Media Computation Workshop Day 3

Mark Guzdial
College of Computing
Georgia Institute of Technology
guzdial@cc.gatech.edu
http://www.cc.gatech.edu/~mark.guzdial
http://www.georgiacomputes.org

Workshop Plan-Day 3

- 9-10:00 am: Introducing objects in a MediaComp way
  - Turtles and MIDI.
  - 10:00-10:15: Break
- 10:15-11:00: Linked lists of MIDI.
- 11:00-12:00: Linked lists and trees of pictures
- 12:00-1:00: Lunch
- 1:00-2:30: Tackling a homework assignment in Media Computation. *Creating linked list music* or *Making a movie with sound*.
  - 2:30-2:45: Break
- 2:45-3:30: Simulations, continuous and discrete
- 3:30-4:30: Creating the wildebeests and villagers: Making movies from simulations .

Creating classes with Turtles

- Overriding our basic object Turtle to make a Confused (or Drunken) Turtl

Creating an Inherited Class

- Create a class ConfusedTurtle that inherits from the Turtle class
  - But when a ConfusedTurtle object is asked to turn left, it should turn right
  - And when a ConfusedTurtle object is asked to turn right, it should turn left

Inheriting from a Class

- To inherit from another class
  - Add extends ClassName to the class declaration

```
public class ConfusedTurtle extends Turtle
{
}
```

- Save in ConfusedTurtle.java
- Try to compile it

Compile Error?

- If you try to compile ConfusedTurtle you will get a compiler error
  - Error: cannot resolve symbol
  - symbol: constructor Turtle()
  - location: class Turtle
- Why do you get this error?

Inherited Constructors

- When one class inherits from another all constructors in the child class will have an implicit call to the no-argument parent constructor as the first line of code in the child constructor
  - Unless an explicit call to a parent constructor is there as the first line of code
    - Using super(paramList);

Why is an Implicit Call to Super Added?

- Fields are inherited from a parent class
  - But fields should be declared private
    - Not public, protected, or package visibility
      - Lose control over field at the class level then
  - But then subclasses can’t access fields directly
  - How do you initialize inherited fields?
    - By calling the parent constructor that initializes them
      - Using super(paramList);

Explanation of the Compile Error

- There are no constructors in ConfusedTurtle
  - So a no-argument one is added for you
    - With a call to super();
  - But, the Turtle class doesn't have a no-argument constructor
    - All constructors take a world to put the turtle in
- So we need to add a constructor to ConfusedTurtle
  - That takes a world to add the turtle to
    - And call super(theWorld);

Add a Constructor that takes a World

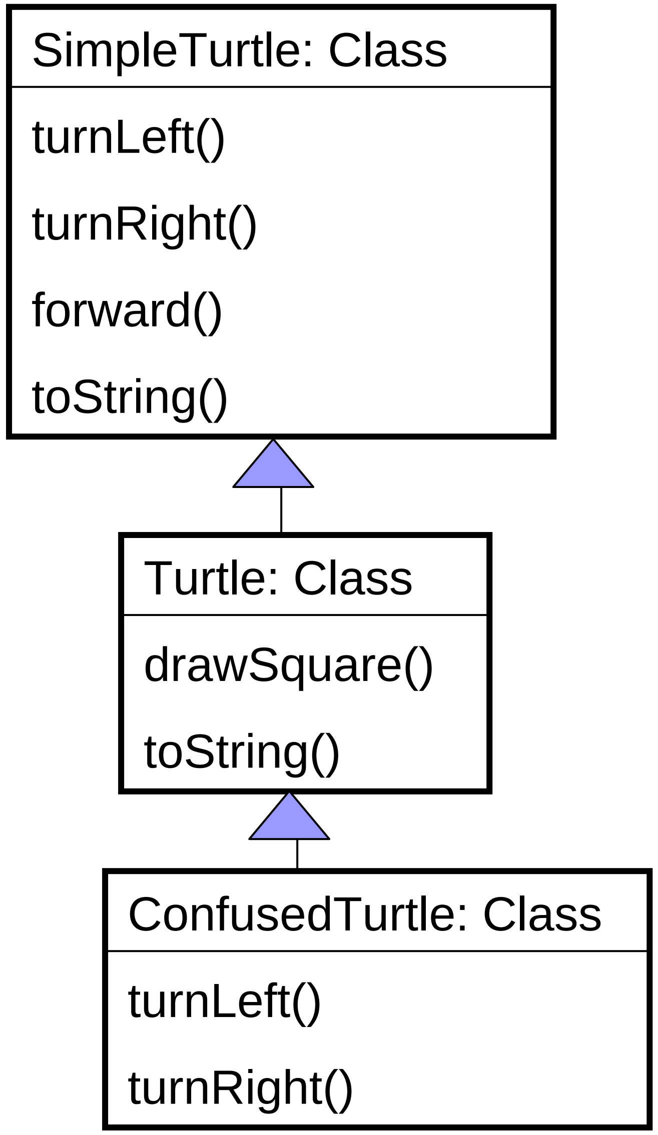
```
public class ConfusedTurtle extends Turtle
{
  /**
   * Constructor that takes a world and
   * calls the parent constructor
   * @param theWorld the world to put the
   * confused turtle in
   */
  public ConfusedTurtle(World theWorld)
  {
    super (theWorld);
  }

}
```

Try this Out

- Compile ConfusedTurtle
  - It should compile
- Try it out
  - It should ask just like a Turtle object
- How do we get it to turn left when asked to turn right?
  - And right when asked to turn left?
    - Use super.turnLeft() and super.turnRight()
      - super is a keyword that means the parent class

Resolving Methods

- When a method is invoked (called) on an object
  - The class that created the object is checked
    - To see if it has the method defined in it
      - If so it is executed
      - Else the parent of the class that created the object is checked for the method
      - And so on until the method is found
- Super means start checking with the parent of the class that created the object



Polymorphism

- Means many forms
- Allows for processing of an object based on the object’s type
- A method can be declared in a parent class
  - And redefined (overriden) by the subclasses
- Dynamic or run-time binding will make sure the correct method gets run
  - Based on the type of object it was called on at run time

Confused Turtle turnLeft and turnRight

```
/**
  * Method to turn left (but confused turtles
  * turn right)
  */
 public void turnLeft()
 {
   super.turnRight();
 }

 /**
  * Method to turn right (but confused turtles
  * turn left)
  */
 public void turnRight()
 {
   super.turnLeft();
 }
```

Try out ConfusedTurtle

```
> World earth = new World();
> Turtle tommy = new Turtle(earth);
> tommy.forward();
> tommy.turnLeft();
> ConfusedTurtle bruce = new ConfusedTurtle(earth);
> bruce.backward();
> bruce.turnLeft();
> bruce.forward();
> tommy.forward();
> tommy.turnRight();
> bruce.turnRight();
```

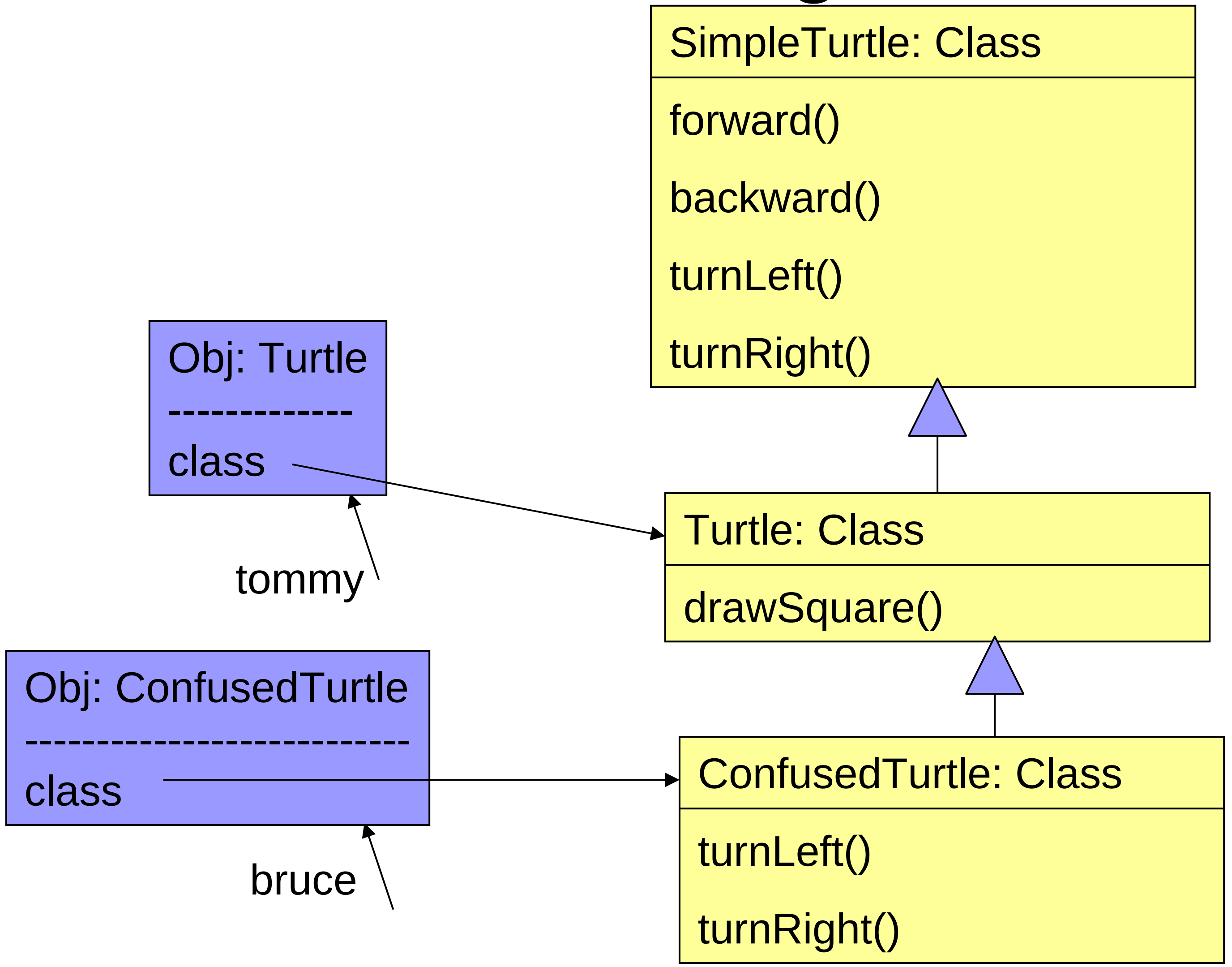
Override Methods

- Children classes inherit parent methods
  - The confused turtle knows how to go forward and backward
    - Because it inherits these from Turtle
- Children can override parent methods
  - Have a method with the same name and parameter list as a parent method
    - This method will be called instead of the parent method
      - Like turnLeft and turnRight

What is Happening?

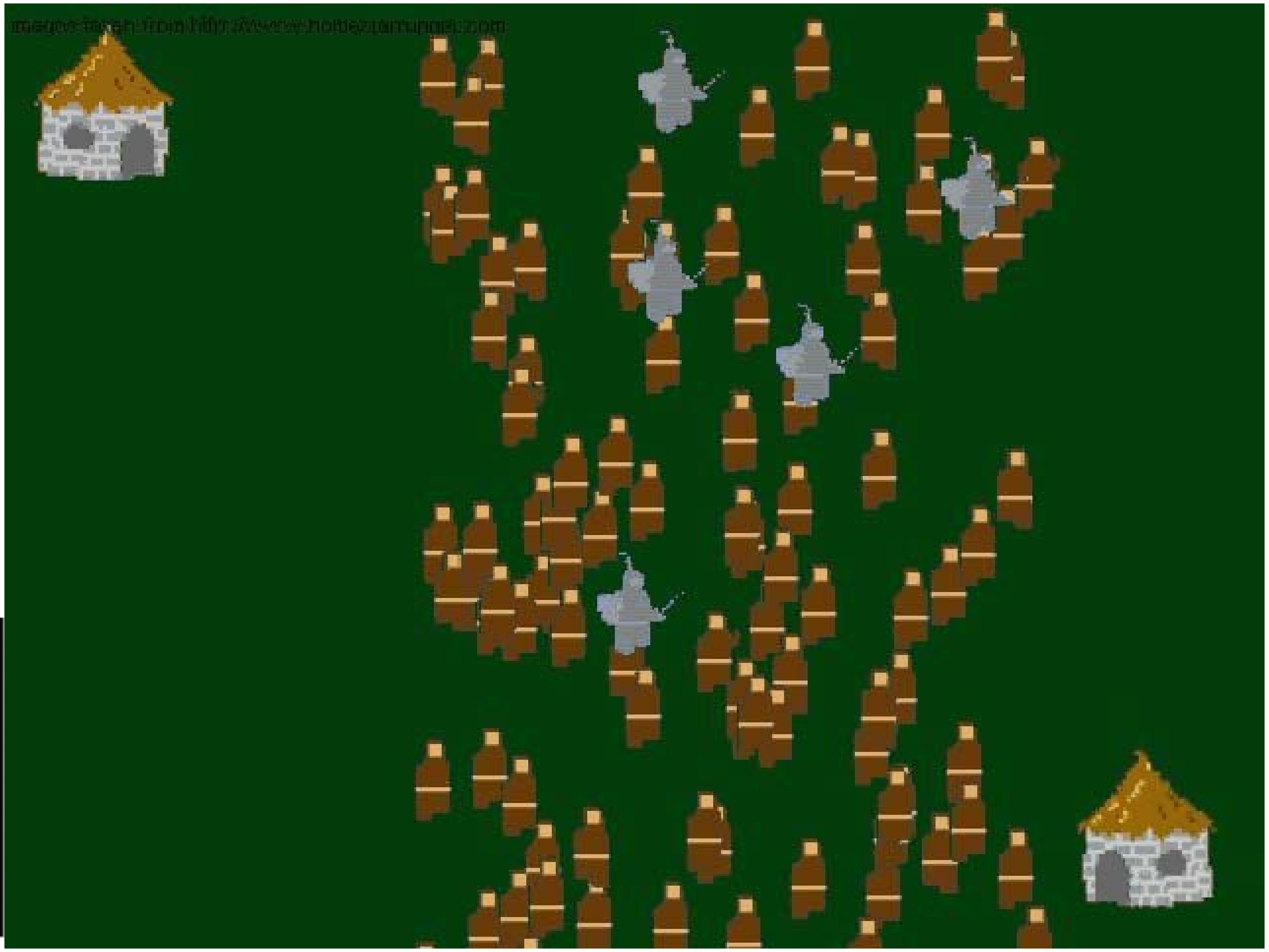
- Each time an object is asked to execute a method
  - It first checks the class that created the object to see if the method is defined in that class
    - If it is it will execute that method
    - If it isn’t, it will next check the parent class of the class that created it
      - And execute that method if one if found
      - If no method with that name and parameter list is found it will check that classes parent
        - And keep going till it finds the method

Method Overloading

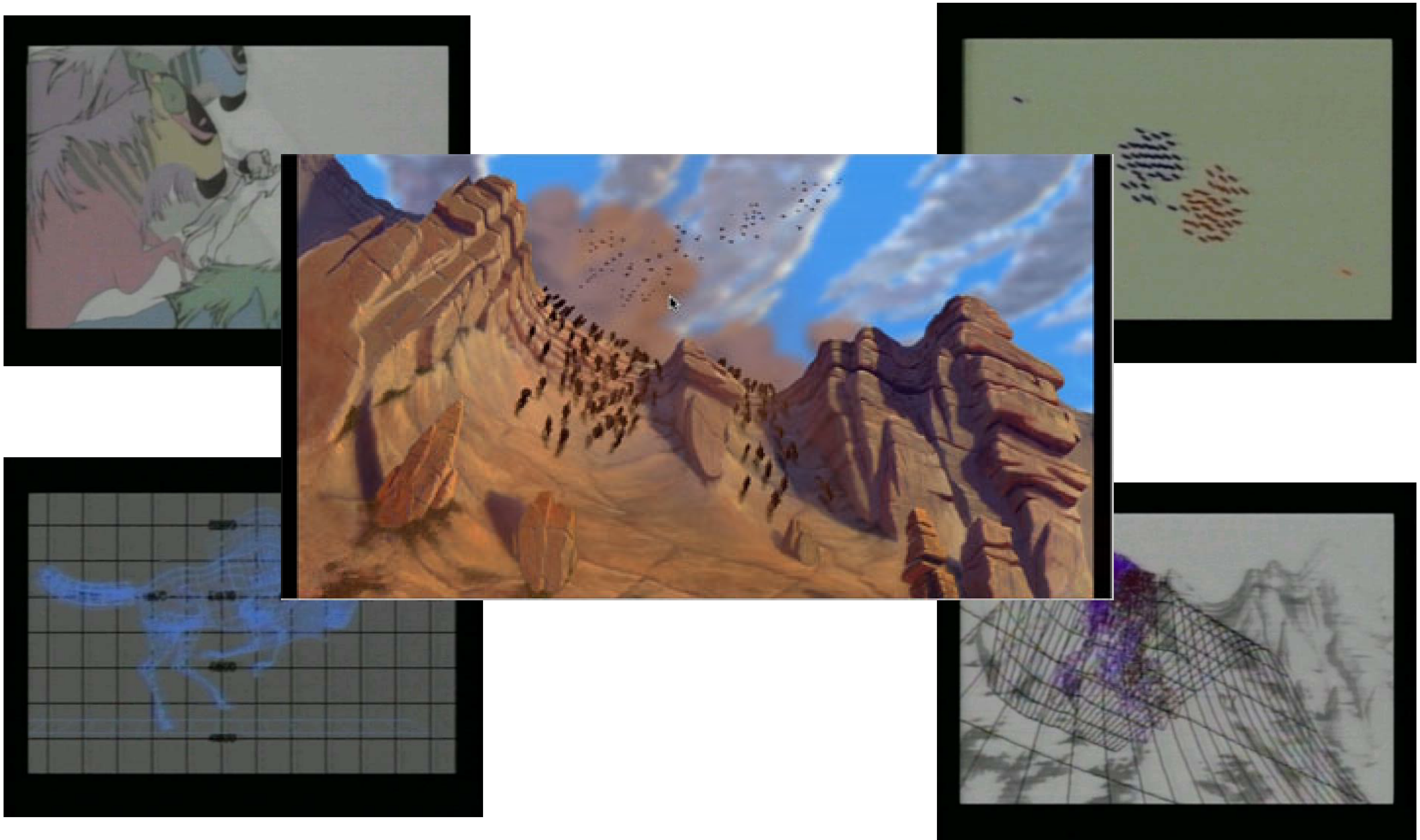


Multimedia CS2 in Java

- Driving question: “How did the wildebeests stampede in *The Lion King?*”
- Spring 2005: 31 students, 75% female, 91% success rate.



Connecting to the Wildebeests
It's all about data structures



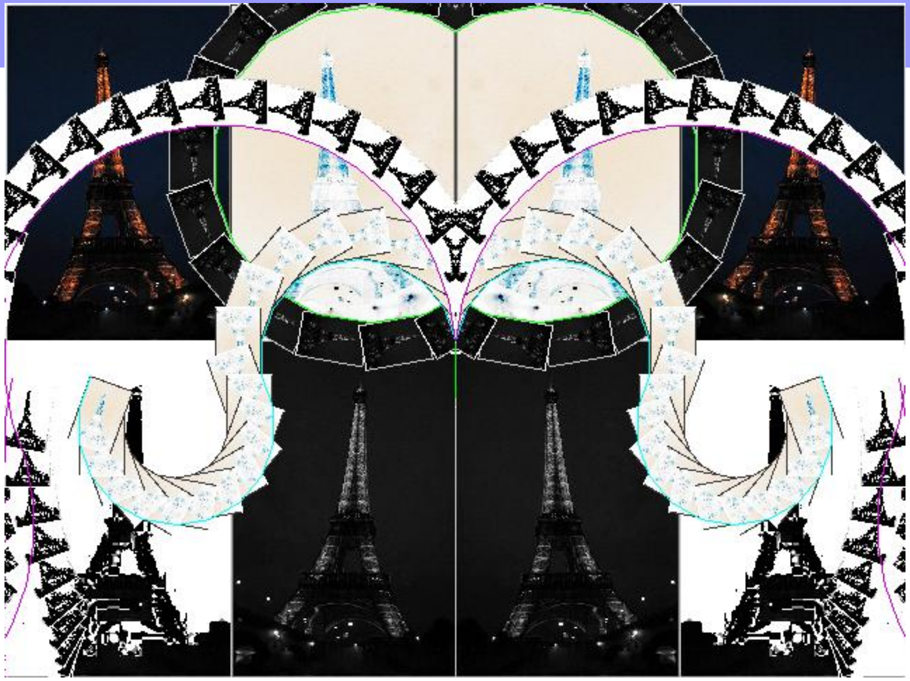
Syllabus

- Introduction to Java and Media Computation
  - Manipulating turtles, images, MIDI, sampled sounds.
  - Insertion and deletion (with shifting) of sampled sounds (arrays).
- Structuring Music
  - Goal: A structure for flexible music composition
  - Put MIDI phrases into linked list nodes.
    - Use Weave and Repeat to create repeating motifs as found in Western Music
    - At very end, create a two-branched list to start on trees.

Swan **Canon**

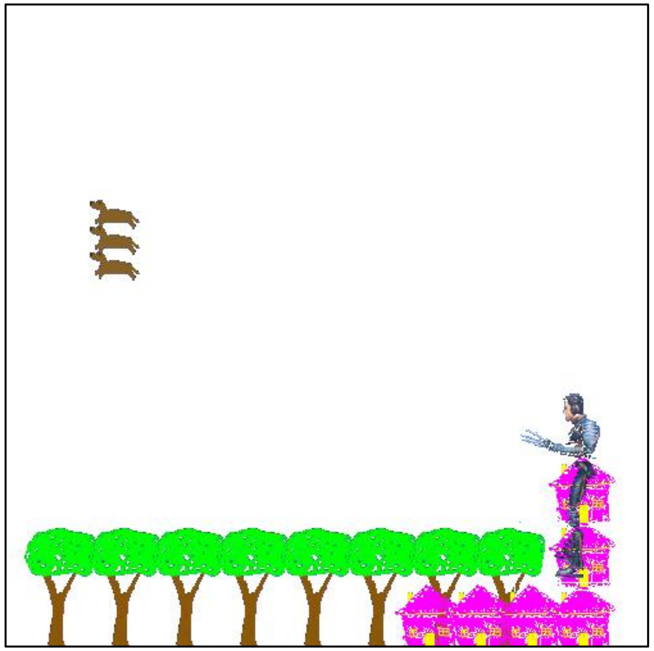
Bells **Fur Elise**

HW2: Create a collage, but must use turtles



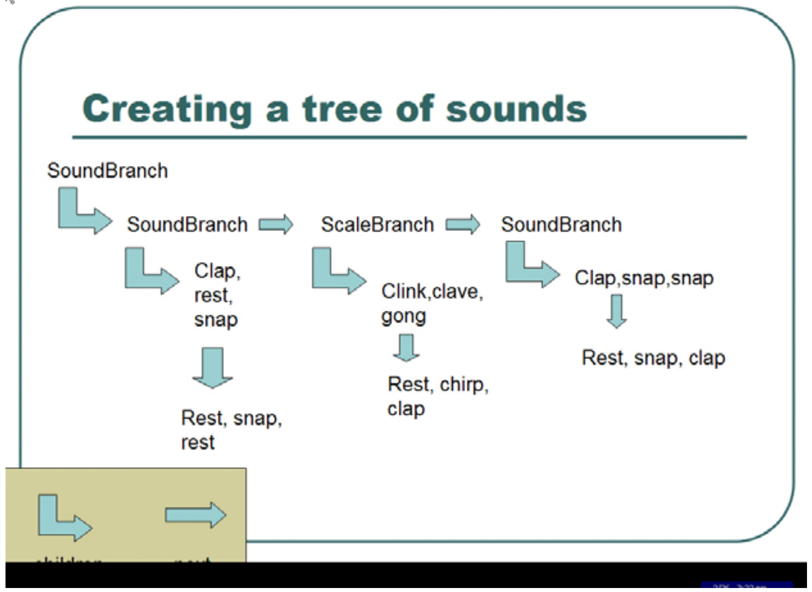
Syllabus (Continued)

- Structuring Images
  - Using linearity in linked list to represent ordering (e.g., left to right)
  - Using linearity in linked list to represent layering (as in PowerPoint)
  - Mixing positioned and layered in one structure, using abstract super classes.
  - Structuring a scene in terms of branches—introducing a *scene graph* (first tree)
- (We’ll see these slides as an example later.)



Syllabus (Cont’d)

- Structuring Sound
  - Collecting sampled sounds into linked lists and trees, as with images.
    - But all traversals are recursive.
  - Use different traversals of same tree to generate different sounds.
  - Replace a sound in-place



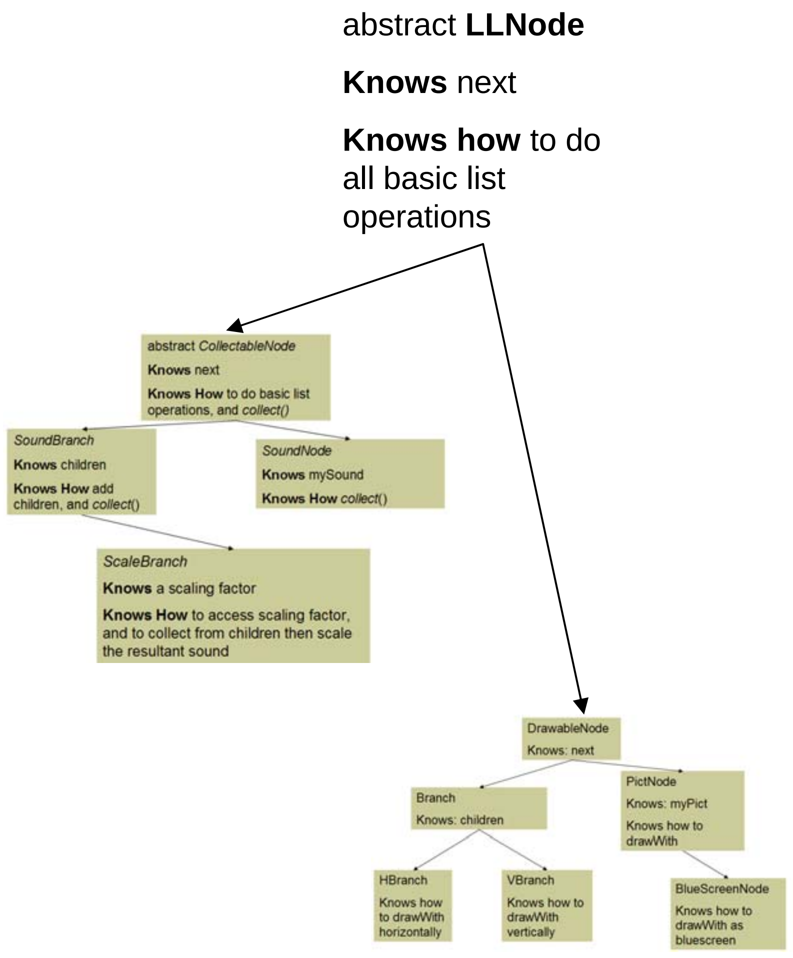
Original

Scale the children

Scale the next

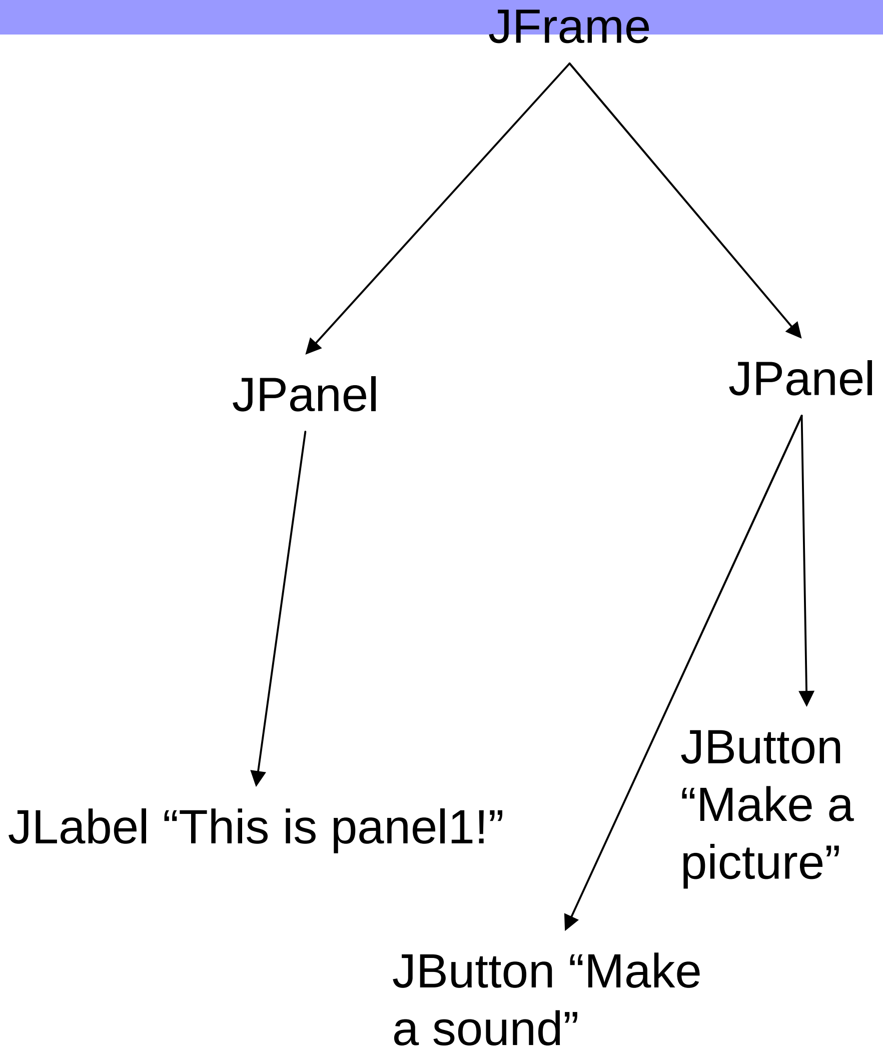
Syllabus (cont'd)

- Generalizing lists and trees
  - Create an abstract class "Linked List Node" (LLNode) on top of the sound and image class hierarchies
- Make all image and sound examples work the same


abstract **LLNode**
Knows next
Knows how to do all basic list operations
abstract *CollectableNode*
Knows next
Knows How to do basic list operations, and *collect()*
SoundBranch
Knows children
Knows How add children, and *collect()*
SoundNode
Knows mySound
Knows How *collect()*
ScaleBranch
Knows a scaling factor
Knows How to access scaling factor, and to collect from children then scale the resultant sound
DrawableNode
Knows: next
Branch
Knows: children
PictNode
Knows: myPict
Knows how to drawWith
HBranch
Knows how to drawWith horizontally
VBranch
Knows how to drawWith vertically
BlueScreenNode
Knows how to drawWith as bluescreen

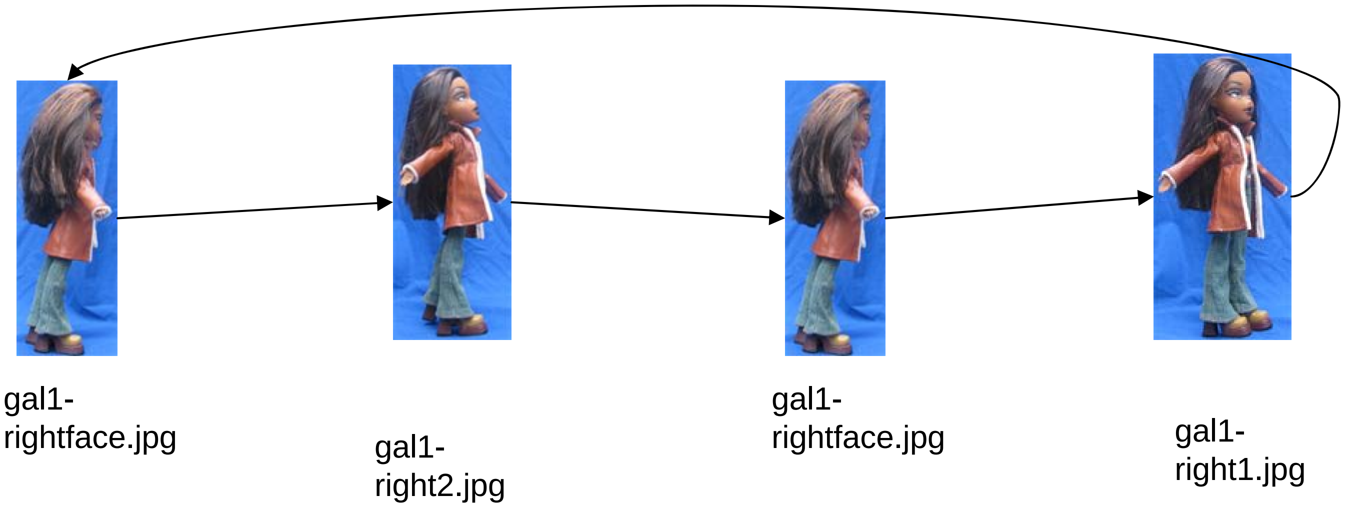
Syllabus (Cont'd)

- GUIs as trees
  - We introduce construction of a Swing frame as construction of a tree.
  - Different layout managers are then different renderers of the same tree.
- Traditional (binary) trees as a specialization
  - Can we make trees that are faster to search?



Syllabus (cont'd)

- Lists that Loop
  - Introduce circular linked lists as a way of create Mario-Brothers' style cel animations.
  - Introduce trees that loop as a way of introducing graphs.

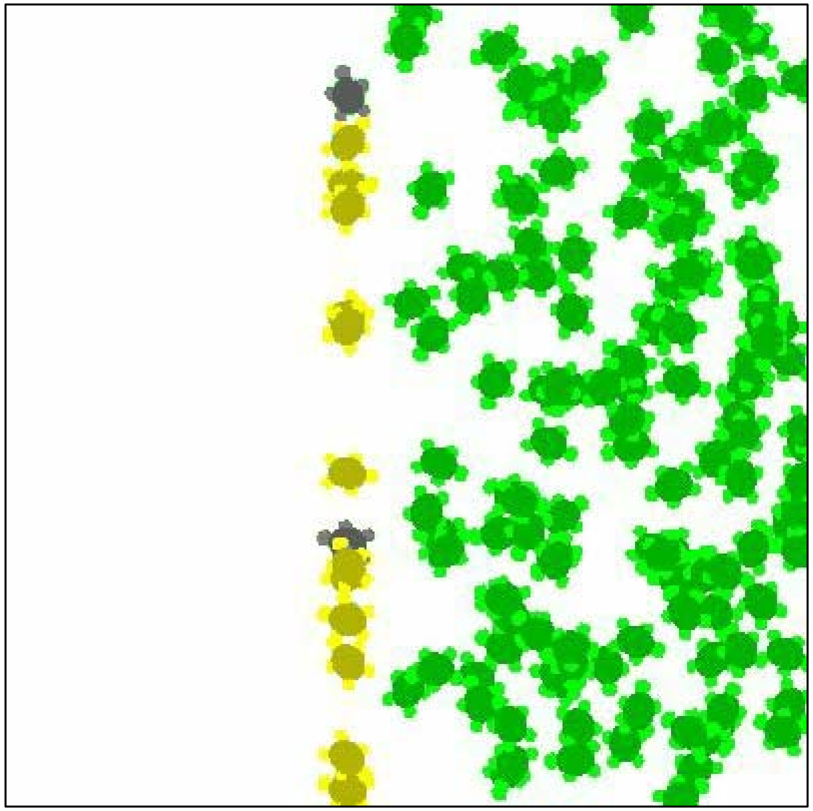


Syllabus (cont'd)

- Introducing Simulations
  - Introduce continuous and discrete event simulations, and Normal and uniform probability distributions
  - We do wolves and deer,
    disease propagation,
    political influence.
  - Create a set of classes for simulation, then re-write our simulations for those classes.
  - Writing results to a file for later analysis
- Finally, Making the Wildebeests and Villagers
  - Mapping from positions of our turtles to an animation frame.
  - Creating an animation from a simulation.

HW7: Simulate European emigration to America

- Students are required to try several different scenarios, aiming for historical accuracy.
- Counts of Europeans, Americans, and in-transit per year are written to a file for graphing in Excel



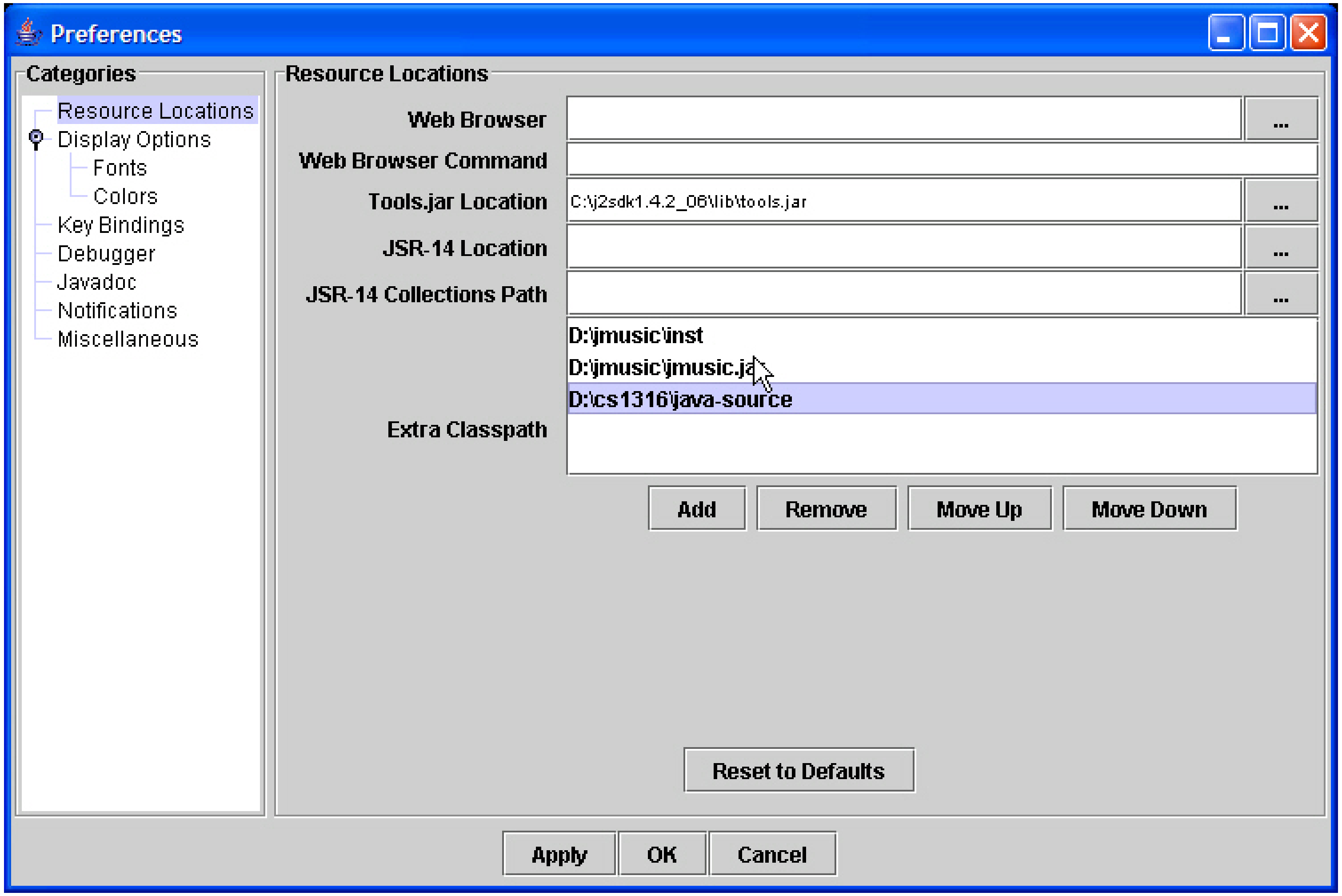
Syllabus (cont'd)

- Introduction to Discrete Event Simulations
  - Create a discrete event simulation of trucks, factories, salespeople, and markets.
  - Use turtles to create an animated display.
  - Now, the real focus is the simulation, and the animation is just a mapping from the simulation.
    - Animation becomes yet another medium in which we can review results, like data in an Excel spreadsheet, music, or sound.

A selection of data structures

- First, structuring music:
  - Notes, Phrases, Parts, and Scores as a nice example of objects and modeling.
- Then linked lists of images
  - Is it about layering (as in Photoshop) or positioning (as in *Lord of the Rings*)?
- Then trees of images
  - A Scene Graph

Telling DrJava where to find files

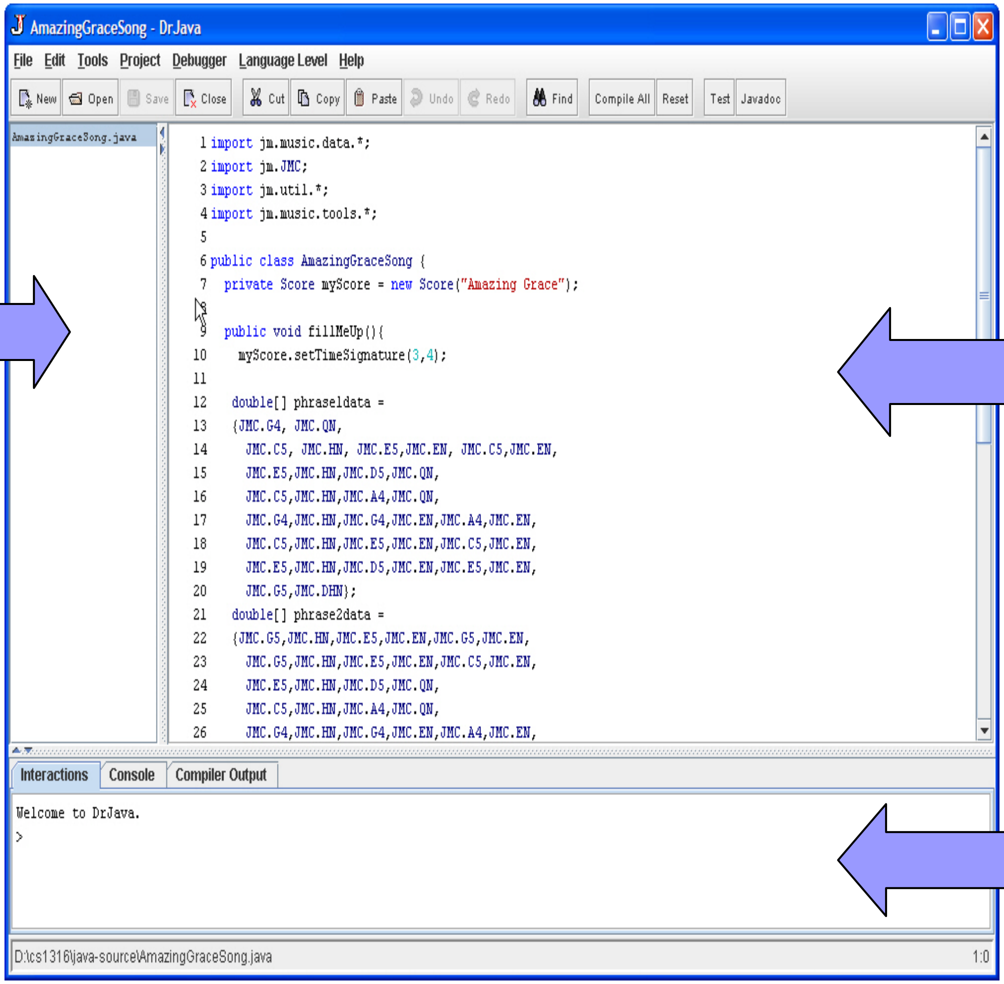


Resource Locations

Web Browser	
Web Browser Command	
Tools.jar Location	C:\j2sdk1.4.2_06\lib\tools.jar
JSR-14 Location	
JSR-14 Collections Path	
Extra Classpath	D:\jmusic\inst D:\jmusic\jmusic.jar D:\cs1316\java-source

Parts of DrJava

List of class files that you have open


AmazingGraceSong - DrJava
File Edit Tools Project Debugger Language Level Help
New Open Save Close Cut Copy Paste Undo Redo Find Compile All Reset Test Javadoc
AmazingGraceSong.java
Interactions Console Compiler Output

```
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[] phraseldata =
   {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,
```

```
Welcome to DrJava.
>
```

D:\cs1316\java-source\AmazingGraceSong.java 1:0

Text of your class file (.java)

Where you interact with Java

An example with Music

```
> import jm.util.*;
> import jm.music.data.*;
> Note n1;
> n1 = new Note(60,0.5);
> // Create an eighth note at
  C octave 4
```

- JMusic pieces need to be *import*ed first to use them.

An example with Music

```
> import jm.util.*;
> import jm.music.data.*;
> Note n1;
> n1 = new Note(60,0.5);
> // Create an eighth note at
   C octave 4
```

- Declare a *Note* variable.

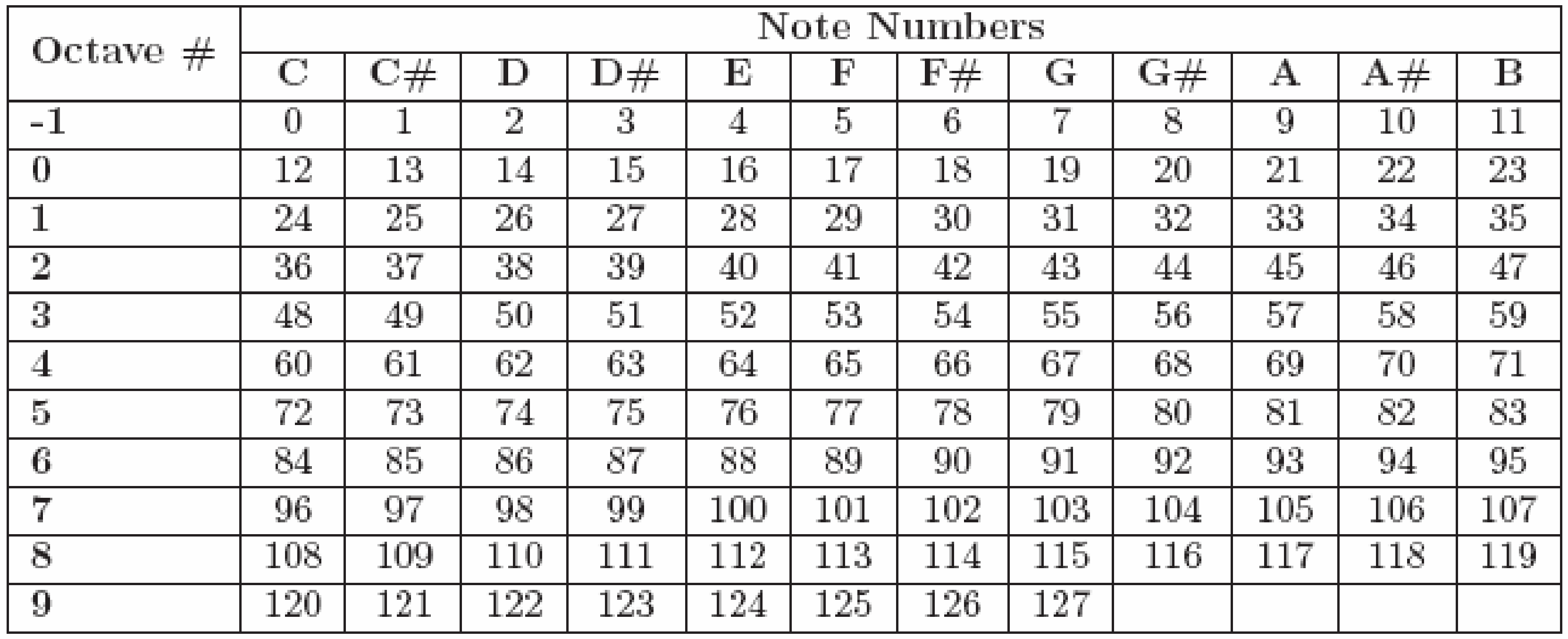
An example with Music

```
> import jm.util.*;
> import jm.music.data.*;
> Note n1;
> n1 = new Note(60,0.5);
> // Create an eighth note at
  C octave 4
```

- *Note* instances have nothing to do with filenames.
- To create a note, you need to know *which* note, and a *duration*

Starting a line with // creates a *comment*—ignored by Java

MIDI notes

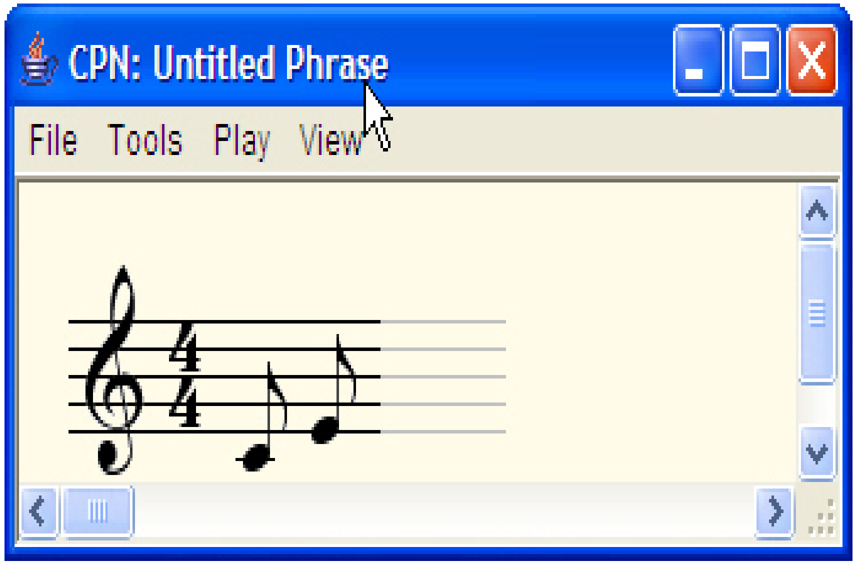


Octave #	Note Numbers											
	C	C#	D	D#	E	F	F#	G	G#	A	A#	B
-1	0	1	2	3	4	5	6	7	8	9	10	11
0	12	13	14	15	16	17	18	19	20	21	22	23
1	24	25	26	27	28	29	30	31	32	33	34	35
2	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59
4	60	61	62	63	64	65	66	67	68	69	70	71
5	72	73	74	75	76	77	78	79	80	81	82	83
6	84	85	86	87	88	89	90	91	92	93	94	95
7	96	97	98	99	100	101	102	103	104	105	106	107
8	108	109	110	111	112	113	114	115	116	117	118	119
9	120	121	122	123	124	125	126	127				

TABLE 2.1: MIDI notes

Making more notes

```
> Note n2=new Note(64,0.5);
> View.notate(n1);
Error: No 'notate' method in
    'jm.util.View' with arguments:
    (jm.music.data.Note)
> Phrase phr = new Phrase();
> phr.addNote(n1);
> phr.addNote(n2);
> View.notate(phr);
-- Constructing MIDI file from'Untitled
    Score'... Playing with JavaSound
    ... Completed MIDI playback -------
    -
```



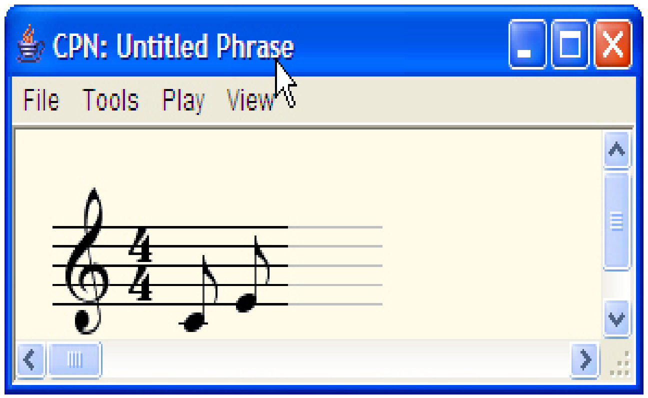
What’s going on here?

```
> Note n2=new Note(64,0.5);
> View.notate(n1);
Error: No 'notate' method in
'jm.util.View' with arguments:
(jm.music.data.Note)
```

- We’ll make another *Note* (at E4, another eighth note)
- There is an object named *View* that knows how to *notate* parts of music, but not an individual note.

What's going on here?

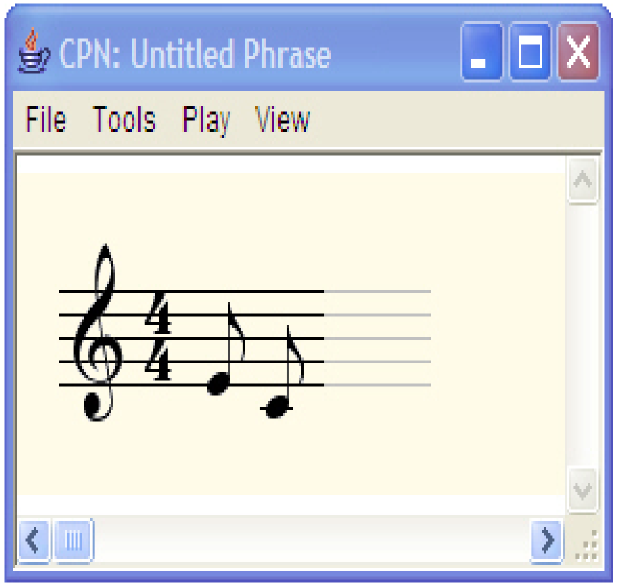
```
> Phrase phr = new Phrase();
> phr.addNote(n1);
> phr.addNote(n2);
> View.notate(phr);
-- Constructing MIDI file
from'Untitled Score'... Playing
with JavaSound ... Completed
MIDI playback --------
```



- We'll create a new *Phrase* instance and make a variable *phr* to refer to it. (*phr* has to be declared to be a *Phrase*.)
- *Phrase* instances know how to *addNote* notes to them. These are methods that take an argument—a *Note* instance.
- The *View* object <u>does</u> know how to *notate* an input *Phrase* instance. It generates this cool window where you see the notes and can play them (or save them as MIDI.)

Playing a different Phrase

```
> Phrase nuphr = new
    Phrase(0.0,JMC.FLUTE);
> nuphr.addNote(n2);
> nuphr.addNote(n1);
> View.notate(nuphr);
```



- We can specify when a phrase starts and with what instrument.
- We can add notes (even the same notes!) in different orders

Modeling Music

- The JMusic package is really *modeling* music.
    - Notes have tones and durations.
    - Musical Phrases are collections of notes.
    - We can play (and View) a musical phrase.
        - A phrase doesn’t have to start when other phrases do, and a phrase can have its own instrument.

Objects know things and can do things

	What instances of this class *know*	**What instances of this class *can do***
Note	A musical note and a duration	<Nothing we’ve seen yet>
Phrase	The notes in the phrase	addNote(aNote)

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(){
```

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.

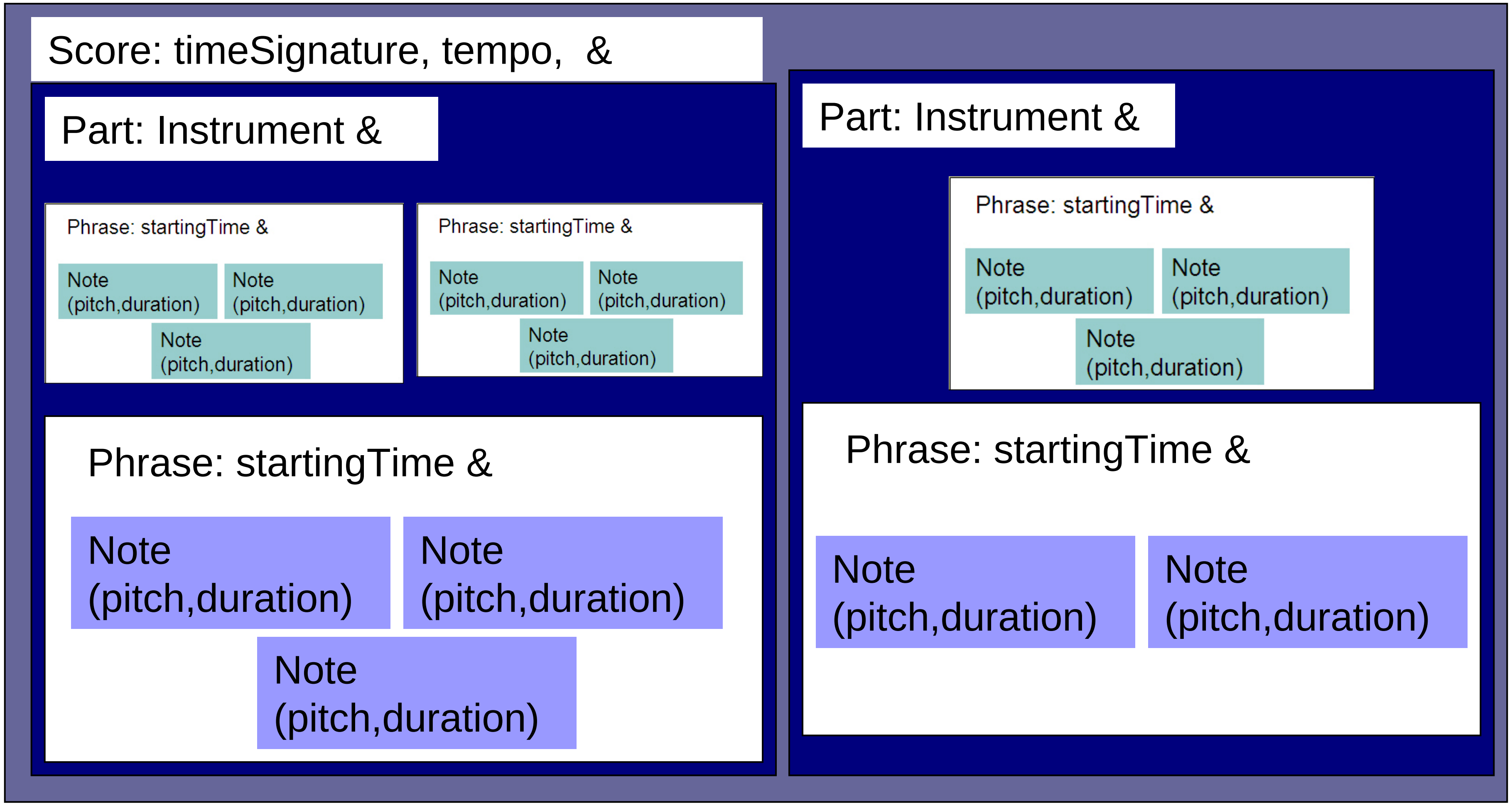
I broke it roughly into halves.

```
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



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.

Then comes notes in arrays,
linked lists of segments of AmazingGrace,
then real and flexible linked lists…

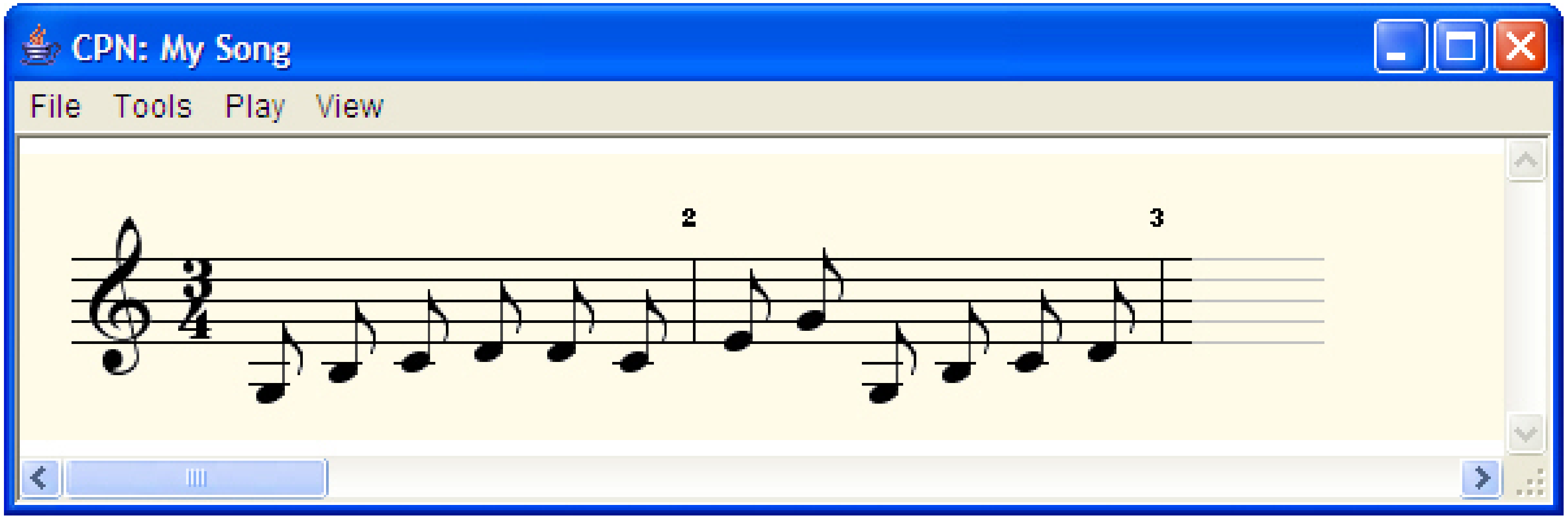
Version 3: SongNode and SongPhrase

- 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.

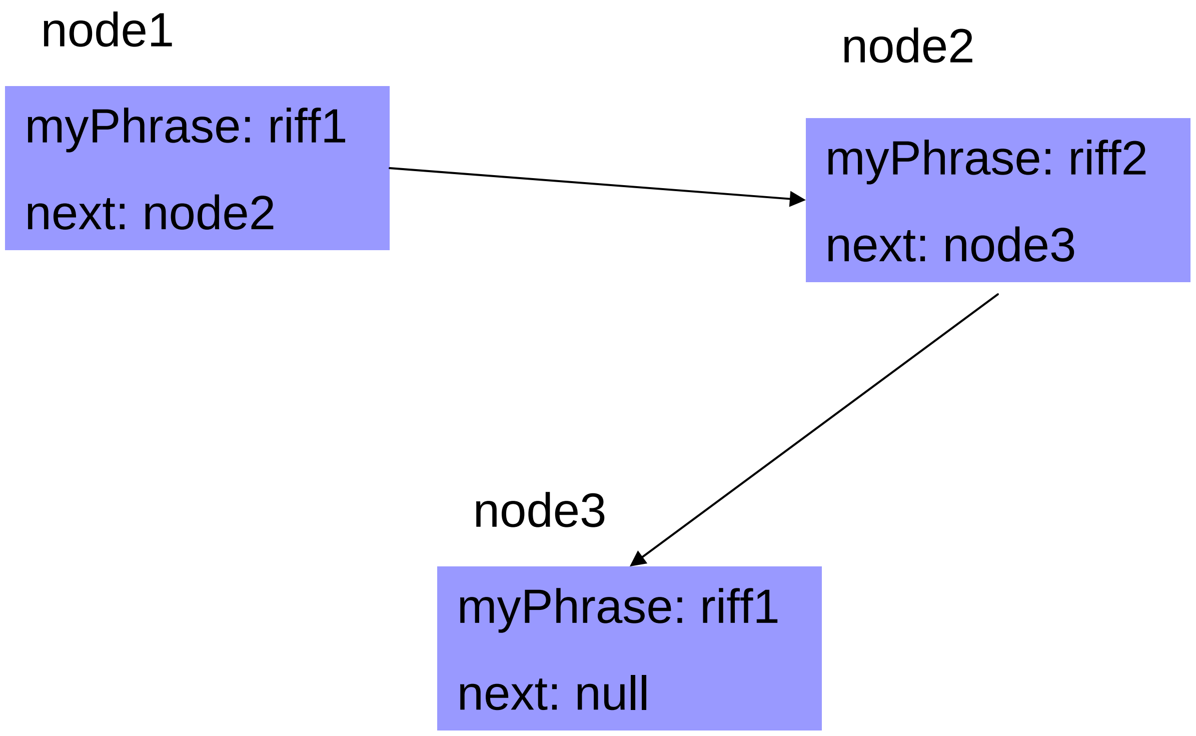
Using SongNode and SongPhrase

```
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,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;
}
```

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;
  }
```

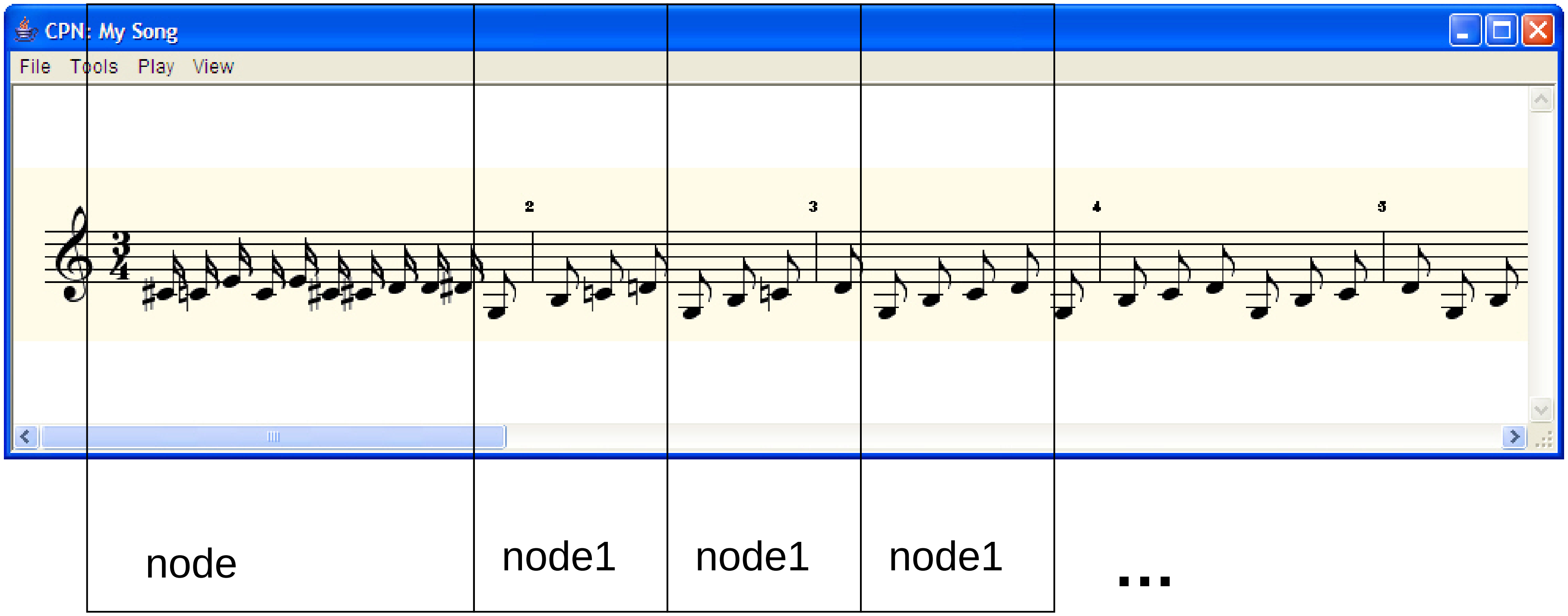
Going beyond connecting nodes

- 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



Repeating

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

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

Really useful: Do this with people as nodes!

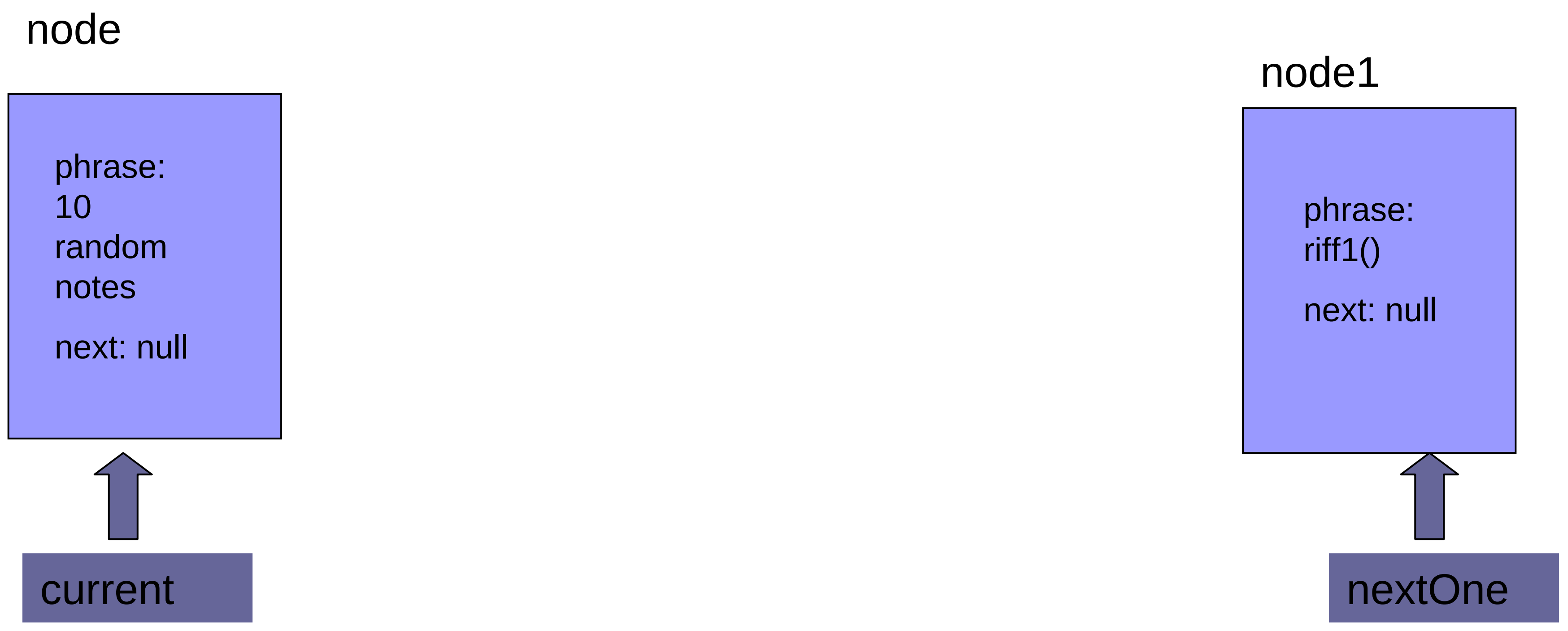
- We give people pieces of paper with “notes” on them.
- Nodes point to their “next”

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;
 }
```

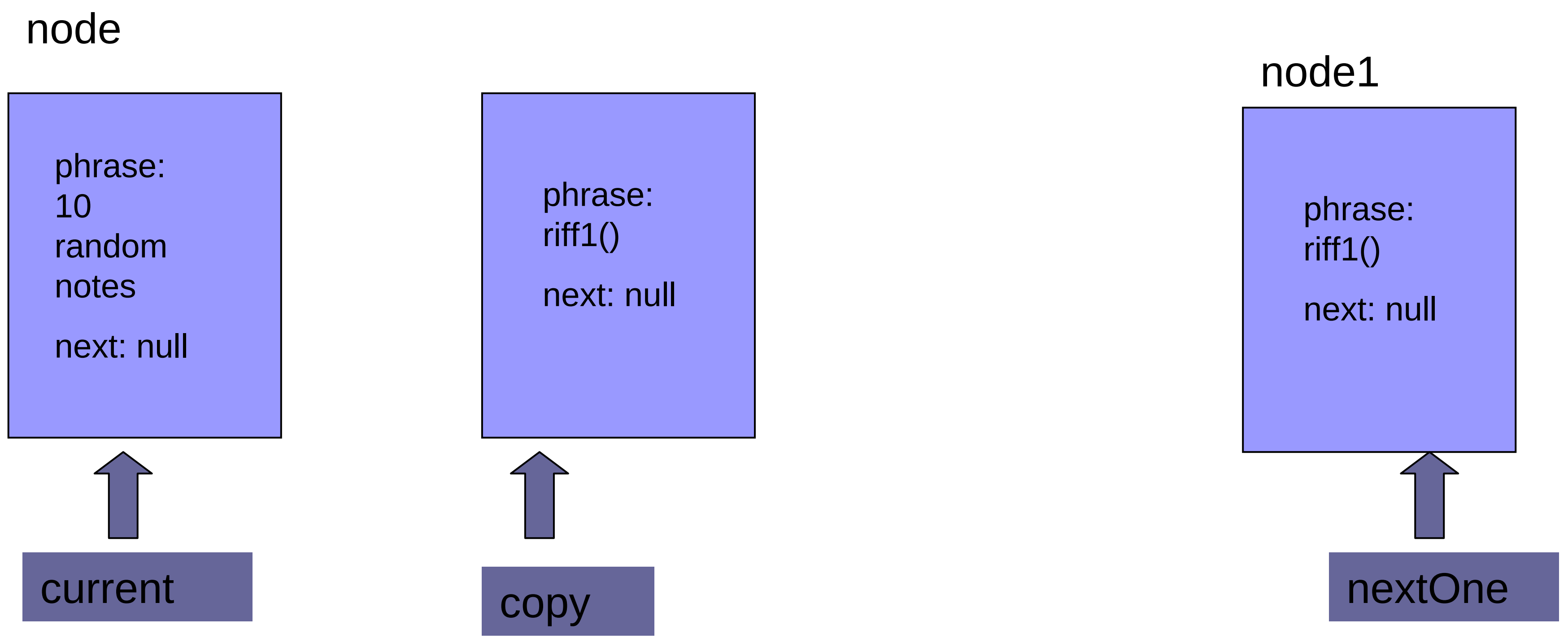
Step 1:

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



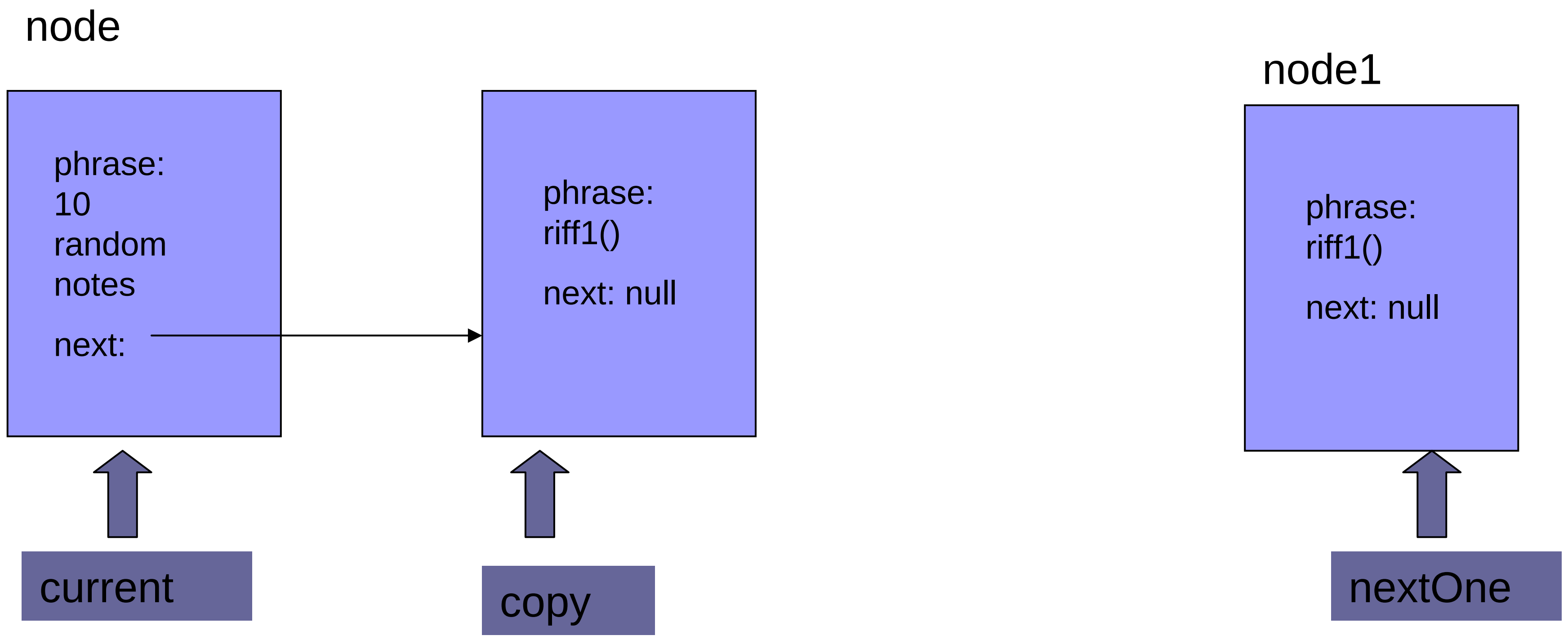
Step 2:

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



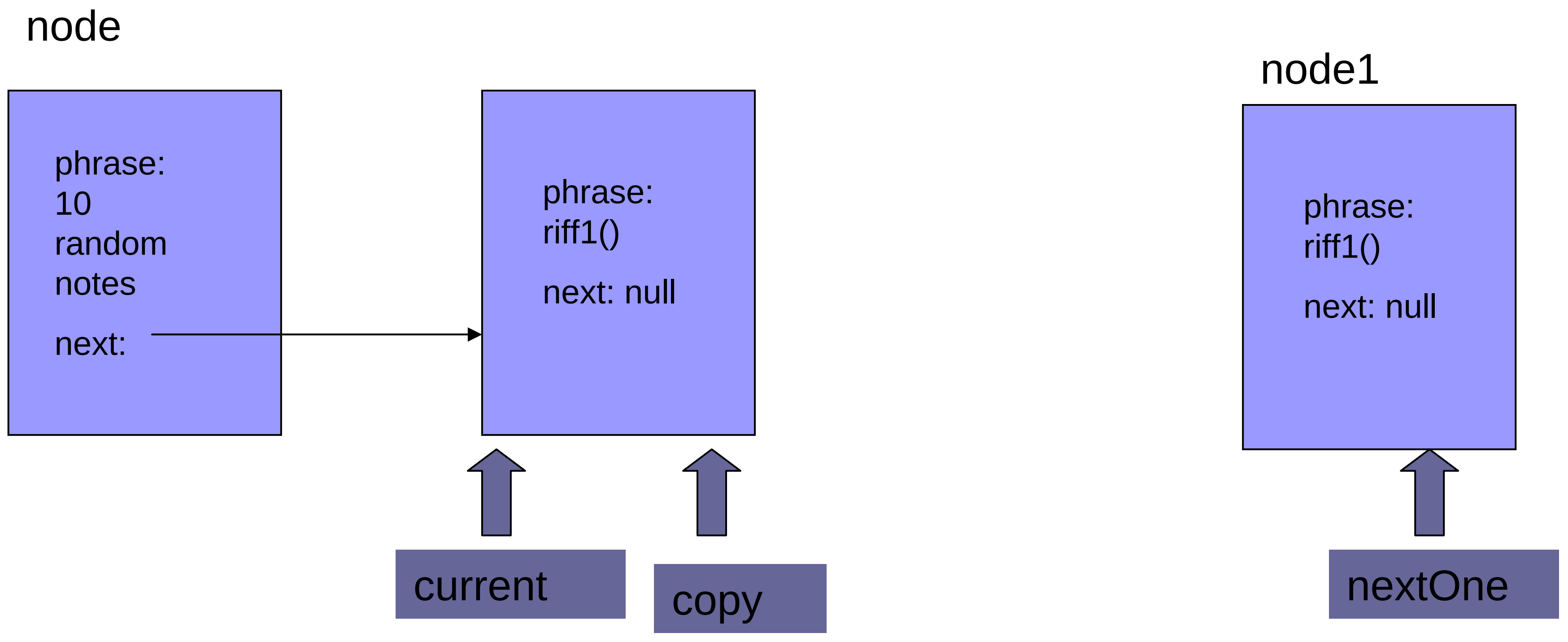
Step 3:

current.setNext(copy); // Set as next



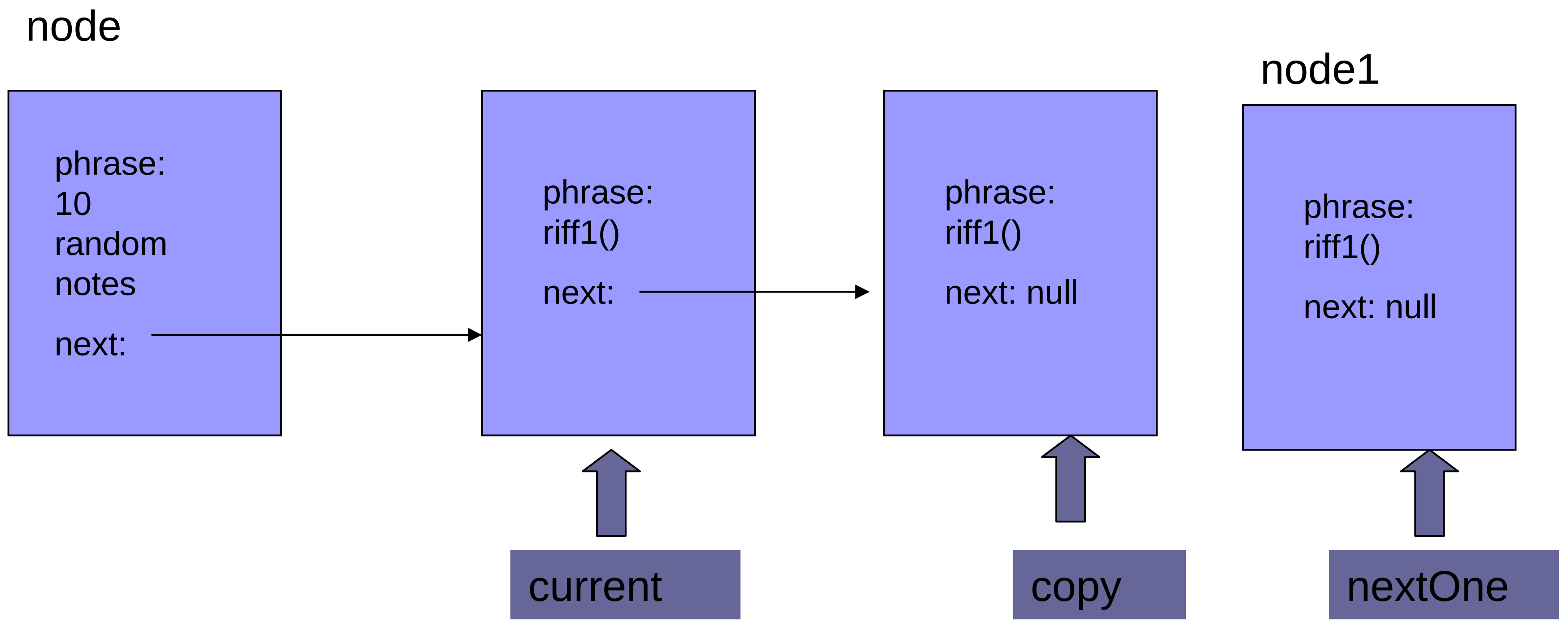
Step 4:

current = copy; // Now append to copy



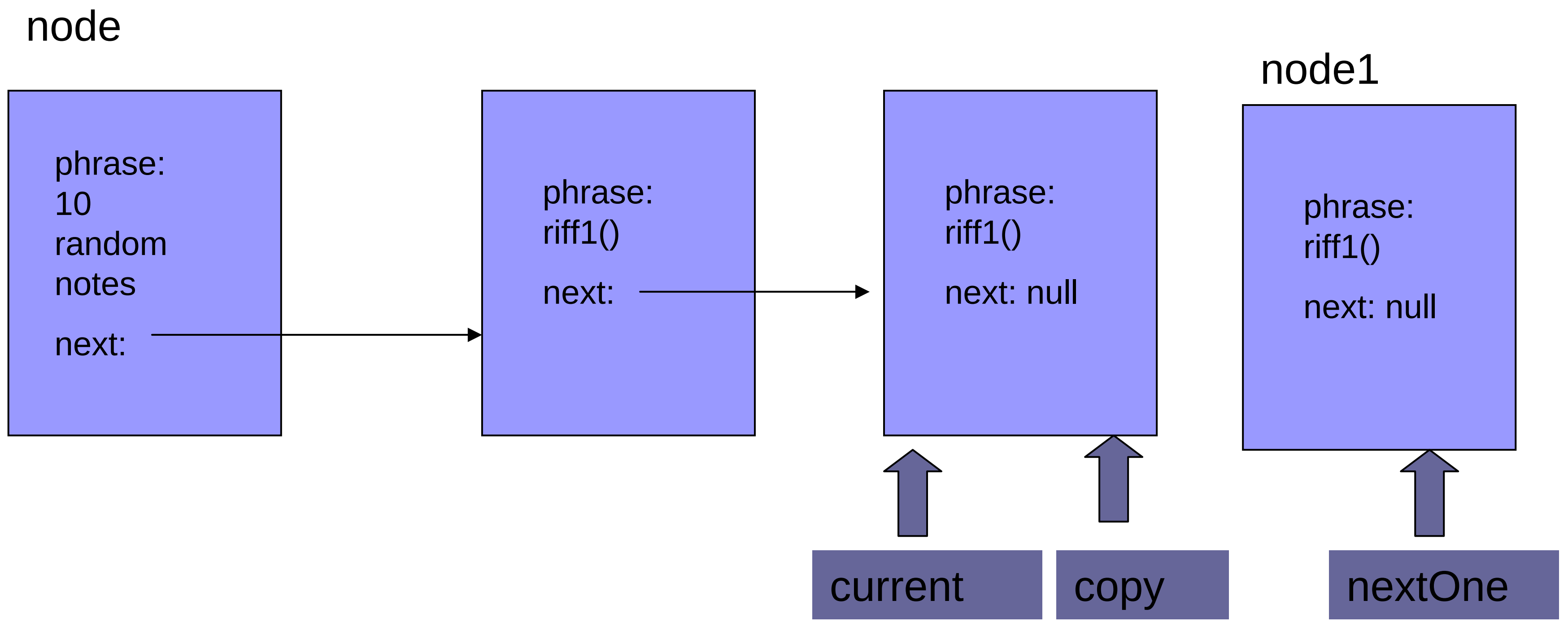
Step 5 & 6:

```
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
   }
 }
```

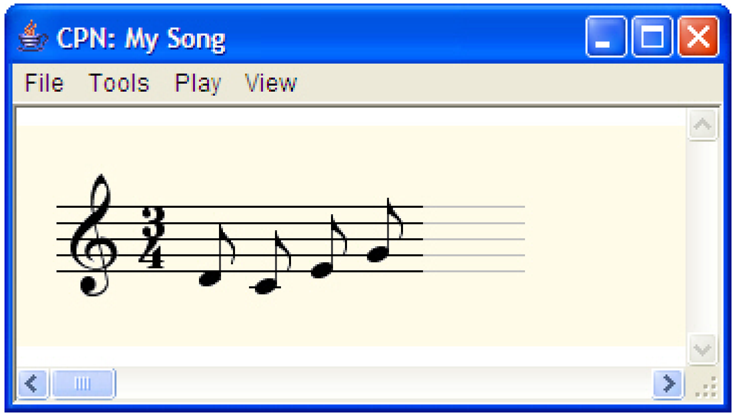
Weaving

```
/**
 * 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);
```

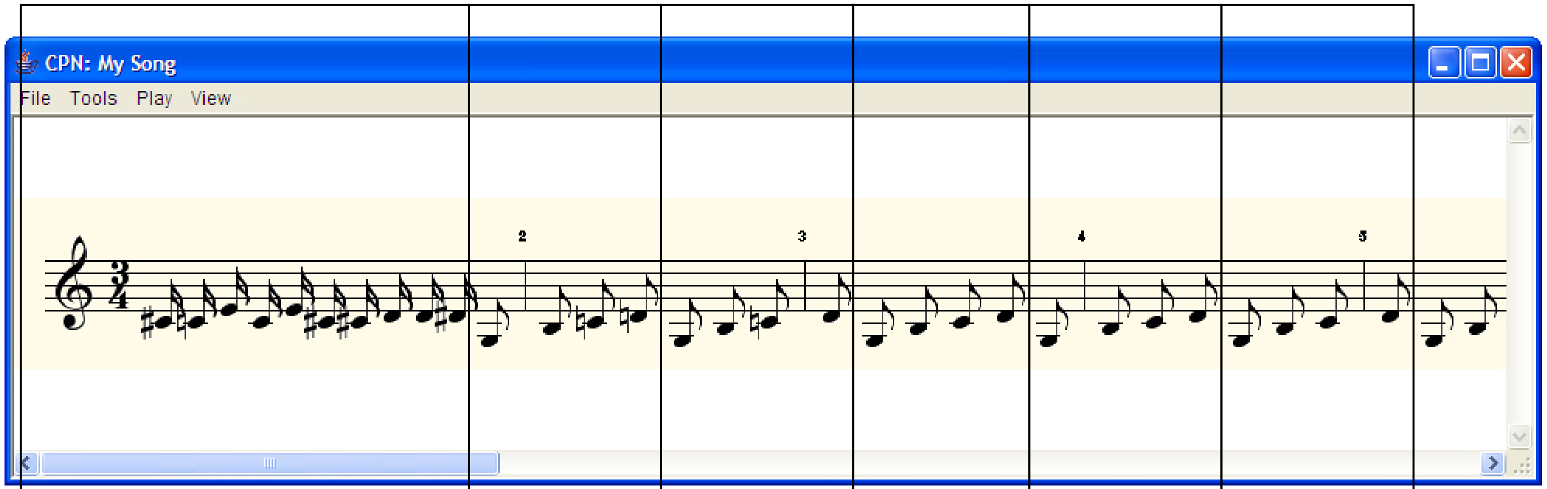


Doing a weave

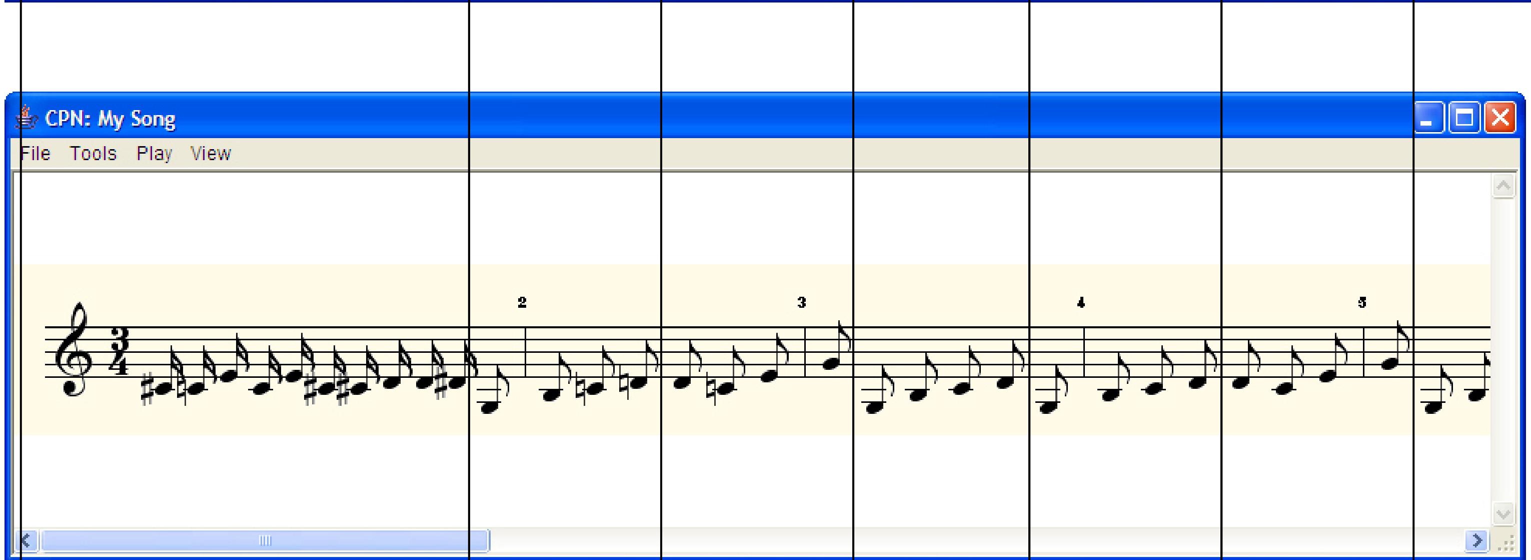
```
> node.weave(node2,4,2);
> node.showFromMeOn(JMC.PIANO);
```

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
  }
```

Shifting to Images

- Now that we’ve introduced linked lists with MIDI, we shift to images.
  - Briefly, two kinds of linked lists.
  - Then scene graphs

Building a Scene

- Computer graphics professionals work at two levels:
    - They define individual characters and effects on characters in terms of pixels.
    - But then most of their work is in terms of the *scene*: Combinations of images (characters, effects on characters).
- To describe scenes, they often use *linked lists* and *trees* in order to assemble the pieces.

Version 1: PositionedSceneElement

```
> FileChooser.setMediaPath("D:/cs1316/MediaSources/");
> PositionedSceneElement tree1 = new PositionedSceneElement(new
    Picture(FileChooser.getMediaPath("tree-blue.jpg")));
> PositionedSceneElement tree2 = new PositionedSceneElement(new
    Picture(FileChooser.getMediaPath("tree-blue.jpg")));
> PositionedSceneElement tree3 = new PositionedSceneElement(new
    Picture(FileChooser.getMediaPath("tree-blue.jpg")));
> PositionedSceneElement doggy = new PositionedSceneElement(new
    Picture(FileChooser.getMediaPath("dog-blue.jpg")));
> PositionedSceneElement house = new PositionedSceneElement(new
    Picture(FileChooser.getMediaPath("house-blue.jpg")));
> Picture bg = new Picture(FileChooser.getMediaPath("jungle.jpg"));
> tree1.setNext(tree2); tree2.setNext(tree3); tree3.setNext(doggy);
    doggy.setNext(house);
> tree1.drawFromMeOn(bg);
> bg.show();
```

In this example, using chromakey to compose..just for the fun of it.

What this looks like:

Slightly different ordering: Put the doggy between tree2 and tree3

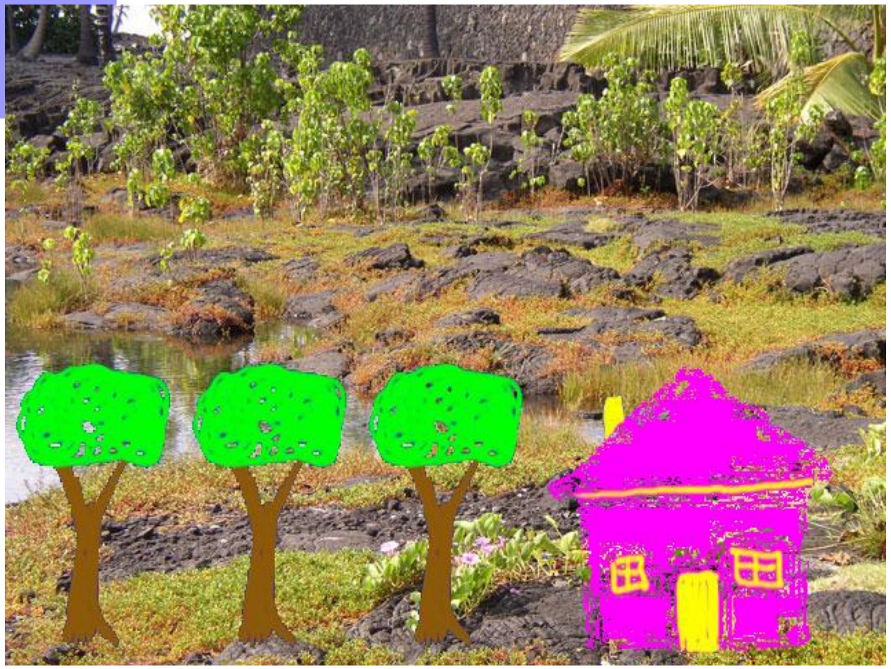
```
> tree3.setNext(house); tree2.setNext(doggy);
   doggy.setNext(tree3);
> bg = new
   Picture(FileChooser.getMediaPath("jungle.jpg"));
> tree1.drawFromMeOn(bg);
> bg.show();
```

Yes, we can put multiple statements in one line.

Slightly different picture

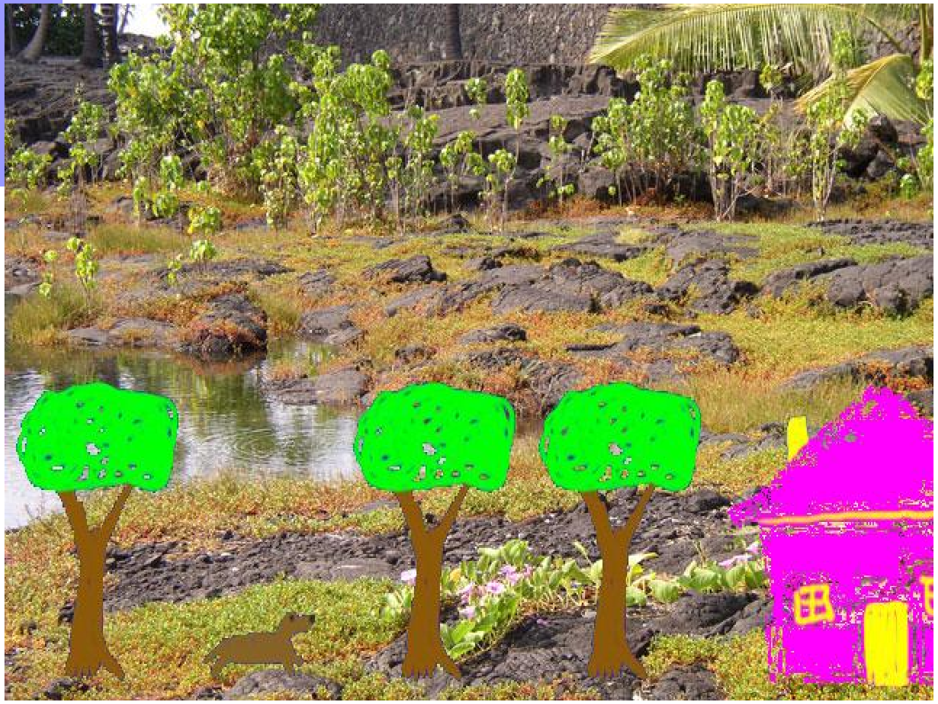
Removing the doggy

```
> tree1.setNext(tree2);
   tree2.setNext(tree3);
   tree3.setNext(doggy);
   doggy.setNext(house);
> tree1.remove(doggy);
> tree1.drawFromMeOn(bg);
```



Putting the mutt back

```
> bg = new
   Picture(FileChooser.getMediaPa
   th("jungle.jpg"));
> tree1.insertAfter(doggy);
> tree1.drawFromMeOn(bg);
```



Animation = (Changing a structure + rendering) * n

- We can use what we just did to create *animation.*
- Rather than think about animation as “a series of frames,”
- Think about it as:
  - Repeatedly:
    - Change a data structure
    - *Render* (draw while traversing) the data structure to create a frame

AnimatedPositionedScene

```
public class AnimatedPositionedScene {

  /**
   * A FrameSequence for storing the frames
   **/
  FrameSequence frames;

  /**
   * We'll need to keep track
   * of the elements of the scene
   **/
  PositionedSceneElement tree1, tree2, tree3, house, doggy, doggyflip;
```

Setting up the animation

```
public void setUp(){
  frames = new FrameSequence("D:/Temp/");


    FileChooser.setMediaPath("D:/cs1316/mediasource
    s/");
  Picture p = null; // Use this to fill elements

  p = new Picture(FileChooser.getMediaPath("tree-
    blue.jpg"));
  tree1 = new PositionedSceneElement(p);

  p = new Picture(FileChooser.getMediaPath("tree-
    blue.jpg"));
  tree2 = new PositionedSceneElement(p);

  p = new Picture(FileChooser.getMediaPath("tree-
    blue.jpg"));
  tree3 = new PositionedSceneElement(p);

  p = new Picture(FileChooser.getMediaPath("house-
    blue.jpg"));
  house = new PositionedSceneElement(p);

  p = new Picture(FileChooser.getMediaPath("dog-
    blue.jpg"));
  doggy = new PositionedSceneElement(p);
  doggyflip = new PositionedSceneElement(p.flip());
}
```

Render the first frame

```
public void make(){
   frames.show();

   // First frame
   Picture bg = new
   Picture(FileChooser.getMediaPath("jungle.jpg"));
   tree1.setNext(doggy); doggy.setNext(tree2);
   tree2.setNext(tree3);
   tree3.setNext(house);
   tree1.drawFromMeOn(bg);
   frames.addFrame(bg);
```

Render the doggy moving right

```
// Dog moving right
   bg = new Picture(FileChooser.getMediaPath("jungle.jpg"));
   tree1.remove(doggy);
   tree2.insertAfter(doggy);
   tree1.drawFromMeOn(bg);
   frames.addFrame(bg);

   bg = new Picture(FileChooser.getMediaPath("jungle.jpg"));
   tree1.remove(doggy);
   tree3.insertAfter(doggy);
   tree1.drawFromMeOn(bg);
   frames.addFrame(bg);

   bg = new Picture(FileChooser.getMediaPath("jungle.jpg"));
   tree1.remove(doggy);
   house.insertAfter(doggy);
   tree1.drawFromMeOn(bg);
   frames.addFrame(bg);
```

Moving left

```
//Dog moving left
   bg = new Picture(FileChooser.getMediaPath("jungle.jpg"));
   tree1.remove(doggy);
   house.insertAfter(doggyflip);
   tree1.drawFromMeOn(bg);
   frames.addFrame(bg);

   bg = new Picture(FileChooser.getMediaPath("jungle.jpg"));
   tree1.remove(doggyflip);
   tree3.insertAfter(doggyflip);
   tree1.drawFromMeOn(bg);
   frames.addFrame(bg);

   bg = new Picture(FileChooser.getMediaPath("jungle.jpg"));
   tree1.remove(doggyflip);
   tree2.insertAfter(doggyflip);
   tree1.drawFromMeOn(bg);
   frames.addFrame(bg);

   bg = new Picture(FileChooser.getMediaPath("jungle.jpg"));
   tree1.remove(doggyflip);
   tree1.insertAfter(doggyflip);
   tree1.drawFromMeOn(bg);
   frames.addFrame(bg);

 }
```

Results

Version 2: Layering

```
> Picture bg = new Picture(400,400);
> LayeredSceneElement tree1 = new LayeredSceneElement(
new Picture(FileChooser.getMediaPath("tree-blue.jpg")),10,10);
> LayeredSceneElement tree2 = new LayeredSceneElement(
new Picture(FileChooser.getMediaPath("tree-blue.jpg")),100,10);
> LayeredSceneElement tree3 = new LayeredSceneElement(
new Picture(FileChooser.getMediaPath("tree-blue.jpg")),200,100);
> LayeredSceneElement house = new LayeredSceneElement(
new Picture(FileChooser.getMediaPath("house-blue.jpg")),175,175);
> LayeredSceneElement doggy = new LayeredSceneElement(
new Picture(FileChooser.getMediaPath("dog-blue.jpg")),150,325);
> tree1.setNext(tree2); tree2.setNext(tree3); tree3.setNext(doggy);
    doggy.setNext(house);
> tree1.drawFromMeOn(bg);
> bg.show();
```

First version of Layered Scene

Reordering the layering

```
> house.setNext(doggy); doggy.setNext(tree3);
  tree3.setNext(tree2); tree2.setNext(tree1);
> tree1.setNext(null);
> bg = new Picture(400,400);
> house.drawFromMeOn(bg);
> bg.show();
```

Basically, we're reversing the list

Reordered (relayered) scene

Think about what’s involved in creating a *method* to *reverse()* a list…

What’s the difference?

- If we were in PowerPoint or Visio, you’d say that we changed the *layering*.
  - “Bring to front”
  - “Send to back”
  - “Bring forward”
  - “Send backward”

These commands are actually changing the *ordering* of the layers in the *list* of things to be redrawn.

• Change the ordering in the list.

• *Render* the scene

• Now it’s a different layering!

Version 3: A List with Both

- Problem 1: Why should we have *only* layered scene elements *or* positioned scene elements?
- Can we have both?
  - SURE! If *each* element knows how to draw itself!
  - But they took different parameters!
    - Layered got their (x,y) passed in.
  - It works if we always pass in a *turtle* that's set to the right place to draw if it's positioned (and let the layered ones do whatever they want!)
- Problem 2: Why is there so much duplicated code?
  - Why do only layered elements know last() and add()?

Using Superclasses

- What we really want is to define a class *SceneElement*
  - That knows most of being a picture element.
  - It would be an *abstract* class because we don’t actually mean to ever create instances of *THAT* class.
- Then create *subclasses*: *SceneElementPositioned* and *SceneElementLayered*
  - We’d actually use these.

Class Structure

Abstract Class *SceneElement*

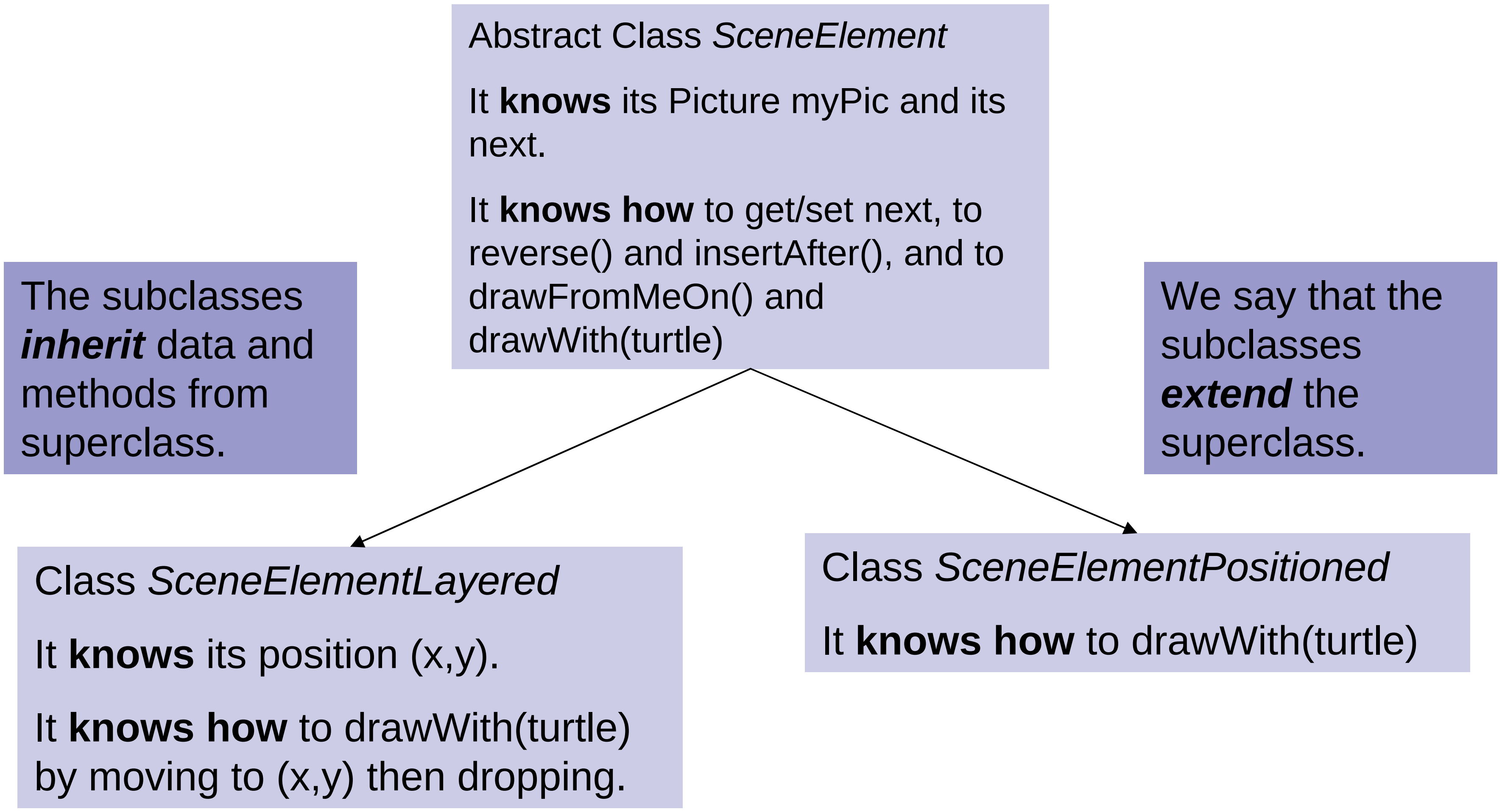
It **knows** its Picture myPic and its next (SceneElement).

It **knows how** to get/set next, to reverse() and insertAfter(), and to drawFromMeOn().

It **defines** drawWith(turtle), but leaves it for its subclasses do complete.

An abstract class defines structure and behavior that subclasses will inherit.

Class Structure



Abstract Class *SceneElement*

It **knows** its Picture myPic and its next.

It **knows how** to get/set next, to reverse() and insertAfter(), and to drawFromMeOn() and drawWith(turtle)

The subclasses ***inherit*** data and methods from superclass.

We say that the subclasses ***extend*** the superclass.

Class *SceneElementLayered*

It **knows** its position (x,y).

It **knows how** to drawWith(turtle) by moving to (x,y) then dropping.

Class *SceneElementPositioned*

It **knows how** to drawWith(turtle)

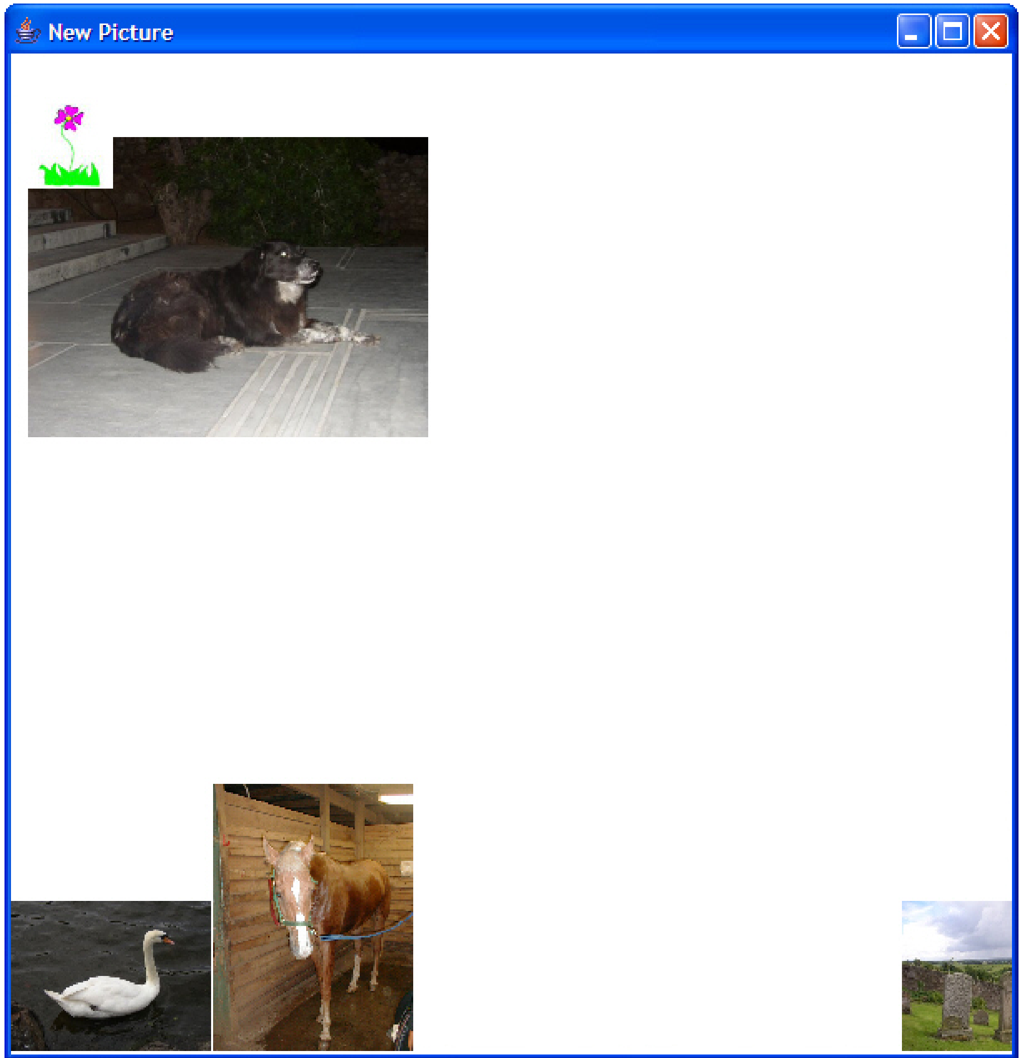
Using the new structure

```
public class MultiElementScene {

  public static void main(String[] args){
    FileChooser.setMediaPath("D:/cs1316/mediasources/");

    // We'll use this for filling the nodes
    Picture p = null;

    p = new Picture(FileChooser.getMediaPath("swan.jpg"));
    SceneElement node1 = new SceneElementPositioned(p.scale(0.25));
    p = new Picture(FileChooser.getMediaPath("horse.jpg"));
    SceneElement node2 = new SceneElementPositioned(p.scale(0.25));
    p = new Picture(FileChooser.getMediaPath("dog.jpg"));
    SceneElement node3 = new SceneElementLayered(p.scale(0.5),10,50);
    p = new Picture(FileChooser.getMediaPath("flower1.jpg"));
    SceneElement node4 = new SceneElementLayered(p.scale(0.5),10,30);
    p = new Picture(FileChooser.getMediaPath("graves.jpg"));
    SceneElement node5 = new SceneElementPositioned(p.scale(0.25));
```

Rendering the scene

```
    node1.setNext(node2); node2.setNext(node3);
    node3.setNext(node4); node4.setNext(node5);

    // Now, let's see it!
    Picture bg = new Picture(600,600);
    node1.drawFromMeOn(bg);
    bg.show();
  }
}
```

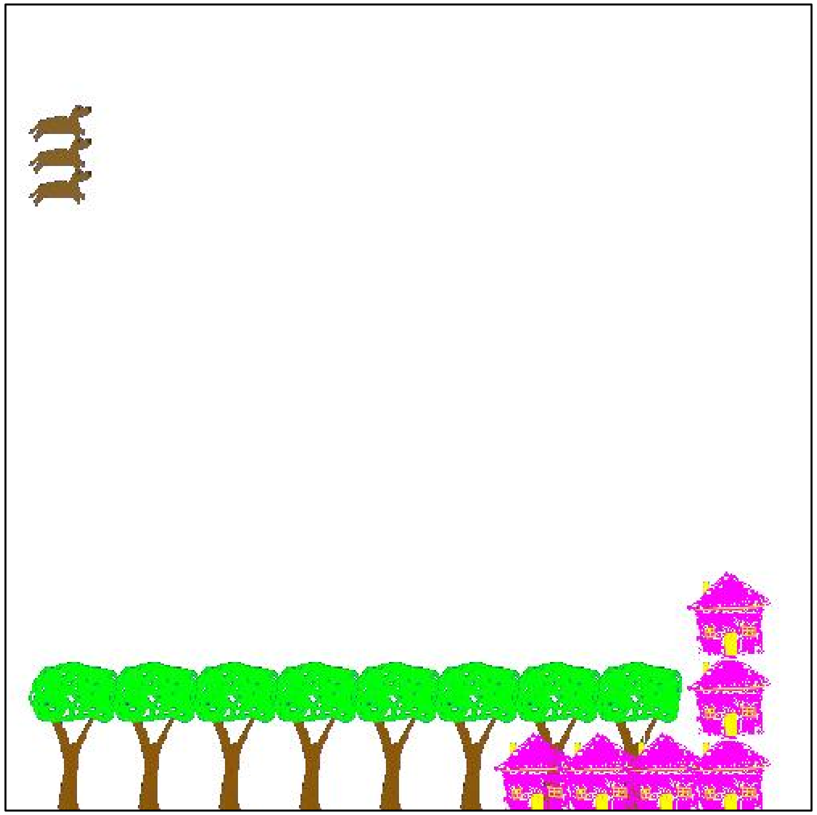
Rendered scene



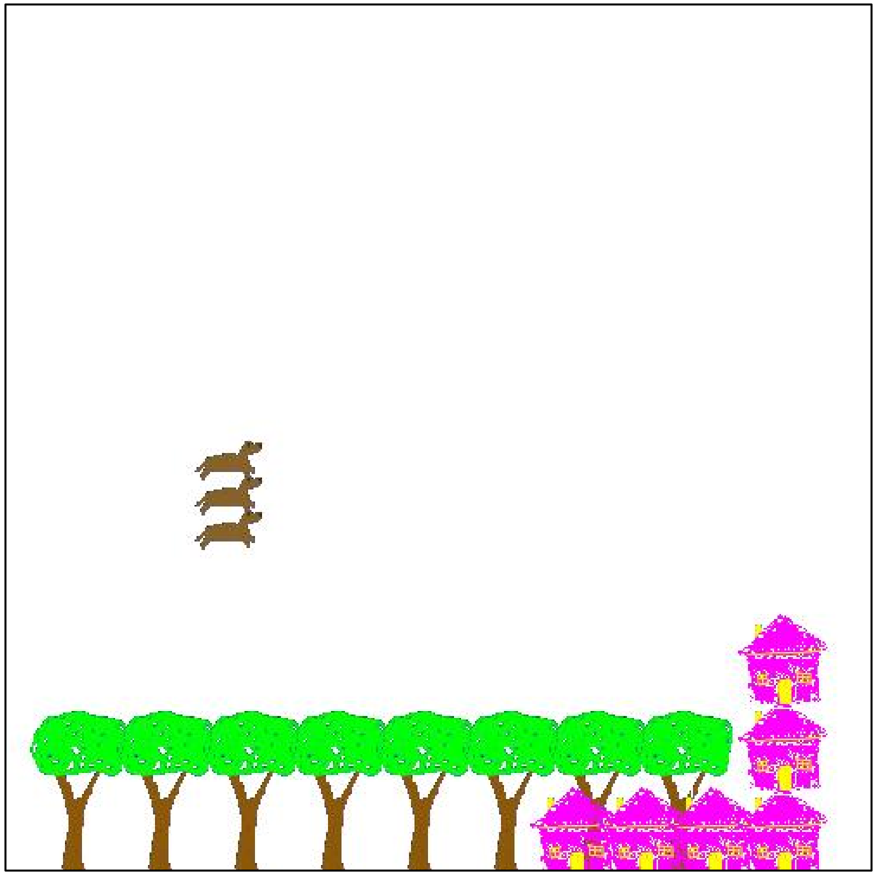
New Version: Trees for defining scenes

- Not everything in a scene is a single list.
  - Think about a pack of fierce doggies, er, wolves attacking the quiet village in the forest.
  - Real scenes *cluster*.
- Is it the responsibility of the elements to know about layering and position?
  - Is that the right place to put that *know how*?
- How do we structure *operations* to perform to sets of nodes?
  - For example, moving a set of them at once?

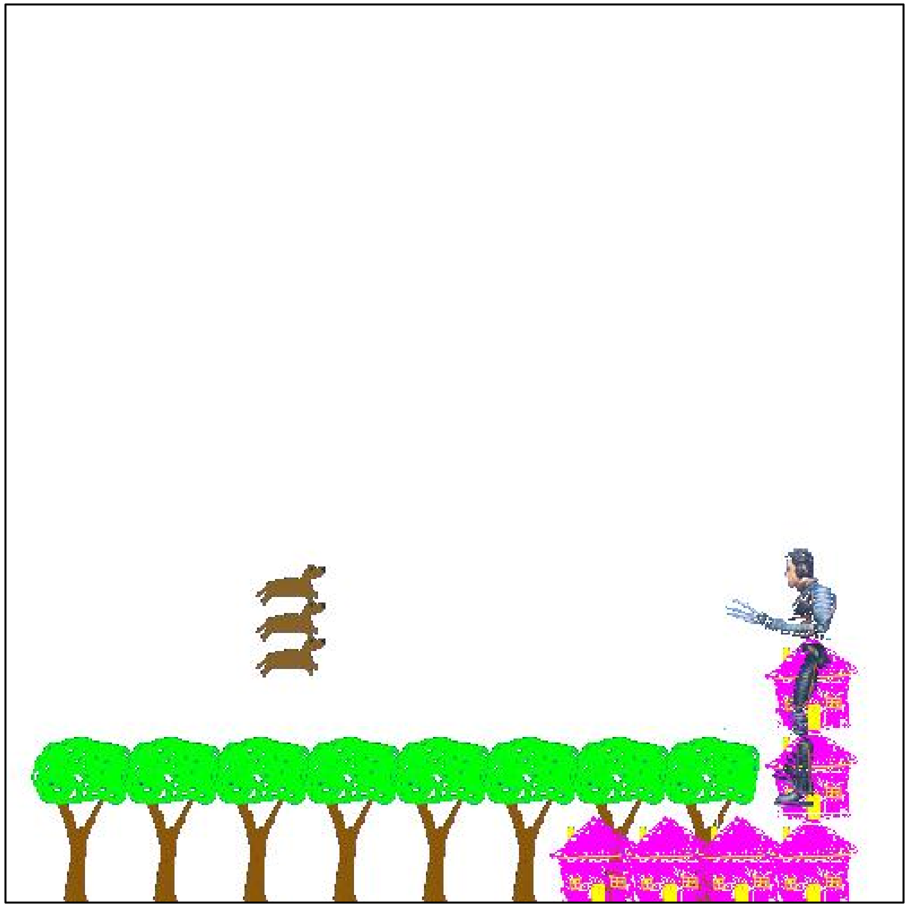
The Attack of the Nasty Wolvies



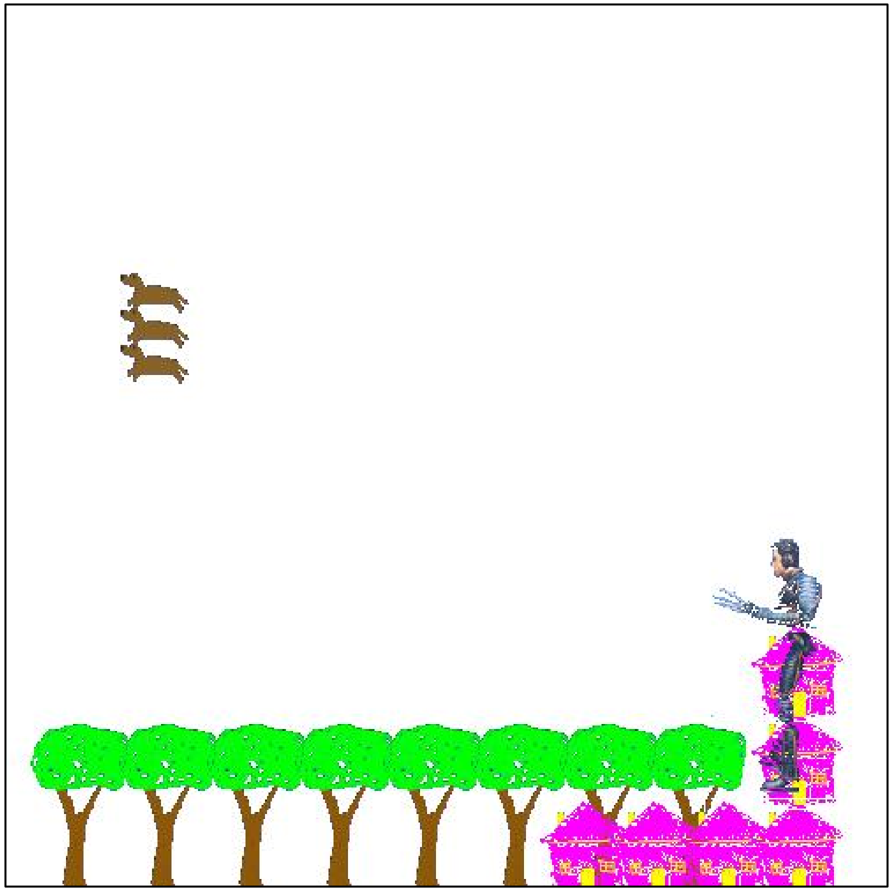
Closer…



Then the Hero Appears!



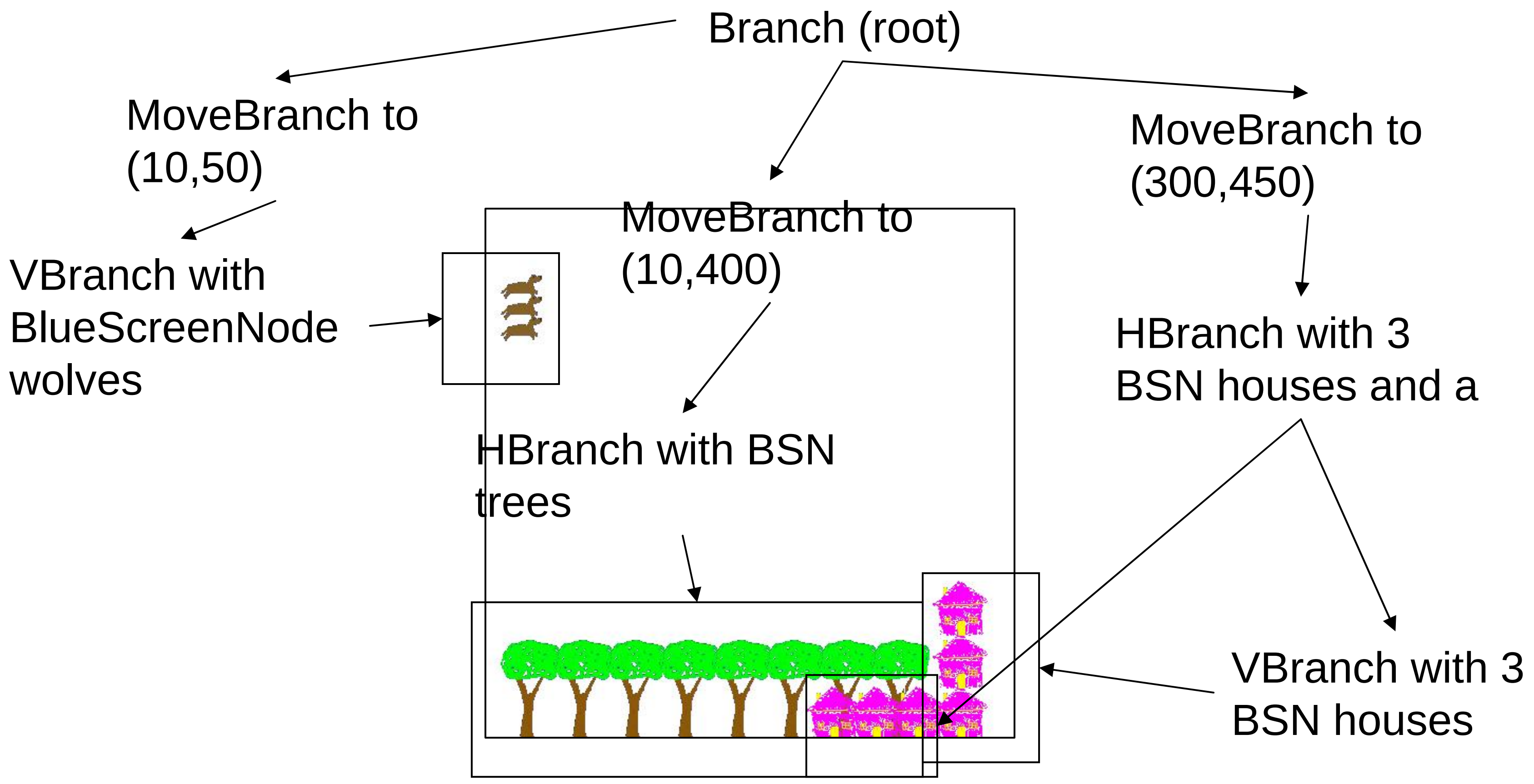
And the Wolvies retreat



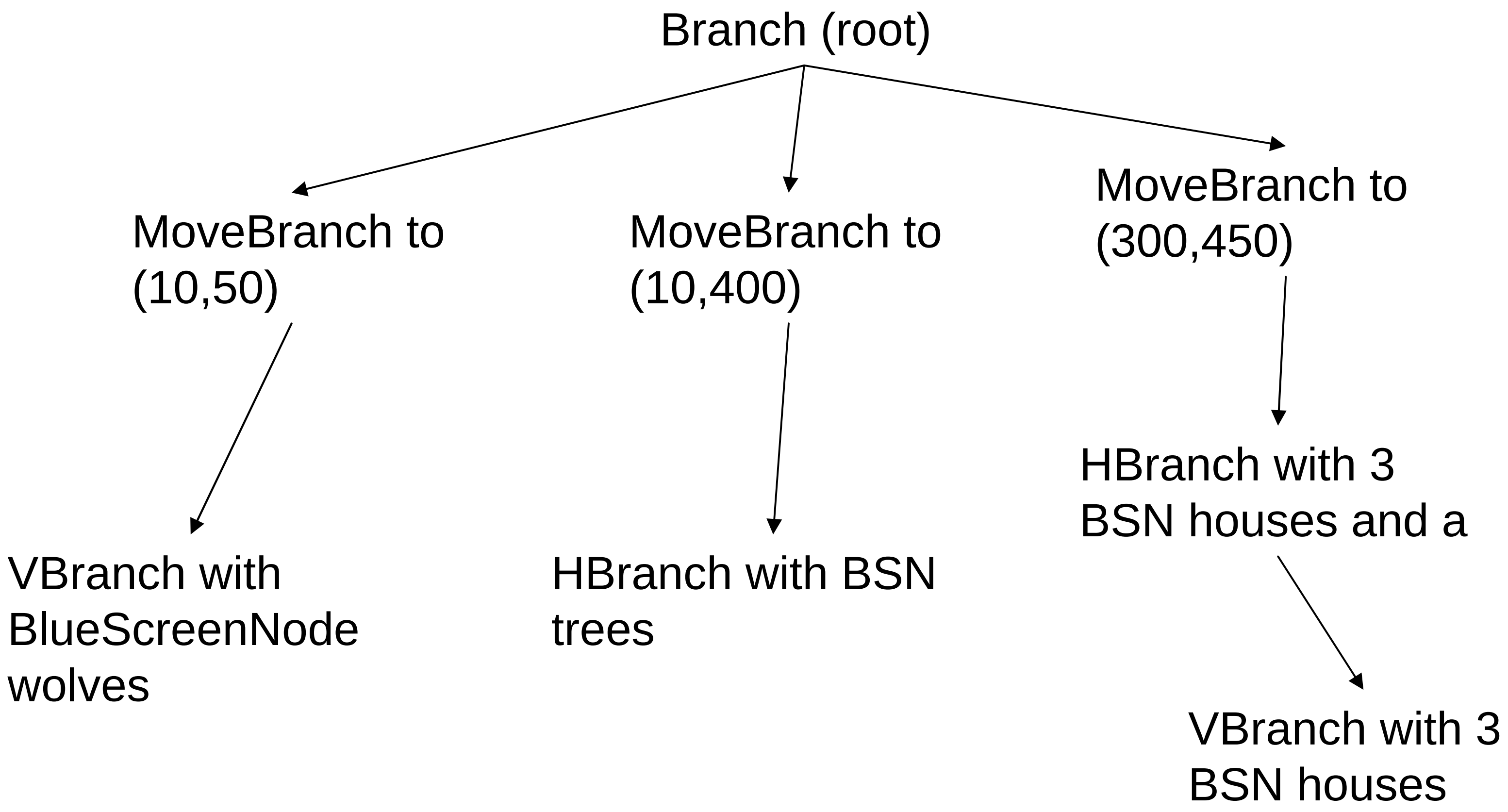
What's underlying this

- This scene is described by a *tree*
  - Each picture is a *BlueScreenNode* in this tree.
  - Groups of pictures are organized in *HBranch* or *VBranch* (Horizontal or Vertical branches)
  - The root of the tree is just a *Branch.*
  - The branches are positioned using a *MoveBranch*.

Labeling the Pieces



It’s a Tree


Branch (root)
MoveBranch to (10,50)
MoveBranch to (10,400)
MoveBranch to (300,450)
VBranch with BlueScreenNode wolves
HBranch with BSN trees
HBranch with 3 BSN houses and a
VBranch with 3 BSN houses

The Class Structure

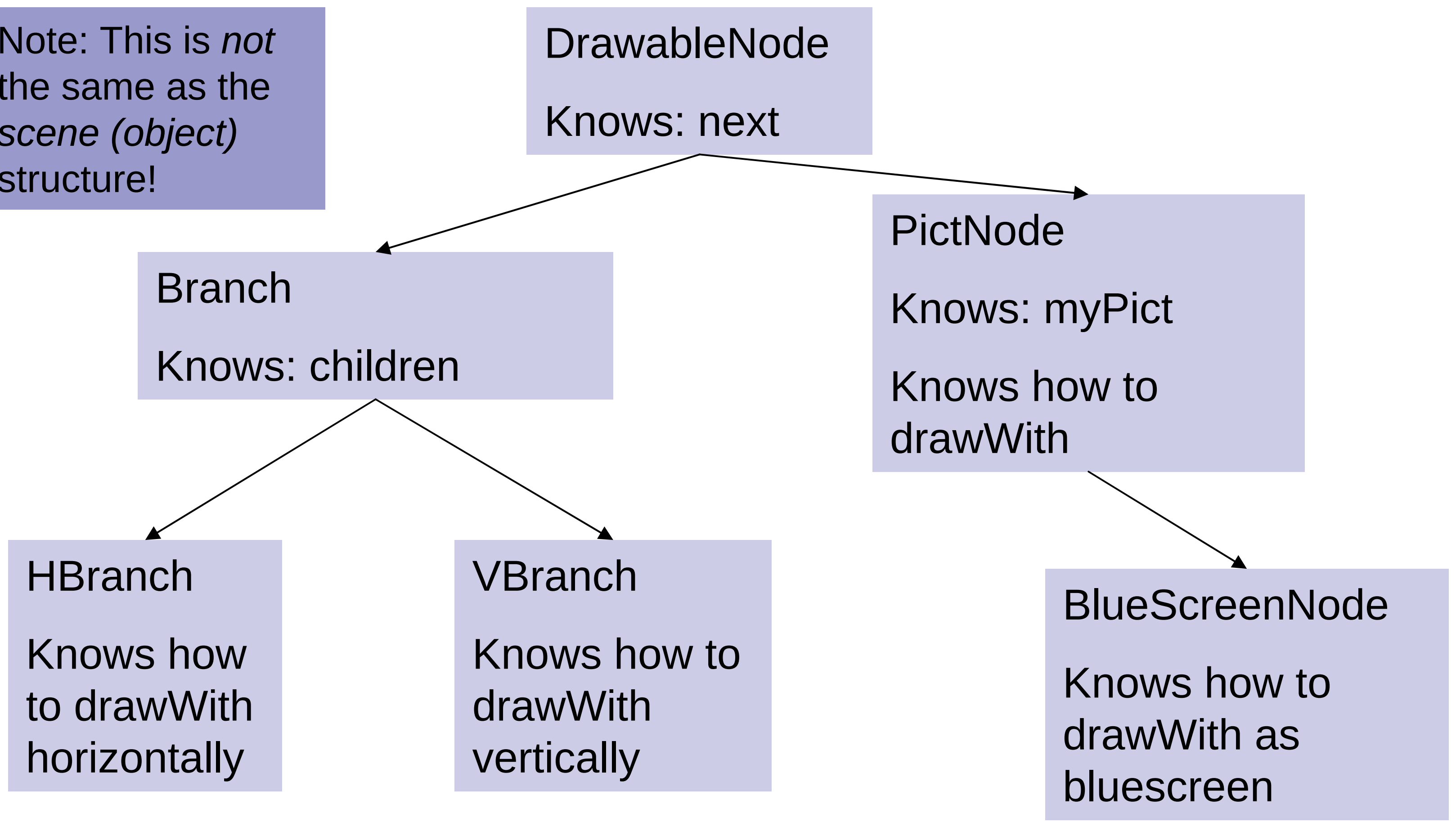
- *DrawableNode* **knows** only *next*, but **knows how** to do everything that our picture linked lists do (insertAfter, remove, last, drawOn(picture)).
  - Everything else is a subclass of that.
- *PictNode* **knows** it's *Picture myPict* and **knows how** to drawWith(turtle) (by dropping a picture)
- *BlueScreenNode* doesn't know new from *PictNode* but **knows how** to drawWith(turtle) by using bluescreen.

Branch Class Structure

- *Branch* **knows** its *children*—a linked list of other nodes to draw. It **knows how** to drawWith by:
  - (1) telling all its children to draw.
  - (2) then telling all its children to draw.
- A **HBranch** draws its children by spacing them out horizontally.
- A **VBranch** draws its children by spacing them out vertically.

The Class Structure Diagram

Note: This is *not* the same as the *scene (object)* structure!



Using these Classes: When doggies go bad!

```
public class WolfAttackMovie {
  /**
   * The root of the scene data structure
   **/
  Branch sceneRoot;

  /**
   * FrameSequence where the animation
   * is created
   **/
  FrameSequence frames;

  /**
   * The nodes we need to track between methods
   **/
  MoveBranch wolfentry, wolfretreat, hero;
```

These are the nodes that change *during* the animation, so must be available outside the local method context

Setting up the pieces

```
/**
 * Set up all the pieces of the tree.
 **/
public void setUp(){
  Picture wolf = new Picture(FileChooser.getMediaPath("dog-
   blue.jpg"));
  Picture house = new Picture(FileChooser.getMediaPath("house-
   blue.jpg"));
  Picture tree = new Picture(FileChooser.getMediaPath("tree-
   blue.jpg"));
  Picture monster = new Picture(FileChooser.getMediaPath("monster-
   face3.jpg"));
```

Making a Forest

```
//Make the forest
  MoveBranch forest = new MoveBranch(10,400); // forest on the bottom
  HBranch trees = new HBranch(50); // Spaced out 50 pixels between
  BlueScreenNode treenode;
  for (int i=0; i < 8; i++) // insert 8 trees
  {treenode = new BlueScreenNode(tree.scale(0.5));
    trees.addChild(treenode);}
  forest.addChild(trees);
```

Make attacking wolves

```
// Make the cluster of attacking "wolves"
   wolfentry = new MoveBranch(10,50); // starting position
   VBranch wolves = new VBranch(20); // space out by 20 pixels
    between
   BlueScreenNode wolf1 = new BlueScreenNode(wolf.scale(0.5));
   BlueScreenNode wolf2 = new BlueScreenNode(wolf.scale(0.5));
   BlueScreenNode wolf3 = new BlueScreenNode(wolf.scale(0.5));
   wolves.addChild(wolf1);wolves.addChild(wolf2);
    wolves.addChild(wolf3);
   wolfentry.addChild(wolves);
```

Make retreating wolves

```
// Make the cluster of retreating "wolves"
   wolfretreat = new MoveBranch(400,50); // starting position
   wolves = new VBranch(20); // space them out by 20 pixels between
   wolf1 = new BlueScreenNode(wolf.scale(0.5).flip());
   wolf2 = new BlueScreenNode(wolf.scale(0.5).flip());
   wolf3 = new BlueScreenNode(wolf.scale(0.5).flip());
   wolves.addChild(wolf1);wolves.addChild(wolf2);
    wolves.addChild(wolf3);
   wolfretreat.addChild(wolves);
```

It takes a Village…

```
// Make the village
  MoveBranch village = new MoveBranch(300,450); // Village on bottom
  HBranch hhouses = new HBranch(40); // Houses are 40 pixels apart
   across
  BlueScreenNode house1 = new BlueScreenNode(house.scale(0.25));
  BlueScreenNode house2 = new BlueScreenNode(house.scale(0.25));
  BlueScreenNode house3 = new BlueScreenNode(house.scale(0.25));
  VBranch vhouses = new VBranch(-50); // Houses move UP, 50 pixels
   apart
  BlueScreenNode house4 = new BlueScreenNode(house.scale(0.25));
  BlueScreenNode house5 = new BlueScreenNode(house.scale(0.25));
  BlueScreenNode house6 = new BlueScreenNode(house.scale(0.25));
  vhouses.addChild(house4); vhouses.addChild(house5);
   vhouses.addChild(house6);
  hhouses.addChild(house1); hhouses.addChild(house2);
   hhouses.addChild(house3);
  hhouses.addChild(vhouses); // Yes, a VBranch can be a child of an
   HBranch!
  village.addChild(hhouses);
```

Making the village's hero

```
// Make the monster
   hero = new MoveBranch(400,300);
   BlueScreenNode heronode = new
  BlueScreenNode(monster.scale(0.75).flip());
   hero.addChild(heronode);
```

Assembling the Scene

```
//Assemble the base scene
   sceneRoot = new Branch();
   sceneRoot.addChild(forest);
   sceneRoot.addChild(village);
   sceneRoot.addChild(wolfentry);
 }
```

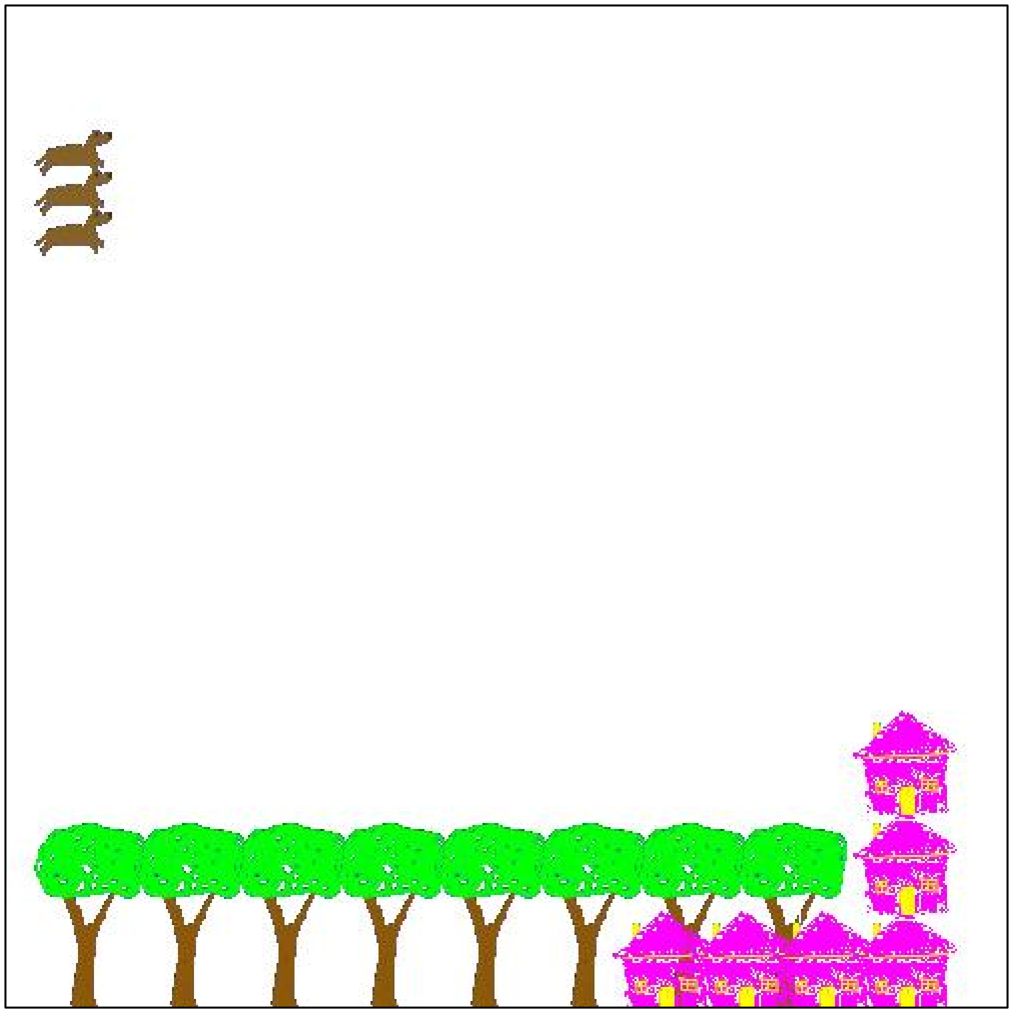
Want the forest on top of the village? Put the village in BEFORE the forest! Then it will get rendered first

Where's the wolfretreat and monster? They'll get inserted into the scene in the *middle* of the movie

Trying out one scene: Very important for testing!

```
/**
  * Render just the first scene
  **/
 public void renderScene() {
   Picture bg = new Picture(500,500);
   sceneRoot.drawOn(bg);
   bg.show();
 }
```

Okay that works



Rendering the whole movie

```
/**
  * Render the whole animation
  **/
 public void renderAnimation() {
   frames = new FrameSequence("D:/Temp/");
   frames.show();
   Picture bg;
```

Wolvies attack! (for 25 frames)

```
// First, the nasty wolvies come closer to the poor village
   // Cue the scary music
   for (int i=0; i<25; i++)
   {
     // Render the frame
     bg = new Picture(500,500);
     sceneRoot.drawOn(bg);
     frames.addFrame(bg);

     // Tweak the data structure
     wolfentry.moveTo(wolfentry.getXPos()+5,wolfentry.getYPos()+10);
   }
```

Inch-by-inch, er, 5-pixels by 10 pixels, they creep closer.

Our hero arrives! (In frame 26)

```
// Now, our hero arrives!
   this.root().addChild(hero);
   // Render the frame
   bg = new Picture(500,500);
   sceneRoot.drawOn(bg);
   frames.addFrame(bg);
```

Exit the threatening wolves, enter the retreating wolves

```
// Remove the wolves entering, and insert the wolves
   retreating
   this.root().children.remove(wolfentry);
   this.root().addChild(wolfretreat);
   // Make sure that they retreat from the same place that
   they were at
   wolfretreat.moveTo(wolfentry.getXPos(),
         wolfentry.getYPos());
   // Render the frame
   bg = new Picture(500,500);
   sceneRoot.drawOn(bg);
   frames.addFrame(bg);
```

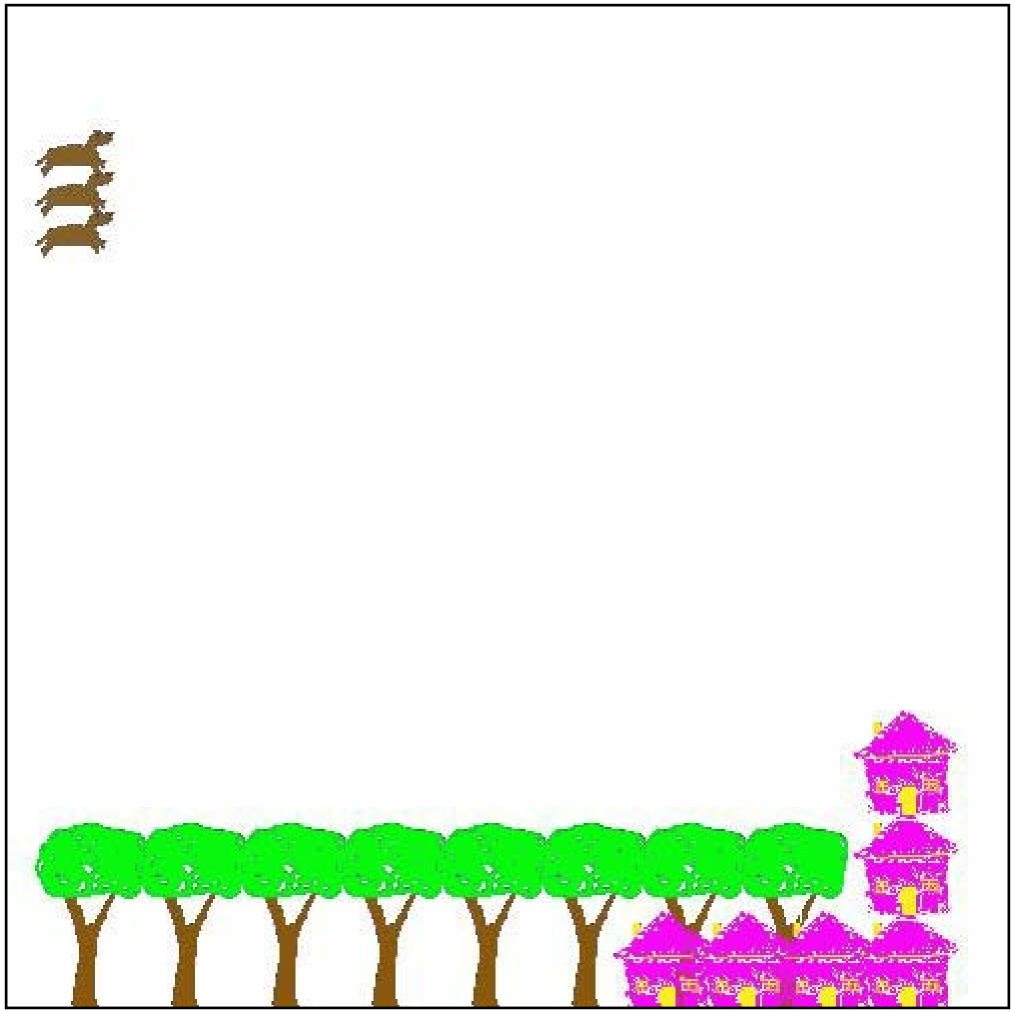
The wolves retreat (more quickly)

```
// Now, the cowardly wolves hightail it out of there!
   // Cue the triumphant music
   for (int i=0; i<10; i++)
   {
     // Render the frame
     bg = new Picture(500,500);
     sceneRoot.drawOn(bg);
     frames.addFrame(bg);

     // Tweak the data structure
     wolfretreat.moveTo(wolfretreat.getXPos()-10,
                              wolfretreat.getYPos()-20);
   }
 }
```

Making the Movie

```
Welcome to DrJava.
> WolfAttackMovie wam = new WolfAttackMovie();
    wam.setUp(); wam.renderScene();


> wam.renderAnimation();


There are no frames to show yet.  When you add a frame it
    will be shown


> wam.replay();
```

The Completed Movie



Homework Assignment!

- Option 1: Create linked list music
  - Create music with at least five calls to repeatInserting or weave
  - It must be at least 20 nodes long.
  - Make one new riff yourself and use it.
  - *Draw* the resultant sound structure
- Option 2: Make a movie—*with sound!*
  - Use a scene graph for the visuals, and a linked list for the sound.
    - Maybe play one node per frame?
  - Can use play() for background sounds, blockingPlay() for foreground sounds
  - Only rule: During rendering, cannot create any new sounds.
    - Must be in the linked list of sounds already.
- Example: HW4 and HW6 at http://coweb.cc.gatech.edu/cs1316/458

Simulations

- We build a simulation “from scratch”: Wolves and deer
  - Based on Anne Fleury’s work about students avoiding abstraction and preferring duplicating/explicit code.
  - We’re trying to make it easier to understand, and *then* introduce abstraction.
- Then we build a simulation package that makes it all easier.

Real programmers don’t make data structures…often

- Programmers almost never make arrays.
- Most programmers don’t make linked lists or trees or graphs, either!
  - Or hashtables/dictionaries, or heaps, or stacks and queues.
- These *core*, general, *abstract* data structures are typically provided in some form through libraries for the language.
  - That’s true for both Python/Jython and Java.

Real programmers make models…often

- The basic processes of modeling is something that every object-oriented programmer does all the time.
  - Aggregation: connecting objects together through references
  - Generalization and Specialization
- ***Learning how data structures work is learning about modeling.***

Real programmers make data structures…sometimes

- Sometimes you *do* make data structures.
  - If you need a specialized structure.
  - If you want just the methods you want in the way that you want them.
    - For example, Java’s **LinkedList** has no **insertAfter()!**
  - If you need it to work faster.

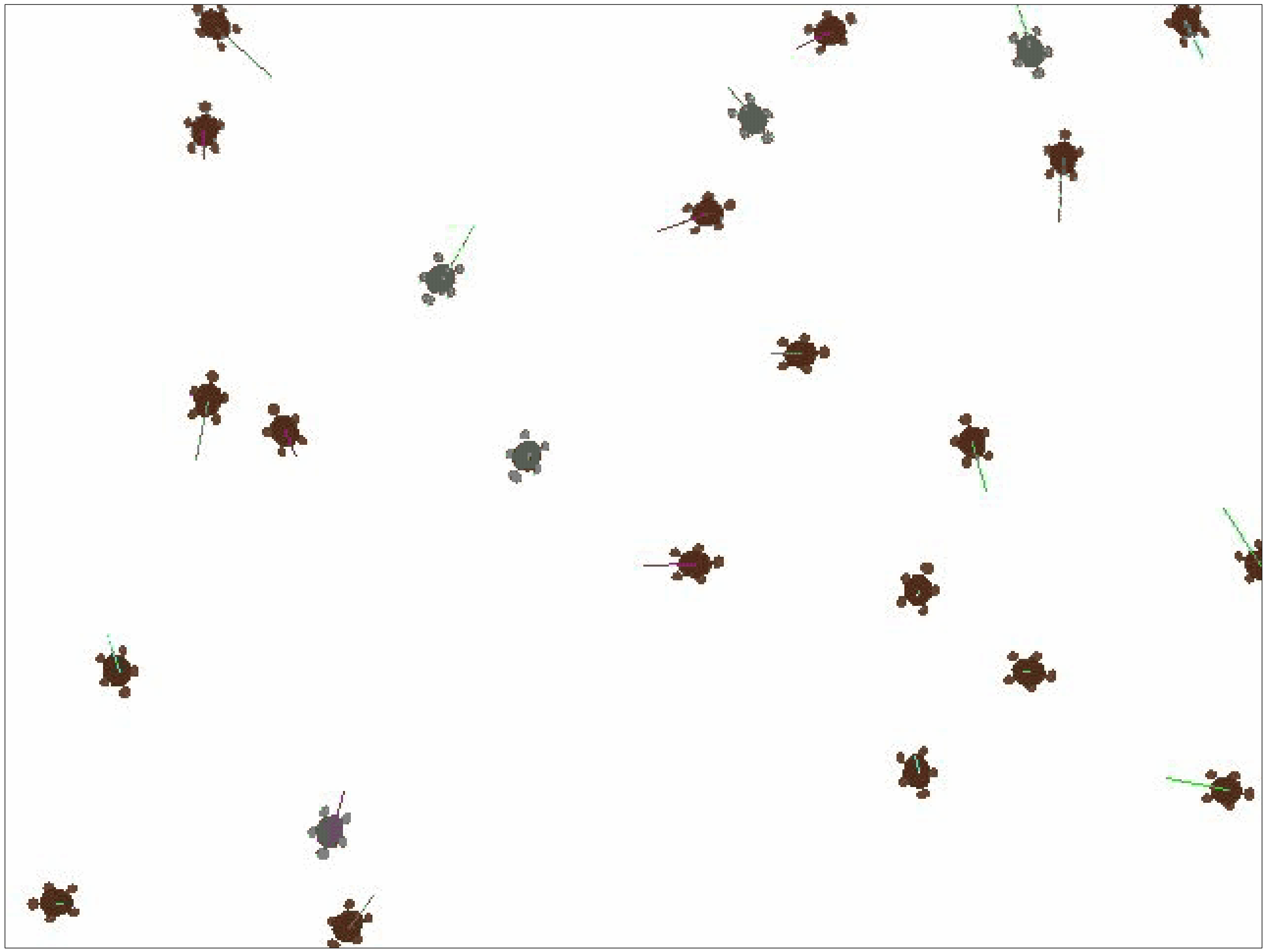
Real programmers make data structures *choices!*

- You *choose* between different data structures all the time.
- Often the choice is based on *running time*.
  - Arrays are faster than linked lists for some things (like accessing element *i*),
    while linked lists are faster for other things (like insertion and deletion).
- Sometimes the choice is for particular properties.
  - Use trees for clustering,
  - Use graphs for cycles,
  - Use hashtables for lookup by String, not index number

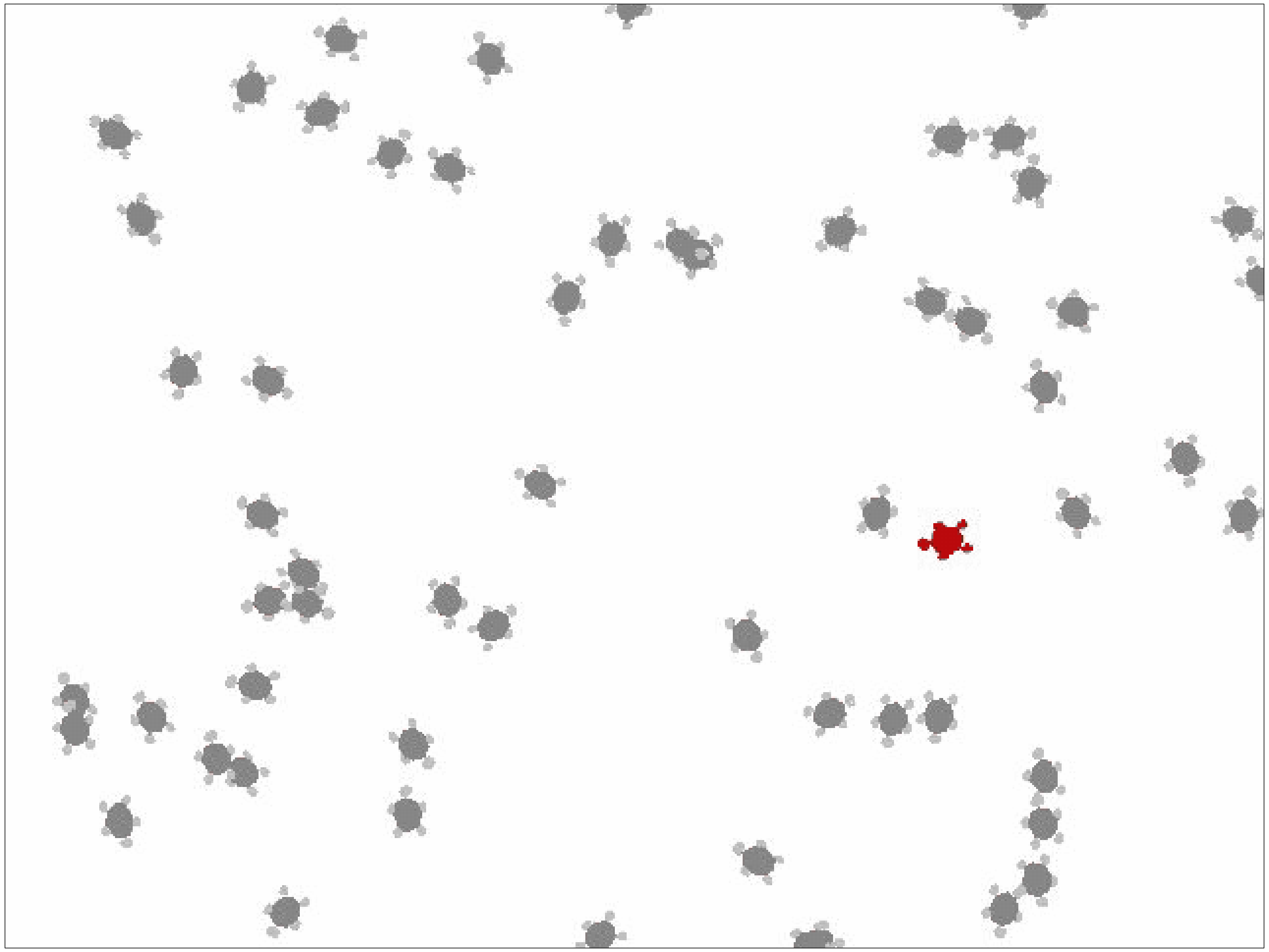
Building a Simulation Package

- Let’s make it *much* easier to build simulations.
  - We’ll use Java’s data structures, rather than build our own.
  - We’ll create **Simulation** and **Agent** as a general simulation, so that we only subclass them to create our specific simulations.
- A classic “real programmer” challenge: Making code designed to be *reused* (by us, but could be anyone) later.

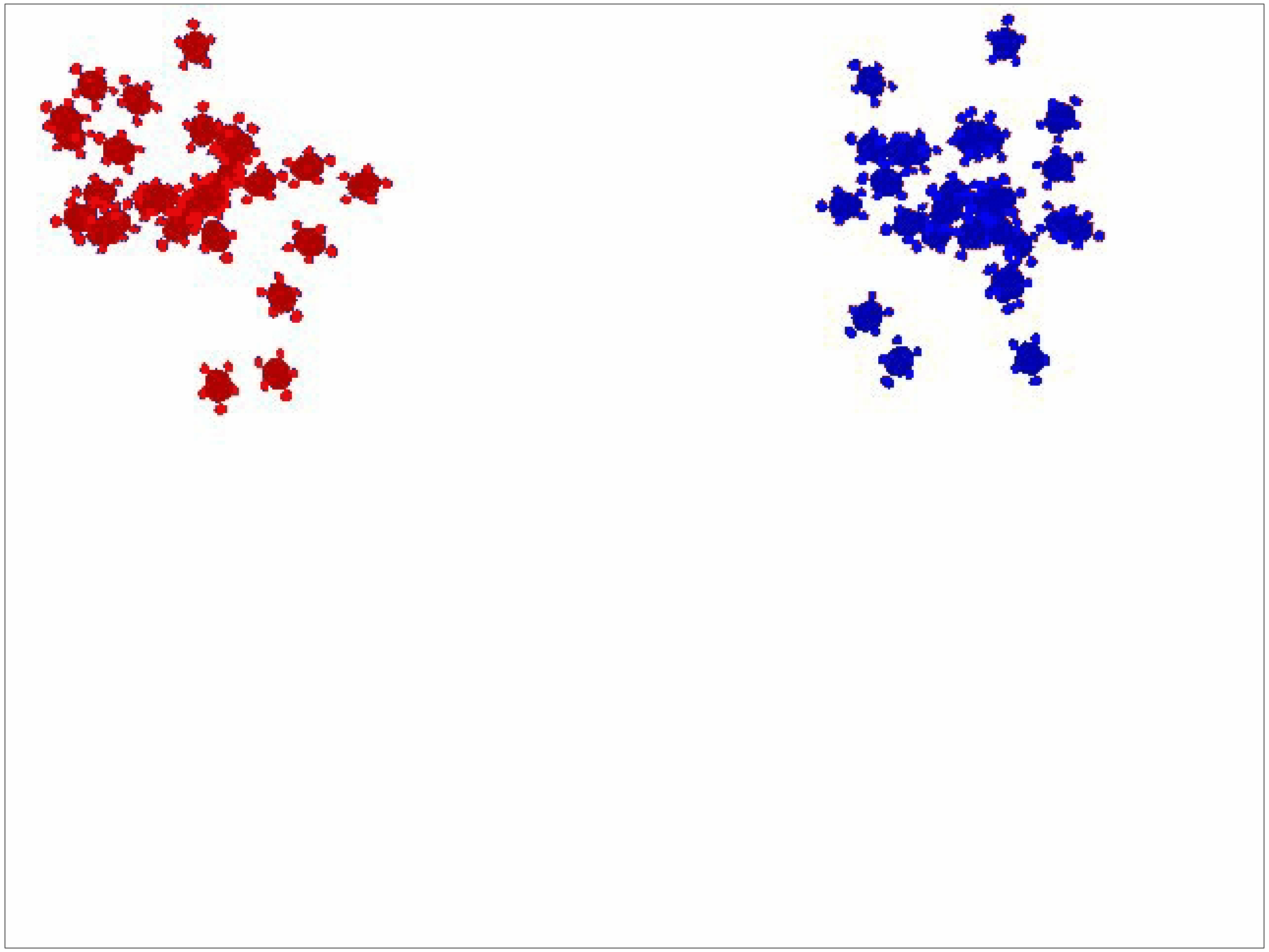
WDSimulation with new package



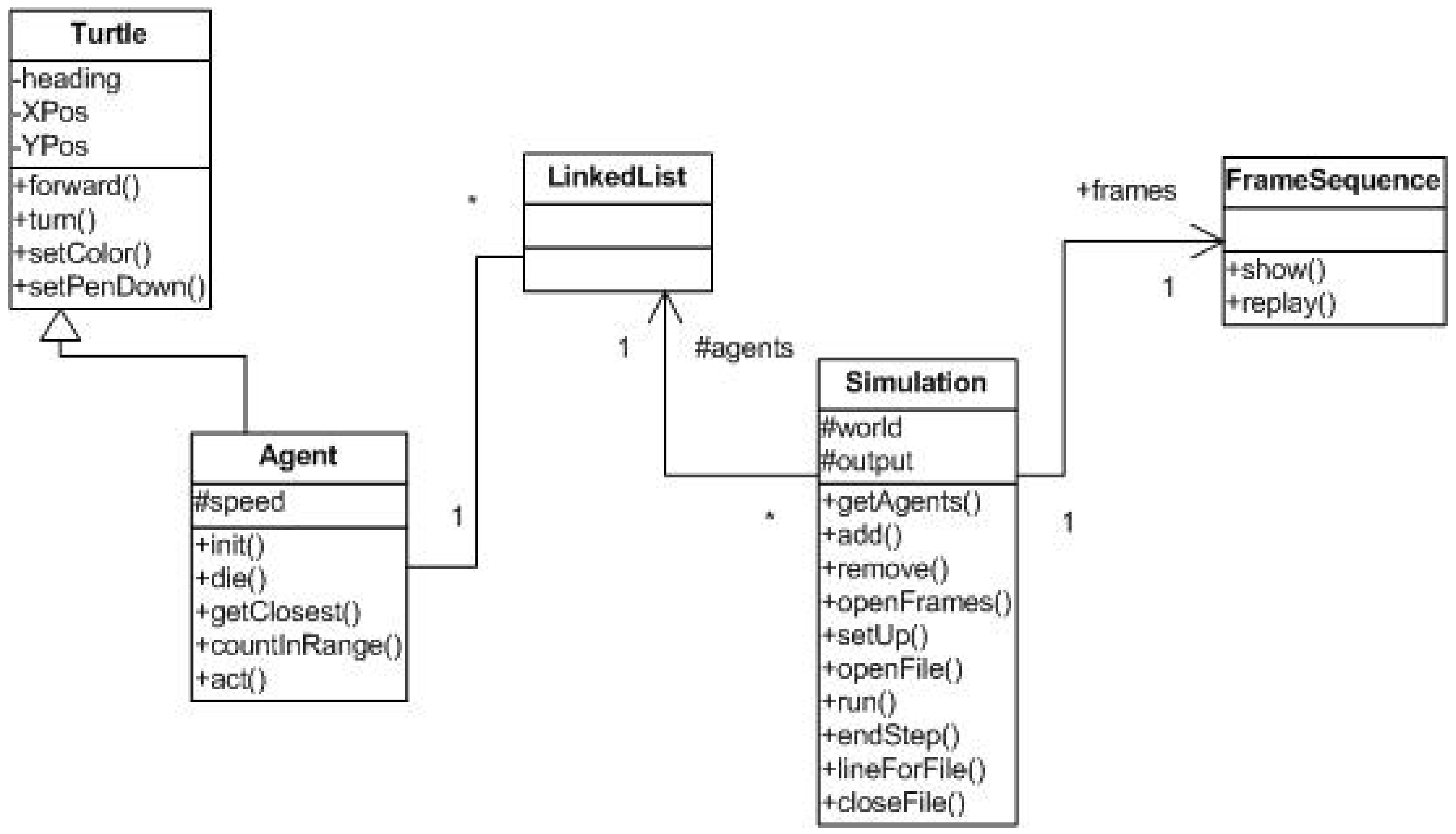
DiseaseSimulation



PoliticalSimulation



Design of the Package



How we use the package

- Subclass **Simulation** to define your general simulation.
  - Override the methods that you want to change.
  - Feel free to call **super.method()** to reuse the general functionality.
- Subclass **Agent** to define your simulation agents/actors.
  - Override the methods that you want to change.
  - Feel free to call **super.method**() to reuse the general functionality.

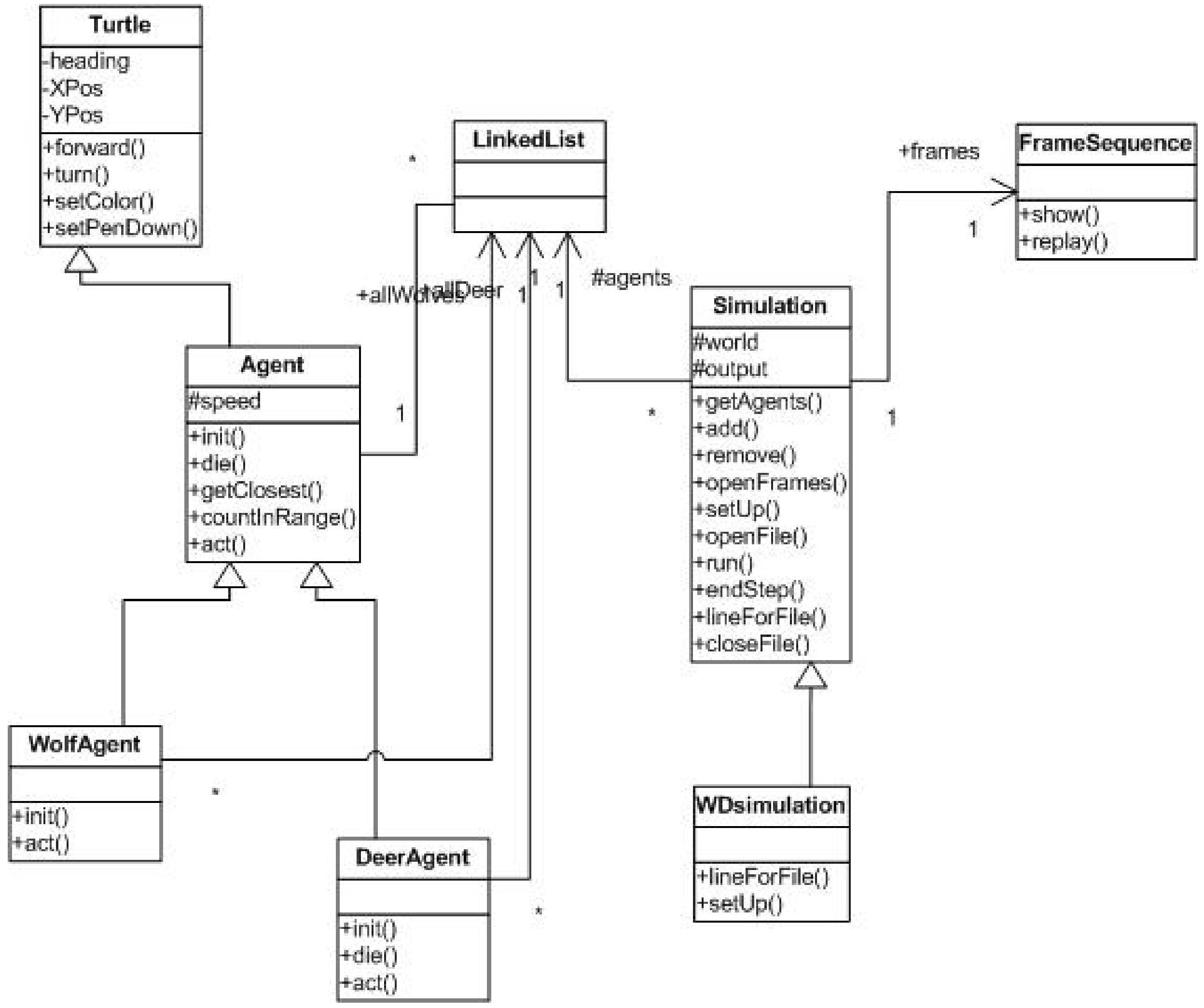
What *Simulation* provides

- getAgents(), add(), remove(): Manipulate the list of all agents
- setUp(): Open a world
- openFile(): Write data to a file
- openFrames(): Write frames to a FrameSequence
- run(): Run the simulation—for a number of timesteps, tell each agent to act()
- endStep(): Print the timestep and write to the file.
- lineForFile(): Define what to print to the file.
- closeFile(): End the file writing

What *Agent* provides

- setSpeed(), getSpeed(): Change/get speed
- init(): Add to simulation agents list
- die(): Make body red and remove from simulation agents list
- getClosest(): Get the closest agent from a list within a range.
- countInRange(): Count the agents within a range that are on a list.
- act(): By default, wander aimlessly

Redefining WDSimulation



WDSimulation

We need **setUp**() to define the world (let **super.setUp**() do that), then fill it with our agents.

```
/**
 * WDSimulation -- using the Simulation class
 **/
public class WDSimulation extends Simulation {

  /**
   * Fill the world with wolves and deer
   **/
  public void setUp(){
    // Let the world be set up
    super.setUp();

    // Just for storing the new deer and wolves
    DeerAgent deer;
    WolfAgent wolf;

    // create some deer
    int numDeer = 20;
    for (int i = 0; i < numDeer; i++)
    {
      deer = new DeerAgent(world,this);
    }

    // create some wolves
    int numWolves = 5;
    for (int i = 0; i < numWolves; i++)
    {
      wolf = new WolfAgent(world,this);
    }

  }
```

Writing out our counts in WDSimulation

```
/**
  * lineForFile -- write out number of wolves and deer
  **/
 public String lineForFile(){
   // Get the size (an int), make it an Integer,
   // in order to turn it into a string. (Whew!)
   return (new Integer(DeerAgent.allDeer.size())).toString()+"/t"+
     (new Integer(WolfAgent.allWolves.size())).toString();
 }
```

It's not easy to convert an integer (size() of the list) to a string.

Defining our Deer

```
import java.awt.Color; // Color for colorizing
import java.util.LinkedList;

/**
 * DeerAgent -- Deer as a subclass of Agent
 **/
public class DeerAgent extends Agent {

  /** class constant for the color */
  private static final Color brown = new Color(116,64,35);

  /** class constant for how far deer can smell */
  private static final double SMELL_RANGE = 50;

  /** Collection of all Deer */
  public static LinkedList allDeer = new LinkedList();
```

Notice **allDeer**!

It's a **LinkedList**—for free!

It's **static**—there's one list shared by *all* instances of the class. It's the list of *all* DeerAgents, and there's only *one* of these lists.

DeerAgent initialization

```
/**
  * Initialize, by adding to Deer list
  **/
 public void init(Simulation thisSim){
   // Do the normal initializations
   super.init(thisSim);

   // Make it brown
   setColor(brown);

   // Add to list of Deer
   allDeer.add(this);
 }
```

DeerAgent’s way of dying

```
/**
 * To die, do normal stuff, but
 * also remove from deer list
 **/
public void die(){
  super.die();
  allDeer.remove(this);
  System.out.println("Deer left: "+allDeer.size());
}
```

DeerAgent's actions

This is it folks!
It's all that we have to write to make **DeerAgents** work!

```
/**
 * How a DeerAgent acts
 **/
public void act()
{
  // get the closest wolf within the smell range
  WolfAgent closeWolf = (WolfAgent)
    getClosest(SMELL_RANGE,
                              WolfAgent.allWolves);

  if (closeWolf != null) {
    // Turn to face the wolf
    this.turnToFace(closeWolf);
    // Now directly in the opposite direction
    this.turn(180);
    // How far to run? How about half of current speed??
    this.forward((int) (speed/2));
  }
  else {
    // Run the normal act() -- wander aimlessly
    super.act();
  }
}
```

Constructors

```
///////////////////////////////// Constructors ////////////////////////
 // Copy this section AS-IS into subclasses, but rename Agent to
 // Your class.

/**
 * Constructor that takes the model display (the original
 * position will be randomly assigned)
 * @param modelDisplayer thing that displays the model
 * @param thisSim my simulation
 */
public DeerAgent (ModelDisplay modelDisplayer,Simulation thisSim)
{
  super(randNumGen.nextInt(modelDisplayer.getWidth()),
        randNumGen.nextInt(modelDisplayer.getHeight()),
        modelDisplayer, thisSim);
}

/** Constructor that takes the x and y and a model
 * display to draw it on
 * @param x the starting x position
 * @param y the starting y position
 * @param modelDisplayer the thing that displays the model
 * @param thisSim my simulation
 */
public DeerAgent (int x, int y, ModelDisplay modelDisplayer,
             Simulation thisSim)
{
  // let the parent constructor handle it
  super(x,y,modelDisplayer,thisSim);
}
```

DON'T care about this!

Copy it in as-is, and make the names match your class.

That's it. Period.

WolfAgent

```
import java.awt.Color;
import java.util.LinkedList;

/**
 * WolfAgent -- Wolf as a subclass of Agent
 **/
public class WolfAgent extends Agent {
  /** class constant for how far wolf can smell */
  private static final double SMELL_RANGE = 50;

    /** class constant for how close before wolf can attack */
  private static final double ATTACK_RANGE = 30;

  /** Collection of all Wolves */
  public static LinkedList allWolves = new LinkedList();
```

WolfAgent initializations

```
/**
  * Initialize, by adding to Wolf list
  **/
 public void init(Simulation thisSim){
   // Do the normal initializations
   super.init(thisSim);

   // Make it brown
   setColor(Color.gray);

   // Add to list of Wolves
   allWolves.add(this);
 }
```

WolfAgent act()

The same constructors are there, but let’s ignore those.

```
/**
  * Chase and eat the deer
  **/
   /**
  * Method to act during a time step
  * pick a random direction and move some random amount up to top speed
  */
 public void act()
 {
   // get the closest deer within smelling range
   DeerAgent closeDeer = (DeerAgent) getClosest(SMELL_RANGE,
                              DeerAgent.allDeer);
   if (closeDeer != null)
   {
     // Turn torward deer
     this.turnToFace(closeDeer);
     // How much to move?  How about minimum of maxSpeed
     // or distance to deer?
     this.forward((int) Math.min(speed,
                 closeDeer.getDistance(this.getXPos(),this.getYPos())));
   }

   // get the closest deer within the attack distance
   closeDeer = (DeerAgent) getClosest(ATTACK_RANGE,
                         DeerAgent.allDeer);

   if (closeDeer != null)
   {
     this.moveTo(closeDeer.getXPos(),
                closeDeer.getYPos());
     closeDeer.die();
   }

   else // Otherwise, wander aimlessly
   {

     super.act();
   } // end else
```

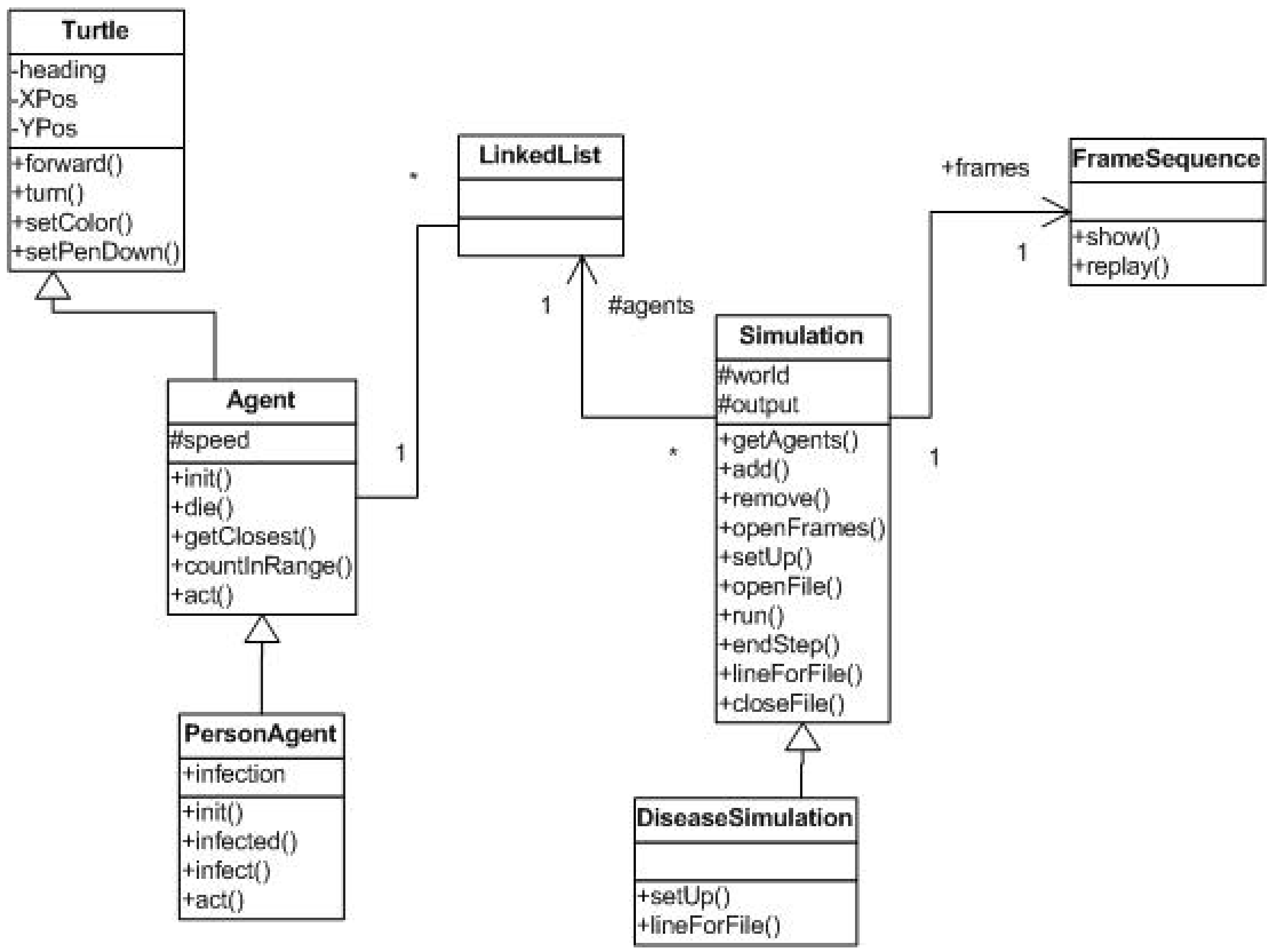
Running the WDSimulation

```
Welcome to DrJava.
> WDSimulation wd = new WDSimulation();
> wd.openFrames("D:/temp/"); // If you want an animation
> wd.openFile(“D:/cs1316/wds-data1.txt”); // If you want an output
    file.
> wd.run();
```

If you just want to run it:

```
> WDSimulation wd = new WDSimulation();
> wd.run();
```

DiseaseSimulation



What happens in a DiseaseSimulation

- We create a bunch of PersonAgents.
- One of them is sick.
- While running:
  - They wander aimlessly.
  - If a Person gets close (within 10? 20?) of an infected person, that Person gets infected, too.

DiseaseSimulation

setUp() just creates 60 people, and the first one becomes *infected.*

```
/**
 * DiseaseSimulation -- using the Simulation class
 **/
public class DiseaseSimulation extends Simulation {
/**
   * Fill the world with 60 persons, one sick
   **/
  public void setUp(){
    // Let the world be set up
    //super.setUp();
    // Or set it up with a smaller world
    world = new World(300,300);
    world.setAutoRepaint(false);

    PersonAgent moi;

    // 60 people
    for (int num = 0; num < 60; num++) {
      moi = new PersonAgent(world,this);
    }

    // Infect the first one
    moi = (PersonAgent) getAgents().get(0);
    moi.infect();
  }
```

Deciding what to store in a file

```
/**
 * lineForFile -- write out number of infected
 **/
public String lineForFile(){
  PersonAgent first;
  first = (PersonAgent) agents.get(0);
  return (new Integer(first.infected())).toString();
}
```

infected() is an instance method that returns the number of infected persons. It doesn't matter which person we ask it of, so we just grab the first one.

Defining a PersonAgent

```
import java.awt.Color; // Color for colorizing
import java.util.LinkedList;

/**
 * PersonAgent -- Person as a subclass of Agent
 **/
public class PersonAgent extends Agent {

  public boolean infection;
```

PersonAgent initialization

```
/**
  * Initialize, by setting color and making move fast
  **/
 public void init(Simulation thisSim){
   // Do the normal initializations
   super.init(thisSim);

   // Make it lightGray
   setColor(Color.lightGray);

   // Don't need to see the trail
   setPenDown(false);

   // Start out uninfected
   infection = false;

   // Make the speed large
   speed = 100;
 }
```

PersonAgent act()

```
/**
  * How a Person acts
  **/
 public void act()
 {
   // Is there a person within infection range of me?
   PersonAgent closePerson = (PersonAgent) getClosest(10,
                              simulation.getAgents());

   if (closePerson != null) {
     // If this person is infected, and I'm not infected
     if (closePerson.infection && !this.infection) {
       // I become infected
       this.infect();
   }
   }

   // Run the normal act() -- wander aimlessly
   super.act();
 }
```

Getting sick

```
/**
 * Become infected
 **/
public void infect(){
  this.infection = true;
  this.setColor(Color.red);

  // Print out count of number infected
  System.out.println("Number infected: "+infected());
}
```

Counting the infected

```
/**
 * Count infected
 **/
public int infected() {
  int count = 0;
  LinkedList agents = simulation.getAgents();
  PersonAgent check;

  for (int i = 0; i<agents.size(); i++){
    check = (PersonAgent) agents.get(i);
    if (check.infection) {count++;}
  }

  return count;
}
```

We could have added them to an infected list and just checked the size(), too.

There are constructors here, too, but we're ignoring them now.

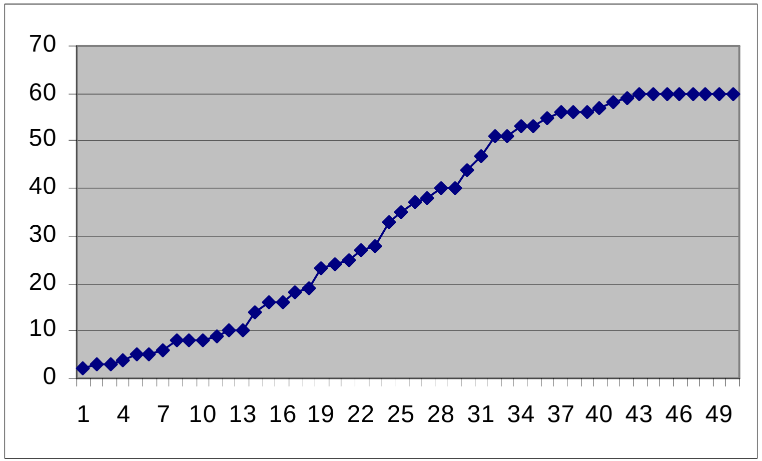
Running a DiseaseSimulation

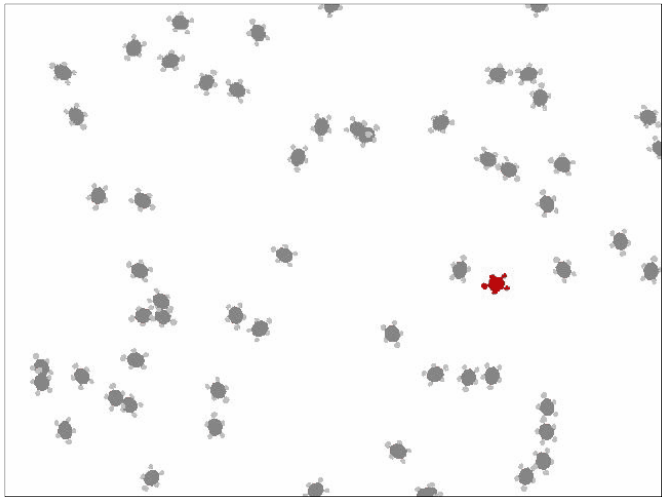
```
DiseaseSimulation ds2 = new
    DiseaseSimulation();
    ds2.openFile(“D:/cs1316/disease-fullsize.txt”);
    ds2.run();
```

Comparing Small and Large Worlds for Disease Propagation

A common activity in this class: Generate data for Excel and analyze it there.

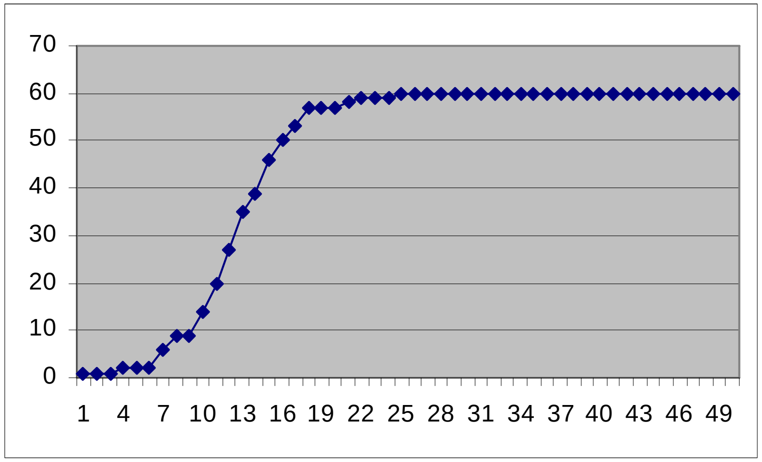
```
public void setUp(){
   // Let the world be set up
   super.setUp();
   // Or set it up with a smaller world
   //world = new World(300,300);
   //world.setAutoRepaint(false);
```

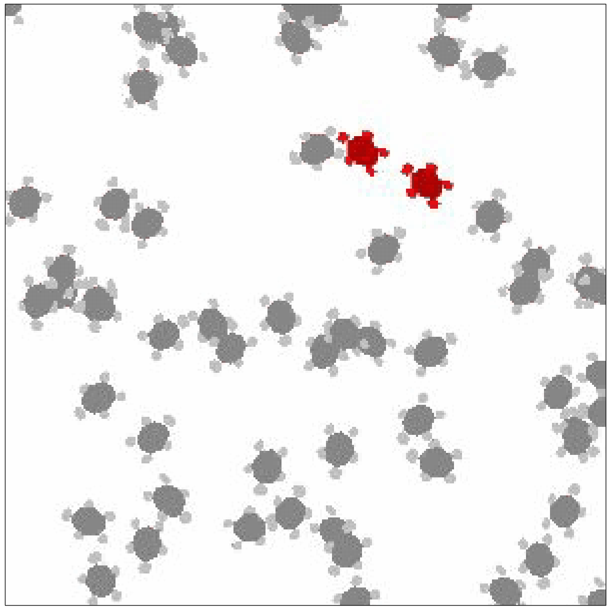




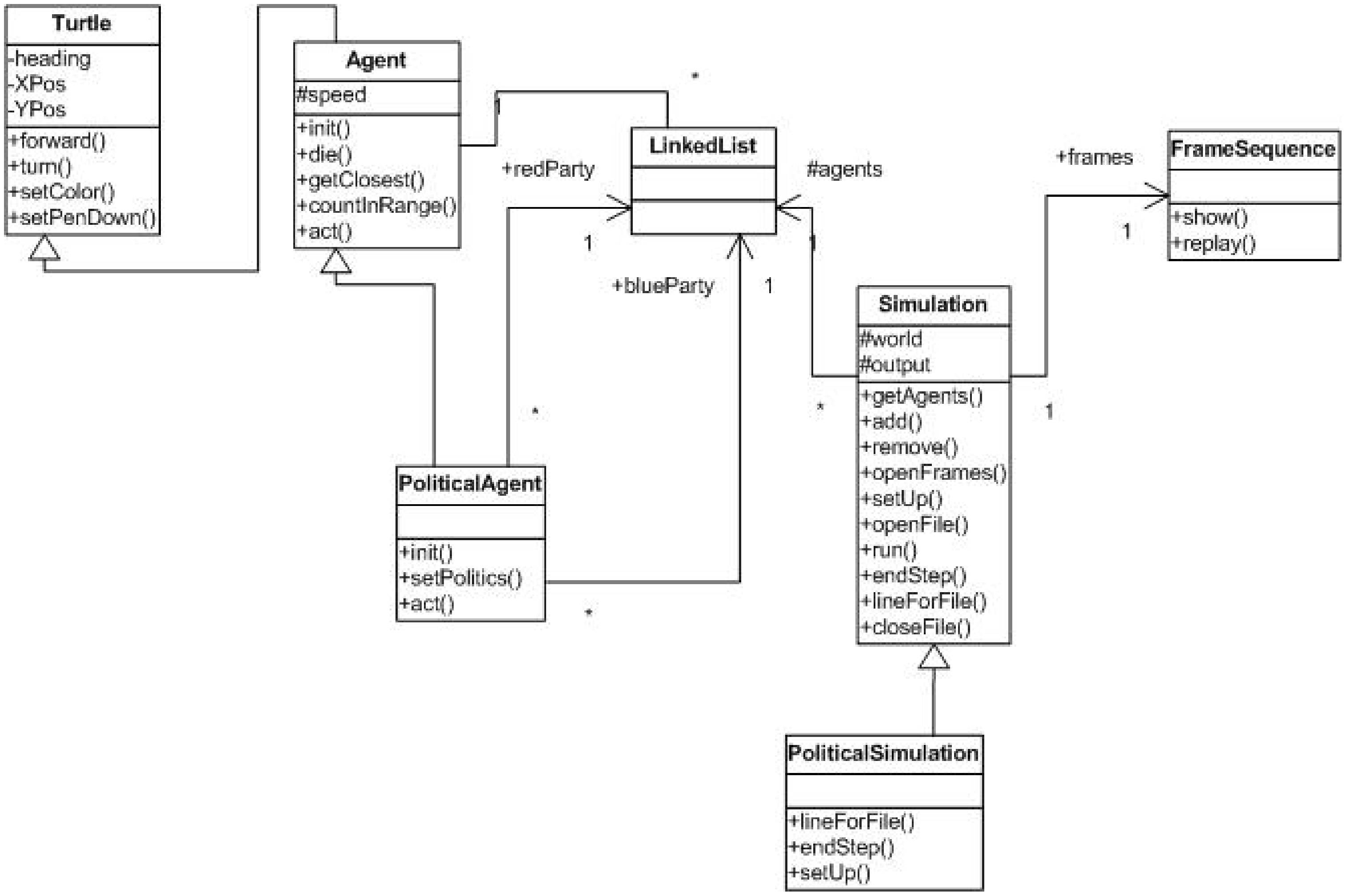
Small world DiseaseSimulation

```
public void setUp(){
  // Let the world be set up
  //super.setUp();
  // Or set it up with a smaller world
  world = new World(300,300);
  world.setAutoRepaint(false);
```





PoliticalSimulation



How it works

- There are two sets of PoliticalAgents: Red and Blue
- Both wander aimlessly, but within constraints.
  - Blue is only to the right, red only to the left.
  - Overlap for 200 pixels in the middle.
- If a Blue gets surrounded (argued down?) by more Red supporters than Blue supporters, the Blue turns Red. And vice-versa.

But there is a problem with getting converted mid-timestep!

Political Simulation

```
/**
 * PoliticalSimulation -- using the Simulation class
 **/
public class PoliticalSimulation extends Simulation {

/**
   * Fill the world with 60 persons
   **/
  public void setUp(){
    // Let the world be set up
    super.setUp();

    PoliticalAgent moi;

    // 60 people
    for (int num = 0; num < 60; num++) {
      moi = new PoliticalAgent(world,this);
      // First 30 are red
      if (num < 30) {
        moi.politics = Color.red;
        moi.moveTo(100,100);
        PoliticalAgent.redParty.add(moi);
      }
      else {
        moi.politics = Color.blue;
        moi.moveTo(500,100);
        PoliticalAgent.blueParty.add(moi);
      }
      moi.setColor(moi.politics);

    } // for loop

  } // setUp()
```

Tracking the PoliticalSimulation

```
/**
 * lineForFile -- write out number of each party
 **/
public String lineForFile(){

  return (new Integer(PoliticalAgent.redParty.size())).toString()+"\t"+
    (new Integer(PoliticalAgent.blueParty.size())).toString();
}

/**
 * EndStep, count the number of each
 **/
public void endStep(int t){
  super.endStep(t);

  System.out.println("Red: "+PoliticalAgent.redParty.size()+" Blue: "+
                    PoliticalAgent.blueParty.size());
}
```

We're accessing here the **static** variables that track the *redParty* and *blueParty*.

PoliticalAgent

```
import java.awt.Color; // Color for colorizing
import java.util.LinkedList;

/**
 * PoliticalAgent -- Red or Blue Stater as a subclass of Agent
 **/
public class PoliticalAgent extends Agent {

  // Red or Blue
  public Color politics;

  public static LinkedList redParty = new LinkedList();
  public static LinkedList blueParty = new LinkedList();
```

Initializing our PoliticalAgents

```
/**
  * Initialize
  **/
 public void init(Simulation thisSim){
   // Do the normal initializations
   super.init(thisSim);

   // Don't need to see the trail
   setPenDown(false);

   // Speed is 100
   speed = 100;

 }
```

Converting political preference

```
/**
  * Set politics
  **/
 public void setPolitics(Color pref){
   System.out.println("I am "+politics+" converting to "+pref);

   if (pref == Color.red) {
     blueParty.remove(this);
     redParty.add(this);
     this.politics = pref;}
   else {
     blueParty.add(this);
     redParty.remove(this);
     this.politics = pref;
   }
   this.setColor(pref);

 }
```

PoliticalAgent act()

```
/**
  * How a PoliticalAgent acts
  **/
 public void act()
 {
   // What are the number of blues and red near me?
   int numBlue = countInRange(30,blueParty);
   int numRed = countInRange(30,redParty);

   if (politics==Color.red){
     // If I'm red, and there are more blue than red near me, convert
     if (numBlue > numRed){
       setPolitics(Color.blue);}
   }
   if (politics==Color.blue){
     // If I'm blue, and there are more red than blue near me, convert
     if (numRed > numBlue) {
     setPolitics(Color.red);}
   }
```

PoliticalAgent act(), cont'd

```
// Run the normal act() -- wander aimlessly
    super.act();

    // But don't let them wander too far!
    // Let them mix only in the middle
    if (politics==Color.red) {
      if (this.getXPos() > 400) { // Did I go too far right?
        this.moveTo(200,this.getYPos());}
    }
     if (politics==Color.blue) {
      if (this.getXPos() < 200) { // Did I go too far left?
        this.moveTo(400,this.getYPos());}
     }

  }
```

How the Simulation Package works

- There are lots of calls to ***this***
  - this.setUp(), for example.
- This will call the *instance's* method.
  - Typically, the subclass!
- If the subclass doesn't have the method, it will ***inherit*** the method from the *superclass*.
- The subclass can still call the *superclass* version using ***super.***

Let’s trace the DiseaseSimulation

```
DiseaseSimulation ds2 = new DiseaseSimulation();
    ds2.run();
```

```
public Simulation() {
   // By default, don't write to a file.
   output = null;
   // And there is no FrameSequence
   frames = null;
 }
```

Here’s how a **Simulation** class instance gets constructed.

ds2.run();

```
/**
 * Run for a default of 50 steps
 **/
public void run(){
  this.run(50);
  this.closeFile();
}
```

Both methods are in **Simulation**

```
/**
 * Ask all agents to run for the number of input
 * steps
 **/
public void run(int timeRange)
{
  // A frame, if we're making an animation
  Picture frame;

  // For storing the current agent
  Agent current = null;

  // Set up the simulation
  this.setUp();
```

Does **ds2** have a **setUp()** method?

this.setUp() in DiseaseSimulation

```
/**
  * Fill the world with 60 persons, one sick
  **/
 public void setUp(){
  // Let the world be set up
  super.setUp();

  PersonAgent moi;

  // 60 people
  for (int num = 0; num < 60; num++) {
    moi = new PersonAgent(world,this);
  }

  // Infect the first one
  moi = (PersonAgent) getAgents().get(0);
  moi.infect();
 }
```

```
public void setUp(){
   // Set up the World
   world = new World();
   world.setAutoRepaint(false);
 }
```

Back to **Simulation** just for a moment, to set up the world, then back again.

Back to public void **run(int timeRange)** in **Simulation**

```
// Set up the simulation
this.setUp();

// loop for a set number of timesteps
for (int t = 0; t < timeRange; t++)
{
  // loop through all the agents, and have them
  // act()
  for (int index=0; index < agents.size(); index++) {
    current = (Agent) agents.get(index);
    current.act();
  }
```

Do **PersonAgents** have **act**()’s?

You bet!

PersonAgent act() to Agent act()

```
public void act()
 {
   // Is there a person within infection range of me?
   PersonAgent closePerson = (PersonAgent)
      getClosest(20,
                          simulation.getAgents());

   if (closePerson != null) {
     // If this person is infected, and I'm not infected
     if (closePerson.infection && !this.infection) {
       // I become infected
       this.infect();
   }
   }

   // Run the normal act() -- wander aimlessly
   super.act();
 }
```

```
 public void act()
  {
    // Default action: wander aimlessly
    // if the random number is > prob of
NOT turning then turn
    if (randNumGen.nextFloat() >
PROB_OF_STAY)
    {

this.turn(randNumGen.nextInt(360));
    }
    // go forward some random amount
forward(randNumGen.nextInt(speed));
  } // end act()
```

Finishing Simulation run()

```
// repaint the world to show the movement
    world.repaint();
    if (frames != null){
      // Make a frame from the world, then
      // add the frame to the sequence
      frame = new Picture(world.getWidth(),world.getHeight());
      world.drawOn(frame);
      frames.addFrame(frame);
    }

    // Do the end of step processing
    this.endStep(t);

    // Wait for one second
    //Thread.sleep(1000);
  }
```

Hang on! Not done yet!

endStep() calls lineForFile() (in DiseaseSimulation)

```
public void endStep(int t){
   // Let's figure out where we stand...
   System.out.println(">>> Timestep: "+t);

   // If we have an open file, write the counts to it
   if (output != null) {
     // Try it
     try{
         output.write(lineForFile());
       output.newLine();
     } catch (Exception ex) {
       System.out.println("Couldn't write the data!");
       System.out.println(ex.getMessage());
       // Make output null so that we don't keep trying
       output = null;
       }
     }
 }  // endStep()
```

```
 public String lineForFile(){
    PersonAgent first;
    first = (PersonAgent) agents.get(0);
    return (new
Integer(first.infected())).toString();
  }
```

How much do you have to know?

- Do you have to know what **Simulation run()** does? Or **Agent act()**?
  - Not in *any* detail!
  - You need to know what it roughly does, so you can decide if you need it via **super**
- You need to know when *your* code will be called, so that you can do what you need to, at the right point in the simulation.

Discrete Event Simulations

- Here’s where we use queues (stacks earlier, for reversing a list), inserting in order, and sorting.

Imagine the simulation…

- There are three Trucks that bring product from the Factory.
  - On average, they take 3 days to arrive.
  - Each truck brings somewhere between 10 and 20 products—all equally likely.
- We’ve got five Distributors who pick up product from the Factory with orders.
  - Usually they want from 5 to 25 products, all equally likely.
- It takes the Distributors an average of 2 days to get back to the market, and an average of 5 days to deliver the products.
- Question we might wonder: How much product gets sold like this?

Don’t use a Continuous Simulation

- We don’t want to wait that number of days in real time.
- We don’t even care about every day.
    - There will certainly be timesteps (days) when *nothing* happens of interest.
- We’re dealing with different probability distributions.
    - Some uniform, some normally distributed.
- Things can get out of synch
    - A Truck may go back to the factory and get more product before a Distributor gets back.
    - A Distributor may have to wait for multiple trucks to fulfill orders (and other Distributors might end up waiting in line)

We use a Discrete Event Simulation

- We don’t simulate every moment *continuously*.
- We simulate *discrete events*.

What’s the difference? No time loop

- In a discrete event simulation: There is no time loop.
  - There are events that are scheduled.
  - At each **run** step, the next scheduled event with the *lowest* time gets processed.
    - The current time is then *that* time, the time that that event is supposed to occur.
- Key: We have to keep the list of scheduled events *sorted* (in order)

What’s the difference? Agents don’t act()

- In a discrete event simulations, agents don’t act().
  - Instead, they wait for events to occur.
  - They schedule new events to correspond to the *next* thing that they’re going to do.
- Key: Events get scheduled according to different probabilities.

What’s the difference? Agents get blocked

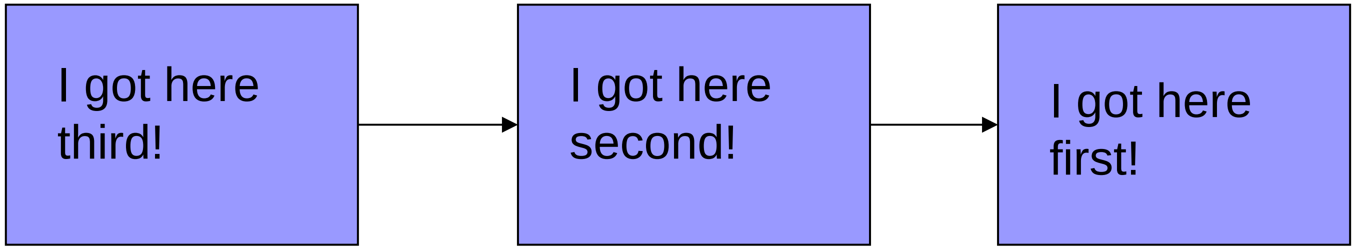
- Agents can’t do everything that they want to do.
- If they want product (for example) and there isn’t any, they get *blocked.*
  - They can’t schedule any new events until they get unblocked.
- Many agents may get blocked awaiting the same resource.
  - More than one Distributor may be awaiting arrival of Trucks
- Key: We have to keep track of the Distributors waiting *in line* (*in the* ***queue***)

Key Ideas

- A Queue
  - A Queue is a queue, no matter how implemented.
- Different kinds of random
- Straightening time
  - Inserting it into the right place
  - Sorting it afterwards

Key idea #1: Introducing a Queue

- First-In-First-Out List
  - First person in line is first person served



This is the *tail* of the queue

This is the *front* or *head* of the queue

Key idea #2:
Different kinds of random

- We've been dealing with *uniform* random distributions up until now, but those are the *least* likely random distribution in real life.
- How can we generate some other distributions, including some that are more realistic?

Key idea #3: Straightening Time

- Straightening time
  - Inserting it into the right place
  - Sorting it afterwards
- We’ll actually do these in reverse order:
  - We’ll add a new event, then sort it.
  - Then we’ll insert it into the right place.

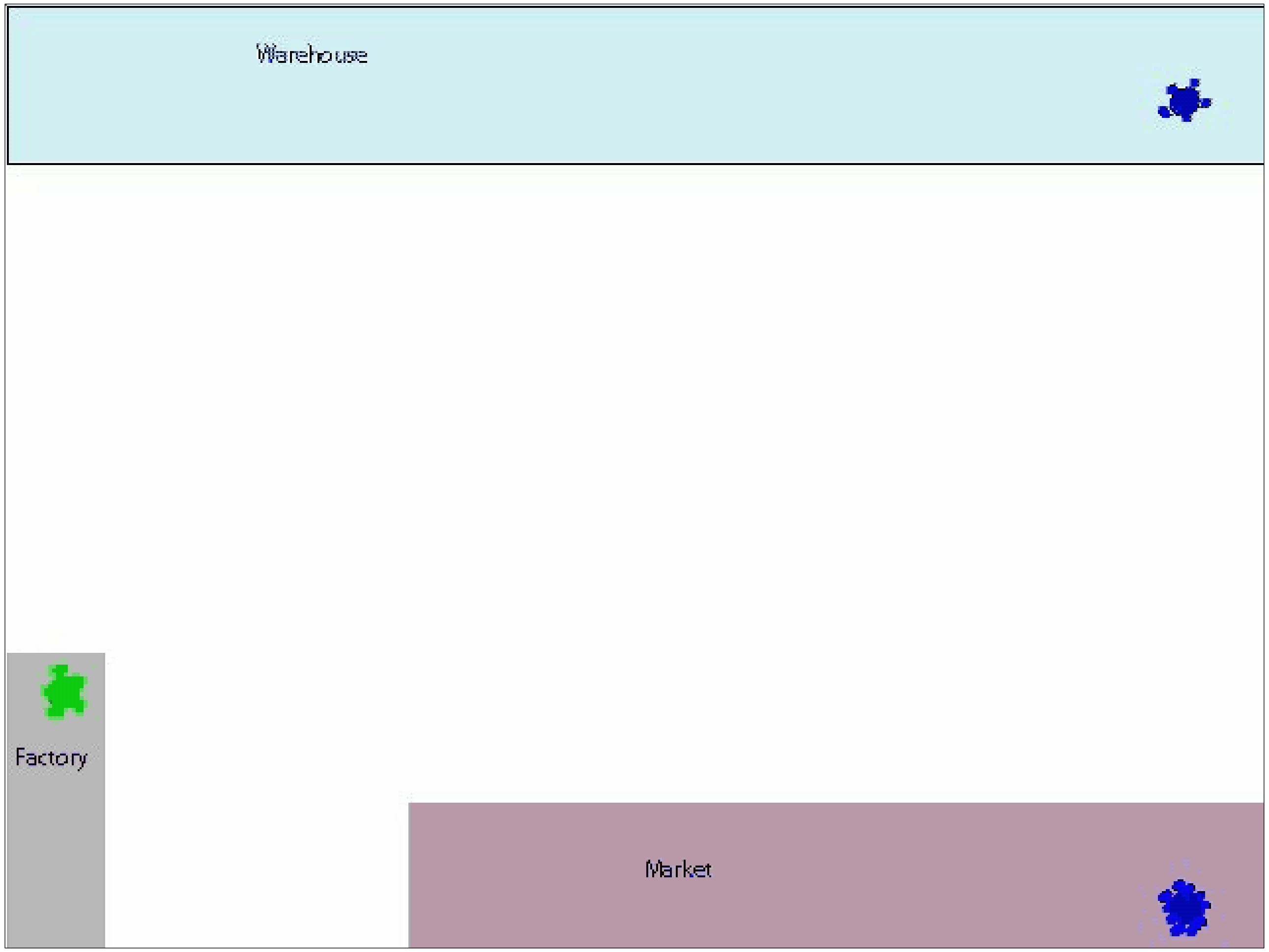
Finally: A Discrete Event Simulation

- Now, we can assemble queues, different kinds of random, and a sorted EventQueue to create a discrete event simulation.

Running a DESimulation

```
Welcome to DrJava.
> FactorySimulation fs = new FactorySimulation();
> fs.openFrames("D:/temp/");
> fs.run(25.0)
```

What we see (not much)



The detail tells the story

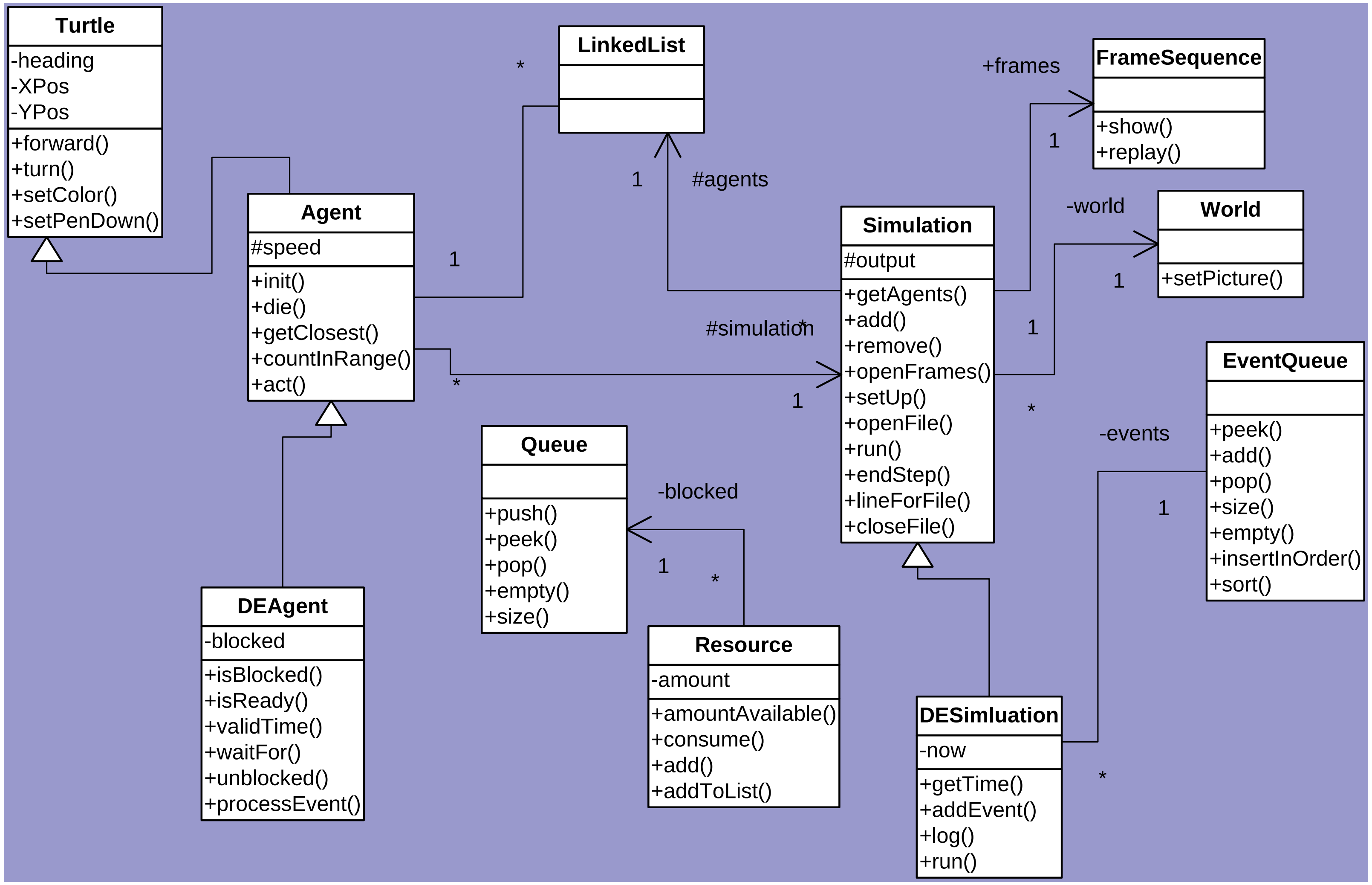
```
Time:      1.7078547183397625   Distributor: 0            Arrived at warehouse
Time:      1.7078547183397625   Distributor: 0            is blocking
>>> Timestep: 1
Time:      1.727166341118611    Distributor: 3            Arrived at warehouse
Time:      1.727166341118611    Distributor: 3            is blocking
>>> Timestep: 1
Time:      1.8778754913001443   Distributor: 4            Arrived at warehouse
Time:      1.8778754913001443   Distributor: 4            is blocking
>>> Timestep: 1
Time:      1.889475045031698    Distributor: 2            Arrived at warehouse
Time:      1.889475045031698    Distributor: 2            is blocking
>>> Timestep: 1
Time:      3.064560375192933    Distributor: 1            Arrived at warehouse
Time:      3.064560375192933    Distributor: 1            is blocking
>>> Timestep: 3
Time:      3.444420374970288    Truck: 2    Arrived at warehouse with load     13
Time:      3.444420374970288    Distributor: 0    unblocked!
Time:      3.444420374970288    Distributor: 0            Gathered product for orders of     11
>>> Timestep: 3
Time:      3.8869697922832698   Truck: 0    Arrived at warehouse with load     18
Time:      3.8869697922832698   Distributor: 3    unblocked!
Time:      3.8869697922832698   Distributor: 3            Gathered product for orders of     12
>>> Timestep: 3
Time:      4.095930381479024    Distributor: 0            Arrived at market
>>> Timestep: 4
Time:      4.572840072576855    Truck: 1    Arrived at warehouse with load     20
Time:      4.572840072576855    Distributor: 4     unblocked!
Time:      4.572840072576855    Distributor: 4            Gathered product for orders of     19
```

Notice that time 2 never occurs!

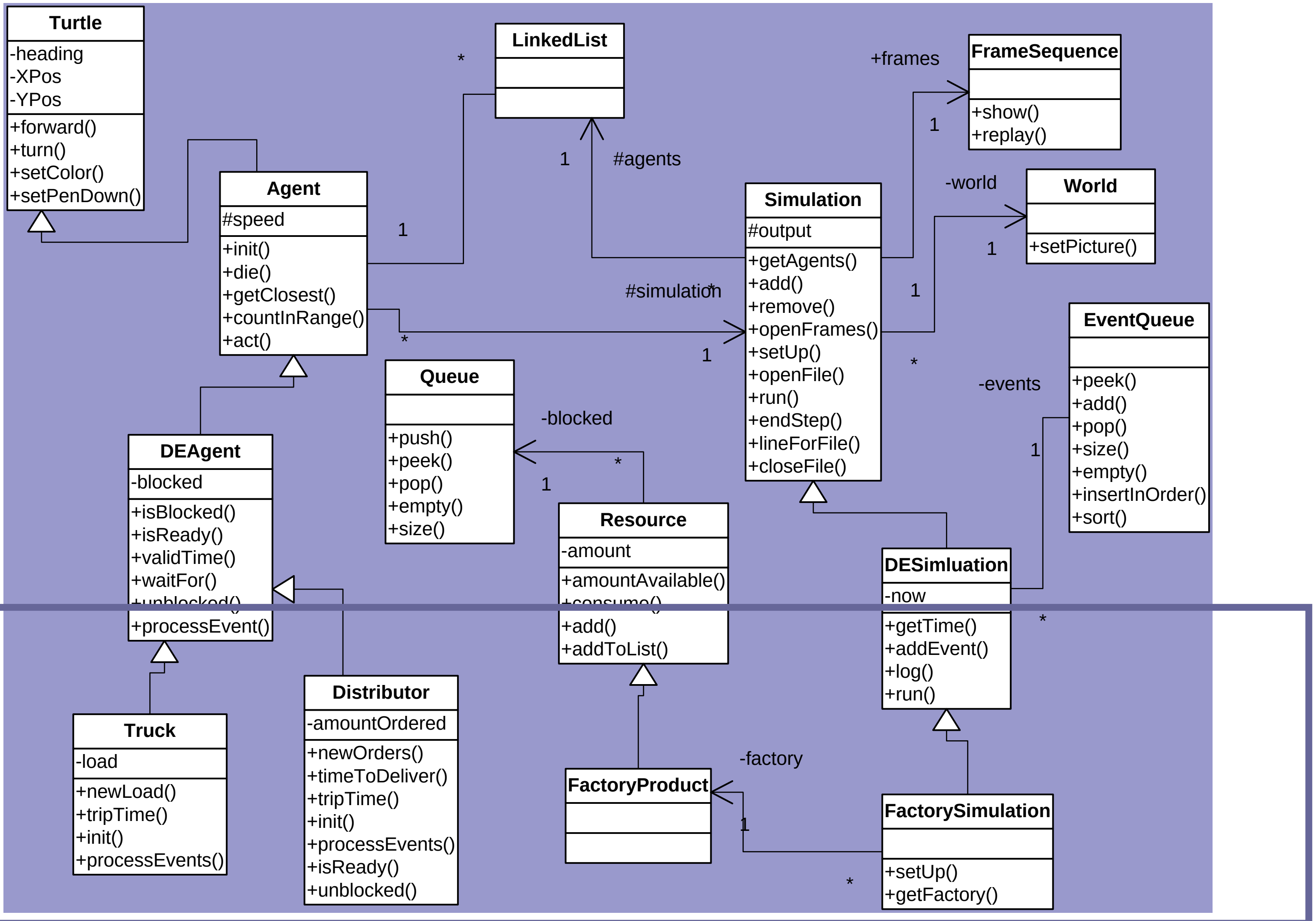
What questions we can answer

- How long do distributors wait?
  - Subtract the time that they unblock from the time that they block
- How much product sits in the warehouse?
  - At each time a distributor leaves, figure out how much is left in the warehouse.
- How long does the line get at the warehouse?
  - At each block, count the size of the queue.
- Can we move more product by having more distributors or more trucks?
  - Try it!

How DESimulation works



FactorySimulation: Extend a few classes



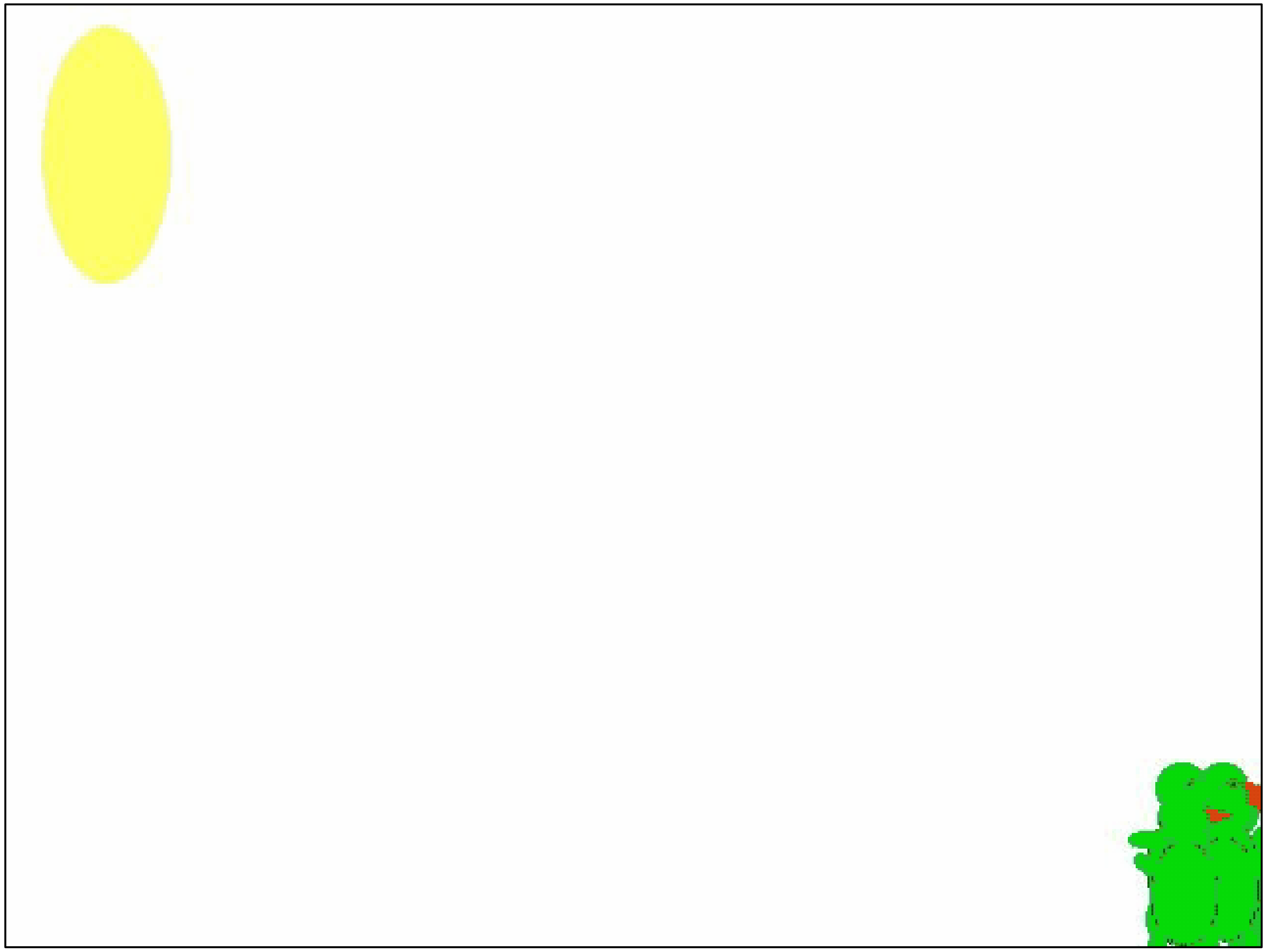
Finally! Making Wildebeests

- We simply copy characters to frames wherever the turtles are.

Story: The Curious Birds

- The turtle-like curious bird things wander, slowly, toward the mysterious egg.
- As they get up close to it—it opens its eyes and shows its fangs!
- They scamper away while the monster shifts around and looks to the left and right.

The Movie



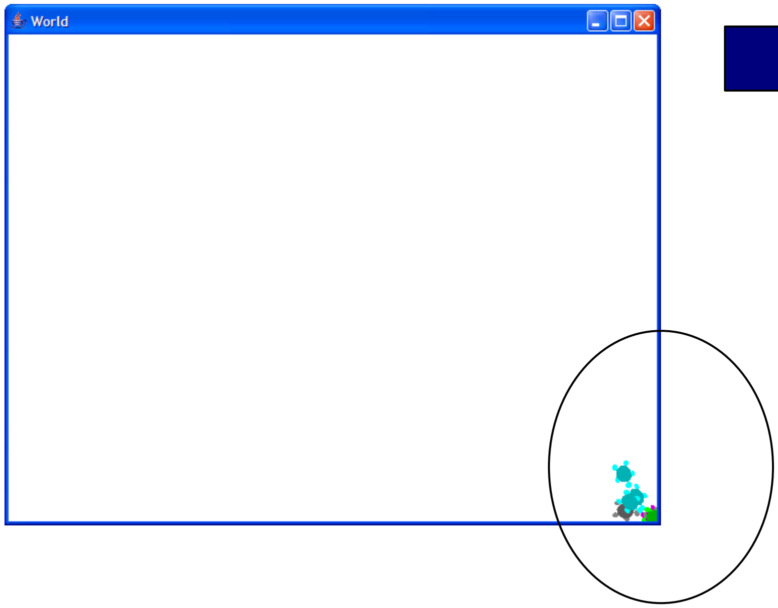
“What’s the big deal?”

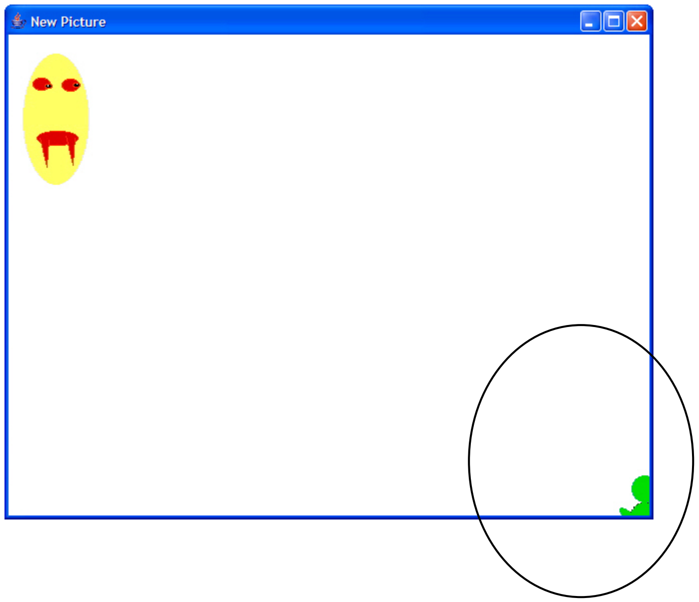
- “Isn’t this *darn* similar to the wolves attacking the village movie?”
- Yes.
- But we didn’t have to build a scene graph and define every frame here.
- We simply built a simulation and said “Go.”
  - Each time that we run this simulation, it’ll be slightly different.
  - We can have as many “takes” as we want, and tweak the rules of behavior as we want.

A Mapping

We move Agents (turtles) in a simulation

And once a timestep, we map these to images and draw them.





BirdSimulation

```
/**
 * BirdSimulation
 * A flock of 10 birds investigate a mysterious egg,
 * which suddenly shows itself to be a monster!
 **/
public class BirdSimulation extends Simulation {

  public EggAgent egg; // We'll need to get this later in BirdAgent
  FrameSequence myFrames; // Need a separate one from Simulations
```

Setting up the Simulation

```
/**
  * Set up the world with 10 birds and the mysterious egg
  **/
 public void setUp(){
   // Set up the world
   super.setUp();
   // We'll need frames for the animation
   myFrames = new FrameSequence("D:/Temp/");
   myFrames.show();

   BirdAgent tweetie;
   // 10 of 'em
   for (int num = 0; num < 10; num++) {
     tweetie = new BirdAgent(world,this);}

   // And the egg
   egg = new EggAgent(world,this);
 }
```

Creating the Animation

```
public void endStep(int t) {
   // Do the normal file processing (if any)
   super.endStep(t);

   // But now, make a 640x480 frame, and copy
   // in pictures from all the agents
   Picture frame = new Picture(640,480);
   Agent drawMe = null;
   for (int index=0; index<this.getAgents().size(); index++) {
     drawMe = (Agent) this.getAgents().get(index);
     drawMe.myPict.bluescreen(frame,drawMe.getXPos(),
                              drawMe.getYPos());
   }
   myFrames.addFrame(frame);
 }
```

Explaining the key lines

- We get our Agent (BirdAgent or EggAgent).
- Get the picture *myPict* then bluescreen it onto the frame at the current position of the agent's turtle.

```
drawMe = (Agent)
        this.getAgents().get(index);

drawMe.myPict.bluescreen(frame,
          drawMe.getXPos(),
          drawMe.getYPos());
```

Getting the timestep

- Since we want something to happen at certain timesteps, we need **act**() to have the timestep.
  - We need **Simulation** to pass us the timestep.
  - We need **Agent**’s **act()** to catch the timestep and pass it on as *no* timestep, so that all the old simulations continue to work.

Simulation change

```
// loop through all the agents, and have them
    // act()
    for (int index=0; index < agents.size(); index++) {
      current = (Agent) agents.get(index);
      current.act(t); // NEW -- pass in timestep
    }
```

Addition to Agent

```
/**
 * act() with a timestep
 **/
public void act(int t){
  // By default, don't act on it
  this.act();
}
```

Think through why we need this.

Simulation is now calling **act(timestep).** Our other simulations don't have **act(int t)**…

BirdAgent

```
/**
 * BirdAgents use the bird character JPEGs
 **/
public class BirdAgent extends Agent{

  public static Picture bird1, bird2, bird3, bird4, bird5, bird6;
```

Why **static**? Would it work without **static**? Yes, but more resource intensive.

Setting up birds

```
/**
  * Set up the birds
  **/
 public void init(Simulation thisSim){
   if (bird1 == null) {
     // Do we have the bird characters defined yet?
     // CHANGE ME!
     FileChooser.setMediaPath("D:/cs1316/MediaSources/");
     bird1 = new Picture(FileChooser.getMediaPath("bird1.jpg"));
     bird2 = new Picture(FileChooser.getMediaPath("bird2.jpg"));
     bird3 = new Picture(FileChooser.getMediaPath("bird3.jpg"));
     bird4 = new Picture(FileChooser.getMediaPath("bird4.jpg"));
     bird5 = new Picture(FileChooser.getMediaPath("bird5.jpg"));
     bird6 = new Picture(FileChooser.getMediaPath("bird6.jpg"));
  }
```

Setting up a bunch of static variables to hold the bird pictures

Finishing BirdAgent init()

```
// Start out with myPict as bird1
   myPict = bird1;

   // Do the normal initializations
   super.init(thisSim);

   // Move all the birds to the far right corner
   this.setPenDown(false);
   this.moveTo(600,400);

   // Set speed to relatively slow
   this.setSpeed(40);

 }
```

What Birds do

```
/**
 * act(t) For first 20 steps, walk toward the egg,
 * +/- 30 degrees.
 * Then walk AWAY from the egg, and with MORE wandering (panic).
 **/
public void act(int t){
  // First, handle motion
  if (t <= 20) {
    // Tell it that this really is a BirdSimulation
    BirdSimulation mySim = (BirdSimulation) simulation;
    // which has an egg
    this.turnToFace(mySim.egg);
    this.turn(randNumGen.nextInt(60)-30);
    forward(randNumGen.nextInt(speed));
  } else {
    // Run away!!
    this.turnToFace(640,480); // Far right corner
    this.turn(randNumGen.nextInt(80)-40);
    forward(randNumGen.nextInt(speed));
  }
```

What's going on with this math is getting +/-. 0 to 60, minus 30, gives you -30 to 30

Birds also change character look (cell animation)

```
// Next, set a new character
   int cell = randNumGen.nextInt(6)+1; // 0 to 5, + 1 => 1 to 6
   switch (cell) {
     case 1:
       myPict = bird1; // this.drop(bird1);
       break;
     case 2:
       myPict = bird2; //this.drop(bird2);
       break;
     case 3:
       myPict = bird3; //this.drop(bird3);
       break;
     case 4:
       myPict = bird4; //this.drop(bird4);
       break;
     case 5:
       myPict = bird5; //this.drop(bird5);
       break;
     case 6:
       myPict = bird6; //this.drop(bird6);
       break;
   } // end switch
 } // end act
```

What's this?

It's called a *switch* or *case* statement.

Instead of 6 *if*'s, we have one big *case*.

Consider drop vs. chromakey? Think about resources and mapping to other media

Zooming in on the *switch*

- We compute a random integer between 1 and 6.
- We announce that we're going to choose between options (*switch*) depending on the value of *cell*.
- We identify each value we're checking with a *case* statement.
- Java executes the one that matches the switch.
- Execution ends and jumps to the end of the *switch* on *break*.

```
int cell =
      randNumGen.nextInt(6)+1;
      // 0 to 5, + 1 => 1 to 6
   switch (cell) {
     case 1:
       myPict = bird1;
       break;
     case 2:
       myPict = bird2;
       break;
```

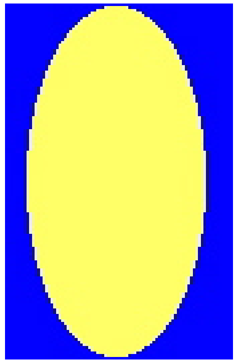
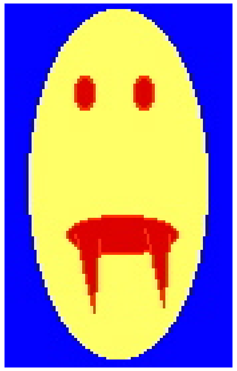
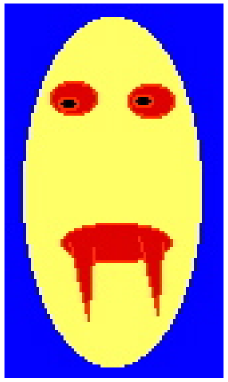
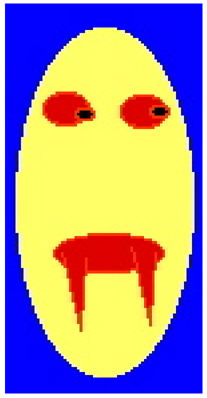
EggAgent

```
/**
 * EggAgent -- big scary egg that sits there until t=15,
 * then emerges as a monster!
 **/
public class EggAgent extends Agent {

  public static Picture egg1, egg2, egg3, egg4;
```

Init() for an EggAgent

```
/**
  * To initialize, set it up as the Egg in the upper lefthand corner
  **/
 public void init(Simulation thisSim){
   if (egg1 == null) { //Initialize
     //CHANGE ME!
     FileChooser.setMediaPath("D:/cs1316/MediaSources/");
     egg1 = new Picture(FileChooser.getMediaPath("egg1.jpg"));
     egg2 = new Picture(FileChooser.getMediaPath("egg2.jpg"));
     egg3 = new Picture(FileChooser.getMediaPath("egg3.jpg"));
     egg4 = new Picture(FileChooser.getMediaPath("egg4.jpg"));
   }
   // Start out as egg1
   myPict = egg1;
```

Meet the Eggs

The rest of EggAgent init()

```
  // Normal initialization
  super.init(thisSim);

  // Make the turtle disappear
  //this.hide(); // Not really necessary
  this.setPenDown(false);

  // Move the egg up to the left hand corner
  this.moveTo(10,10);
}
```

Eggs don’t move in act(int t)

```
/**
  * To act, just drop the Egg for 15 steps,
  * then be the eyes opened for five steps,
  * then be the eyes switching back-and-forth
  **/
 public void act(int t) {
   if (t < 19) {
     myPict = egg1;}
     //this.drop(egg1);}

   if (t>19 && t<24) {
     myPict = egg2;}
     //this.drop(egg2);}
```

Even when they're looking scary

```
if (t>23) {
     int choose=randNumGen.nextInt(2);
     if (choose == 1) {
       myPict = egg3;}
     //this.drop(egg3);}
    else {
      myPict = egg4;}
     //this.drop(egg4);}
   }
 } // end act()
```

To Explore

- Start the birds out all over the screen
  - So that we can see them wave, etc. better.
- Have the birds react to a *state* of the egg
  - For example: “egg.scary()==true”, so that we’re not tied to a particular timestep (frame #)
- Have the birds react to one another.
  - “I’m crowded, I’m going to move that way.”
- ***Big idea!*** An animation is only *one* kind of representation to make from a simulation.
  - How about playing certain sounds or MIDI phrases from different characters at different times?