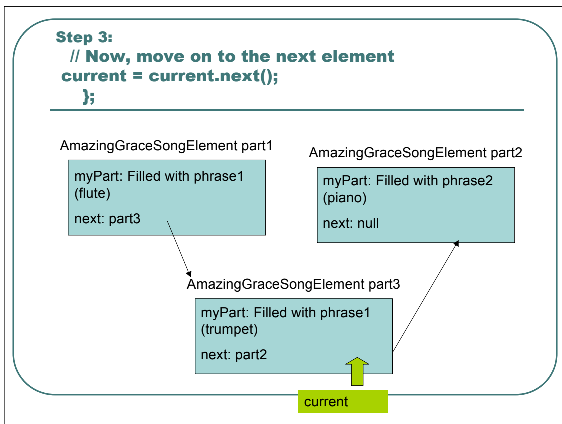
```
public void showFromMeOn(){
  // Make the score that we'll assemble the elements into
  // We'll set it up with the time signature and tempo we like
   Score myScore = new Score("Amazing Grace");
   myScore.setTimeSignature(3,4);
   myScore.setTempo(120.0);

   // 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);

 }
```

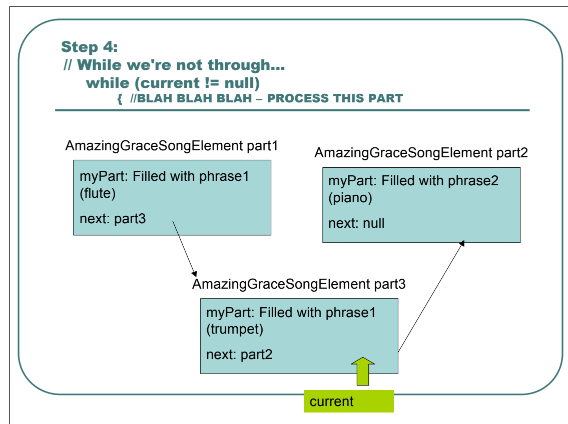
## The Key Part

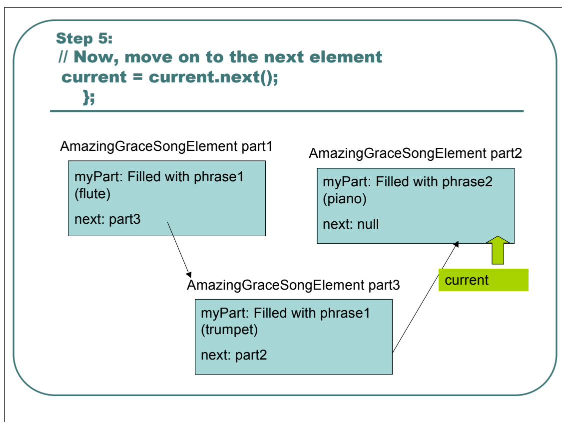
```
   // Start from this element (this)
   AmazingGraceSongElement2 current = this;
   // While we're not through...
   while (current != null)
   {
     // Set the channel, increment the channel, then add it in.
//BLAH BLAH BLAH (Ignore this part for now)

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

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

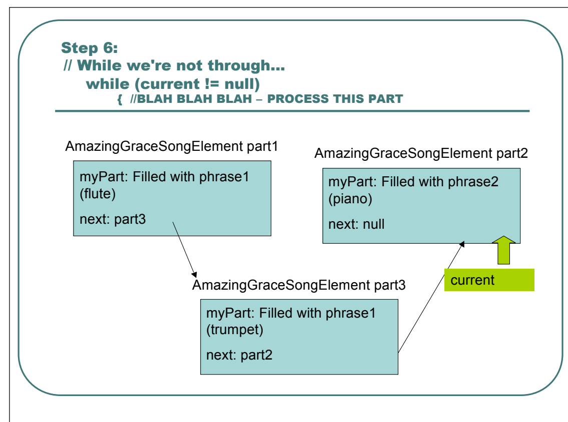

Step 1:
// Start from this element (this)
AmazingGraceSongElement2 current = this;
AmazingGraceSongElement part1
myPart: Filled with phrase1 (flute)
next: part3
AmazingGraceSongElement part2
myPart: Filled with phrase2 (piano)
next: null
current
AmazingGraceSongElement part3
myPart: Filled with phrase1 (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
AmazingGraceSongElement part2
myPart: Filled with phrase2 (piano)
next: null
current
AmazingGraceSongElement part3
myPart: Filled with phrase1 (trumpet)
next: part2

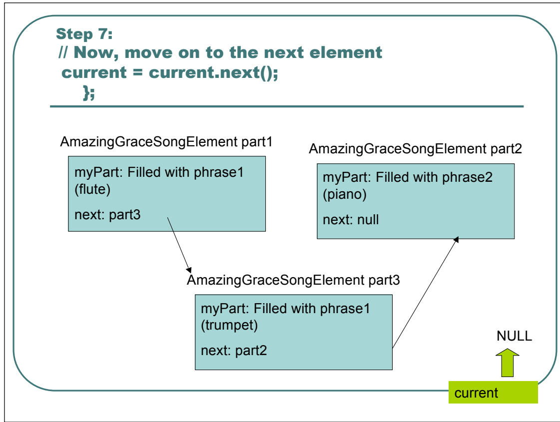

Step 3:
// Now, move on to the next element
current = current.next();
};
AmazingGraceSongElement part1
myPart: Filled with phrase1 (flute)
next: part3
AmazingGraceSongElement part2
myPart: Filled with phrase2 (piano)
next: null
AmazingGraceSongElement part3
myPart: Filled with phrase1 (trumpet)
next: part2
current


Step 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();
};
AmazingGraceSongElement part1
myPart: Filled with phrase1 (flute)
next: part3
AmazingGraceSongElement part2
myPart: Filled with phrase2 (piano)
next: null
current
AmazingGraceSongElement part3
myPart: Filled with phrase1 (trumpet)
next: part2

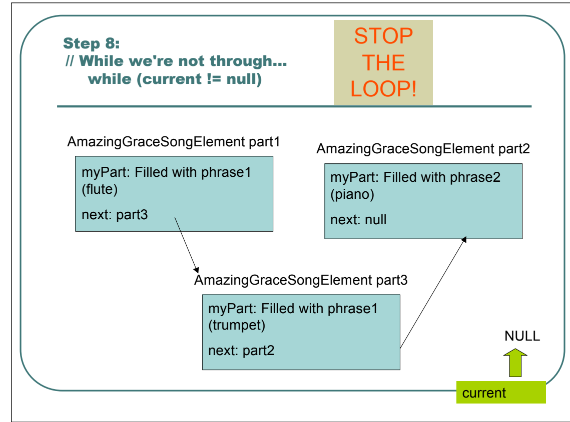

Step 6:
// While we're not through...
while (current != null)
{ //BLAH BLAH BLAH – PROCESS THIS PART
AmazingGraceSongElement part1
myPart: Filled with phrase1 (flute)
next: part3
AmazingGraceSongElement part2
myPart: Filled with phrase2 (piano)
next: null
current
AmazingGraceSongElement part3
myPart: Filled with phrase1 (trumpet)
next: part2

**Step 7:**
**// Now, move on to the next element**
**current = current.next();**
**};**




**Step 8:**
**// While we're not through…**
**while (current != null)**

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(AmazingGraceSong
    Element2 nextOne){
   this.next = nextOne;
 }

 public
    AmazingGraceSongElement
    2 next(){
   return this.next;
 }
```

```
> part1.setNext(part3);
> part3.setNext(part2);
> part1.showFromMeOn();
```

## Inserting into arrays

```
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/MediaSourc
    es/thisisatest.wav");
> test.insertAfter(test2,
    40000)
> test.play()
```

## More on Arrays vs. Lists

- Arrays
  - Much easier to traverse
  - Very fast to access a specific ($n^{th}$) 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?