Structuring Images

CS1316: Representing Structure and Behavior

Story

- Structuring images into scenes
  - Version 1: Representing linearity through elements order.
  - Animation through rendering and data structure tweaking
  - Version 2: Representing layering through order.
  - Version 3: Allowing both in a single list
    - Introducing subclasses and superclasses
    - Including abstract classes
    - Passing a turtle along for processing.
  - Version 4: Creating trees of images
    - Making the branches do something

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.

Use an array?

```java
Picture[] myarray = new Picture[5];
myarray[0]=new Picture(FileChooser.getMediaPath("katie.jpg"));
myarray[1]=new Picture(FileChooser.getMediaPath("barbara.jpg"));
myarray[2]=new Picture(FileChooser.getMediaPath("tower1.jpg"));
myarray[3]=new Picture(FileChooser.getMediaPath("flower2.jpg"));
myarray[4]=new Picture(FileChooser.getMediaPath("butterfly.jpg"));
Picture background = new Picture(400,400);
for (int i = 0; i < 5; i++)
{
    myarray[i].scale(0.5).compose(background,i*10,i*10);
}
background.show();
```

Yeah, we could. But:
- Inflexible
- Hard to insert, delete.

Using a linked list

- Okay, so we'll use a linked list.
- But what should the ordering represent?
  - Version 1: Linearity
    - The order that things get drawn left-to-right.
  - Version 2: Layering
    - The order that things get drawn bottom-to-top

Version 1: PositionedSceneElement

```java
> PositionedSceneElement tree1 = new PositionedSceneElement(new Picture(FileChooser.getMediaPath("tree-blue.jpg")));
> PositionedSceneElement tree2 = new PositionedSceneElement(new Picture(FileChooser.getMediaPath("dog-blue.jpg")));
> PositionedSceneElement tree3 = new PositionedSceneElement(new Picture(FileChooser.getMediaPath("house-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();
```
What this looks like:

Slightly different ordering:
Put the doggy between tree2 and tree3

```java
> 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.

PositionedSceneElement

```java
public class PositionedSceneElement {
    /**
    * the picture that this element holds
    */
    private Picture myPic;

    /**
    * the next element in the list
    */
    private PositionedSceneElement next;
}
```

Pretty darn similar to our music linked lists!

Constructor

```java
/**
 * Make a new element with a picture as input, and
 * next as null.
 * @param heldPic Picture for element to hold
 **/
public PositionedSceneElement(Picture heldPic){
    myPic = heldPic;
    next = null;
}
```

Pretty darn similar!

Linked list methods

```java
/**
 * Methods to set and get next elements
 * @param nextOne next element in list
 **/
public void setNext(PositionedSceneElement nextOne){
    this.next = nextOne;
}
```

Again, darn similar!

```java
public PositionedSceneElement getNext(){
    return this.next;
}
```
Traverse the list

Traversing the list in order to draw the scene is called rendering the scene: Realizing the picture described by the data structure.

```java
public void drawFromMeOn(Picture bg) {
    PositionedSceneElement current;
    int currentX = 0, currentY = bg.getHeight()-1;
    current = this;
    while (current != null) {
        current.drawMeOn(bg, currentX, currentY);
        currentX = currentX + current.getPicture().getWidth();
        current = current.getNext();
    }
}
```

Drawing the individual element

```java
private void drawMeOn(Picture bg, int left, int bottom) {
    // Bluescreen takes an upper left corner
    this.getPicture().bluescreen(bg, left, bottom-this.getPicture().getHeight());
}
```

Generalizing

- Reconsider these lines:

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

- This is actually a general case of:
  - Removing the doggy from the list
  - Inserting it after tree2

Removing the doggy

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

Putting the mutt back

```java
> bg = new Picture(chooser.selectImage("jungle.jpg"));
> tree1.insertAfter(doggy);
> tree1.drawFromMeOn(bg);
```
Removing an element from the list

**Method to remove node from list, fixing links appropriately.**

- @param node element to remove from list.

```java
public void remove(PositionedSceneElement node) {
    if (node == this) {
        System.out.println("I can't remove the first node from the list.");
        return;
    }
    PositionedSceneElement current = this;
    while (current.getNext() != null) {
        if (current.getNext() == node) {
            // Simply make node's next be this next
            current.setNext(node.getNext());
            // Make this node point to nothing
            node.setNext(null);
            return;
        }
        current = current.getNext();
    }
}
```

Note: How would you remove the first element from the list?

Error checking and printing

**Method to remove node from list, fixing links appropriately.**

- @param node element to remove from list.

```java
public void remove(PositionedSceneElement node) {
    if (node == this) {
        System.out.println("I can't remove the first node from the list.");
        return;
    }
    PositionedSceneElement current = this;
    while (current.getNext() != null) {
        if (current.getNext() == node) {
            // Simply make node's next be this next
            current.setNext(node.getNext());
            // Make this node point to nothing
            node.setNext(null);
            return;
        }
        current = current.getNext();
    }
}
```

The Removal Loop

```java
PositionedSceneElement current = this;
while (current.getNext() != null) {
    if (current.getNext() == node) {
        // Simply make node's next be this next
        current.setNext(node.getNext());
        // Make this node point to nothing
        node.setNext(null);
        return;
    }
    current = current.getNext();
}
```

We're checking `getNext()` because we need to stop the loop before.

Error checking and printing

```java
public void remove(PositionedSceneElement node) {
    if (node == this) {
        System.out.println("I can't remove the first node from the list.");
        return;
    }
    PositionedSceneElement current = this;
    while (current.getNext() != null) {
        if (current.getNext() == node) {
            // Simply make node's next be this next
            current.setNext(node.getNext());
            // Make this node point to nothing
            node.setNext(null);
            return;
        }
        current = current.getNext();
    }
}
```

insertAfter

```java
public void insertAfter(PositionedSceneElement node) {
    // Save what "this" currently points at
    PositionedSceneElement oldNext = this.getNext();
    this.setNext(node);
    node.setNext(oldNext);
}
```

Think about what's involved in creating `insertBefore()`.

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

```java
public class AnimatedPositionedScene {
    /**
     * A FrameSequence for storing the frames
     * @param frames FrameSequence frames;
     * @param scene an AnimatedPositionedScene
     * @param tree1, tree2, tree3, house, doggy, doggyflip;
    */
    public AnimatedPositionedScene {
    }
}
```
Setting up the animation

```java
public void setUp()
{
    frames = new FrameSequence("D:/Temp/!
    p = new Picture(FileChooser.isDirectory("tree-blue.jpg"));
    tree1 = new PositionedSceneElement(p);
    p = new Picture(FileChooser.isDirectory("tree-blue.jpg"));
    tree2 = new PositionedSceneElement(p);
    p = new Picture(FileChooser.isDirectory("tree-blue.jpg"));
    tree3 = new PositionedSceneElement(p);
    p = new Picture(FileChooser.isDirectory("house-blue.jpg"));
    house = new PositionedSceneElement(p);
    p = new Picture(FileChooser.isDirectory("dog-blue.jpg"));
    doggy = new PositionedSceneElement(p);
    doggyflip = new PositionedSceneElement(p.flip());
}
```

Render the first frame

```java
public void make()
{
    frames.show();
    // First frame
    Picture bg = new Picture(FileChooser.isDirectory("jungle.jpg"));
    tree3.setNext(doggy); doggy.setNext(tree2);
    tree2.setNext(tree3);
    tree1.drawFromMeOn(bg);
    frames.addFrame(bg);
}

Moving left

```java
bg = new Picture(FileChooser.isDirectory("jungle.jpg"));
    tree1.remove(doggyflip);
    house.insertAfter(doggyflip);
    tree1.drawFromMeOn(bg);
    frames.addFrame(bg);
}
```

Version 2: Layering

```java
> Picture bg = new Picture(400,400);
> LayeredSceneElement tree1 = new LayeredSceneElement
    new Picture(FileChooser.isDirectory("tree-blue.jpg")),10,10);
> LayeredSceneElement tree2 = new LayeredSceneElement
    new Picture(FileChooser.isDirectory("tree-blue.jpg")),100,10);
> LayeredSceneElement tree3 = new LayeredSceneElement
    new Picture(FileChooser.isDirectory("house-blue.jpg")),175,175);
> LayeredSceneElement doggy = new LayeredSceneElement
    new Picture(FileChooser.isDirectory("dog-blue.jpg")),150,325);
> bg.show();
```
First version of Layered Scene

Reordered (relayered) scene

LayeredSceneElement

Constructor

```java
public class LayeredSceneElement {
  /**
   * the picture that this element holds
   */
  private final Picture myPic;
  /**
   * the next element in the list
   */
  private final LayeredSceneElement next;
  /**
   * The coordinates for this element
   */
  private final int x, y;

  public LayeredSceneElement(Picture heldPic, int xpos, int ypos){
    myPic = heldPic;
    next = null;
    x = xpos;
    y = ypos;
  }
}
```
**Linked List methods (We can sort of assume these now, right?)**

```java
/**
 * Methods to set and get next elements
 * @param nextOne next element in list
 */
public void setNext(LayeredSceneElement nextOne)
{
    this.next = nextOne;
}

public LayeredSceneElement getNext()
{
    return this.next;
}
```

**Traversing**

```java
/**
 * Method to draw from this node on in the list, using bluescreen.
 * Each new element has it's lower-left corner at the lower-right of
 * the previous node. Starts drawing from left-bottom
 * @param bg Picture to draw drawing on
 */
public void drawFromMeOn(Picture bg)
{
    LayeredSceneElement current;
    current = this;
    while (current != null)
    {
        current.drawMeOn(bg);
        current = current.getNext();
    }
}

/**
 * Method to draw from this picture, using bluescreen.
 * @param bg Picture to draw drawing on
 */
private void drawMeOn(Picture bg)
{
    this.getPicture().bluescreen(bg,x,y);
}
```

**Linked list traversals are all the same**

```java
current = this;
while (current != null)
{
    current.drawMeOn(bg);
    current = current.getNext();
}
```

**Doing a reverse()**

```java
/**
 * Reverse the list starting at this,
 * and return the last element of the list.
 * The last element becomes the FIRST element
 * of the list, and THIS points to null.
 */
public LayeredSceneElement reverse()
{
    LayeredSceneElement reversed, temp;
    // Handle the first node outside the loop
    reversed = this.last();
    this.remove(reversed);
    while (this.getNext() != null)
    {
        temp = this.last();
        this.remove(temp);
        reversed.add(temp);
    }
    // Now put the head of the old list on the end of
    // the reversed list.
    reversed.add(this);
    // At this point, reversed
    // is the head of the list
    return reversed;
}
```

**Getting the last()**

```java
/**
 * Return the last element in the list
 */
public LayeredSceneElement last()
{
    LayeredSceneElement current;
    current = this;
    while (current.getNext() != null)
    {
        current = current.getNext();
    }
    return current;
}
```

**Adding to the end**

```java
/**
 * Add the input node after the last node in this list.
 * @param node element to insert after this.
 */
public void add(LayeredSceneElement node)
{
    this.last().insertAfter(node);
}
```

Pretty easy, huh? Find the last(), and insertAfter()
Does it work?

```java
> 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")),10,10);
> LayeredSceneElement house = new LayeredSceneElement(
new Picture(FileChooser.getMediaPath("house-blue.jpg")),10,10);
> tree1.setNext(tree2); tree2.setNext(house);
> LayeredSceneElement rev = tree1.reverse();
> rev.drawFromMeOn(bg);
bg.show();
```

Let's add this up then...

```java
while (this.getNext() != null) {
    temp = this.last();
    this.remove(temp);
    reversed.add(temp);
}
```

Total cost: For each of the \( n \) nodes, reversing takes two traversals (2\( n \))
\( \Rightarrow O(n^2 \cdot 2n) \Rightarrow O(n^3) \)

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`
  - `SceneElementLayered`
  - We’d actually use these.

Class Structure

- An abstract class defines structure and behavior that subclasses will inherit.
- The subclasses inherit data and methods from superclass.
- We say that the subclasses extend the superclass.

```java
class SceneElement {
    private Picture myPic;
    private SceneElement next;

    public SceneElement() {
        this.myPic = new Picture();
    }

    public void setNext(SceneElement next) {
        this.next = next;
    }

    public SceneElement getNext() {
        return this.next;
    }

    public void drawWithTurtle(Turtle turtle) {
        turtle.draw(myPic);
    }
}
```

```
class SceneElementPositioned extends SceneElement {
    private Point position;

    public SceneElementPositioned() {
        super();
        this.position = new Point();
    }

    public void setPosition(Point position) {
        this.position = position;
    }

    public Point getPosition() {
        return this.position;
    }

    public void drawWithTurtle(Turtle turtle) {
        turtle.setPenColor(Turtle.PenColor.BLUE);
        turtle.penUp();
        turtle.moveTo(position);
        turtle.penDown();
        turtle.draw(myPic);
        turtle.penUp();
    }
}
```
Using the new structure

```java
public class MultiElementScene {
    public static void main(String[] args) {
        // 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));

        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();
    }
}
```

Rendering the scene

```java
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();
```

SceneElement

```java
/**
 * An element that knows how to draw itself in a scene with a turtle
 * @param nextOne next element in list
 */
public abstract class SceneElement{
    /**
     * the picture that this element holds
     */
    protected Picture myPic;
    /**
     * the next element in the list -- any SceneElement
     */
    protected SceneElement next;

    public void setNext(SceneElement nextOne){
        this.next = nextOne;
    }

    public SceneElement getNext(){
        return this.next;
    }
}
```

Linked List methods in SceneList

```java
/**
 * Methods to set and get next elements
 * @param nextOne next element in list
 */
public void setNext(SceneElement nextOne){
    this.next = nextOne;
}
```

drawFromMeOn()

```java
public void drawFromMeOn(Picture bg){
    SceneElement current;
    if (current == this) {
        // Position the turtle for the next positioned element
        current = current.getNext();
        pen.setPenDown(false); // Pick the pen up
    }
    while (current != null) {
        int currentX = bg.getWidth();
        int currentY = bg.getHeight();
        Turtle pen = new Turtle(bg);
        current.drawWith(pen); pen.setPenDown(true); // Pick the pen down
        current = current.next;
    }
}
```
But SceneElements can’t drawWith()

```java
/*
 * Use the given turtle to draw oneself
 * @param t the Turtle to draw with
 **/
public abstract void drawWith(Turtle t);
// No body in the superclass
```

SceneElementLayered drawWith()

```java
/**
 * Method to draw from this picture.
 * @param pen Turtle to draw with
 **/
public void drawWith(Turtle pen) {
    // We just ignore the pen’s position
    pen.moveTo(x,y);
    pen.drop(this.getPicture());
}
```

SceneElementPositioned

```java
public class SceneElementPositioned extends SceneElement {
    /**
     * Make a new element with a picture as input, and
     * next as null.
     * @param heldPic Picture for element to hold
     * @param xpos x position desired for element
     * @param ypos y position desired for element
     **/
    public SceneElementPositioned(Picture heldPic,
    int xpos, int ypos) {
        myPic = heldPic;
        next = null;
        x = xpos;
        y = ypos;
    }
    /**
     * Method to draw from this picture.
     * @param pen Turtle to use for drawing
     **/
    public void drawWith(Turtle pen) {
        pen.drop(this.getPicture());
    }
}
```

Version 4: 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.

Nesting is a tree relationship!

Labeling the Pieces

Moves

- MoveBranch to (10,50)
- MoveBranch to (10,400)
- MoveBranch to (300,450)

Branch (root)

HBranch with 3 BSN houses and a

VBranch with 3 BSN houses
It's a Tree (of instances!)

Branch (root)
- MoveBranch to (10,50)
- MoveBranch to (10,400)
- HBranch with 3 BSN houses and a
  VBranch with 3 BSN houses

The Class Structure
(another tree) 1. nodes
- 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 knows how to drawWith(turtle) by using bluescreen.

The Class Structure
(another tree) 2. branches
- Branch knows its children—a linked list of other nodes to draw. It knows how to drawWith by:
  1. (1) telling all its children to draw.
  2. (2) then telling its next 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

Using these Classes:
When doggies go bad!

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

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

  /**
   * 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
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 houses = 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);
houses.addChild(house1); houses.addChild(house2);
houses.addChild(house3);
houses.addChild(houses); // Yes, vBranch can be a child of an
village.addChild(houses);
```

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);
```
Trying out one scene:
Very important for testing!

```java
/**
 * 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

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

Wolvies attack! (for 25 frames)

```java
// First, the nasty wolives 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.

Exit the threatening wolves, enter the retreating wolves

```java
// 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);
```

Our hero arrives! (In frame 26)

```java
// Now, our hero arrives!
this.root().addChild(hero);
// 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 (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

Okay, how’d we do that?

• This part is important!
• Remember: You have to do this for your animation with sound!
  • You need to understand how this actually works!
  • And, by the way, there’s a lot of important Java in here!

DrawableNode: The root of the class structure

abstract public class DrawableNode {
    /**
     * Stuff that all nodes and branches in the scene tree know.
     */
    public DrawableNode next;
    public DrawableNode()
    {
        next = null;
    }
    public DrawableNode(DrawableNode nextOne)
    {
        this.next = nextOne;
    }
    public DrawableNode getNext()
    {
        return this.next;
    }
}

DrawableNodes know how to be (chained into) linked lists

/**
 * Methods to set and get next elements
 * @param nextOne next element in list
 */
public void setNext(DrawableNode nextOne){
    this.next = nextOne;
}
public DrawableNode getNext(){
    return this.next;
}
DrawableNodes know how to draw themselves (and list)

```java
/**
 * Use the given turtle to draw oneself
 * @param t the Turtle to draw with
 **/
 abstract public void drawWith(Turtle t);
// No body in the superclass
/**
 * Draw on the given picture
 **/
 public void drawOn(Picture bg){
    Turtle t = new Turtle(bg);
    t.setPenDown(false);
    this.drawWith(t);
}

An abstract method is one that superclasses MUST override—they have to provide their own implementation of it.
```

DrawableNodes know all that linked list stuff

```java
/**
 * Method to remove node from list, fixing links appropriately.
 * @param node element to remove from list.
 **/
 public void remove(DrawableNode node){
...
}

/**
 * Insert the input node after this node.
 * @param node element to insert after this.
 **/
 public void insertAfter(DrawableNode node){
...
}

/**
 * Return the last element in the list
 **/
 public DrawableNode last() {
...
}

/**
 * Add the input node after the last node in this list.
 * @param node element to insert after this.
 **/
 public void add(DrawableNode node){
    this.last().insertAfter(node);
}
```

PictNode is a kind of DrawableNode

```java
/**
 * PictNode is a class representing a drawn picture
 * node in a scene tree.
 **/
 public class PictNode extends DrawableNode {
    
    /**
     * The picture I'm associated with
     **/
    Picture myPict;
}
```

To construct a PictNode, first, construct a DrawableNode

```java
/**
 * Make me with this picture
 * @param pict the Picture I'm associated with
 **/
 public PictNode(Picture pict){
    super(); // Call superclass constructor
    myPict = pict;
} 
```

How PictNodes drawWith

```java
/**
 * Use the given turtle to draw oneself
 * @param pen the Turtle to draw with
 **/
 public void drawWith(Turtle pen){
    pen.drop(myPict);
} 
```

BlueScreenNodes know nothing new

```java
/**
 * BlueScreenNode is a PictNode that composes the picture using the bluescreen() method in Picture
 **/
 public class BlueScreenNode extends PictNode {
    
    /**
     * Construct does nothing fancy
     **/
    public BlueScreenNode(Picture p{
        super(p); // Call superclass constructor
    }
```
BlueScreenNodes draw differently

```java
/*
 * Use the given turtle to draw oneself
 * Get the turtle's picture, then bluescreen onto it
 * @param pen the Turtle to draw with
 */
public void drawWith(Turtle pen)
{
    Picture bg = pen.getPicture();
    myPict.bluescreen(bg, pen.getXPos(), pen.getYPos());
}
```

Branches add children

```java
public class Branch extends DrawableNode
{
    /*
     * A list of children to draw
     */
    public DrawableNode children;

    /*
     * Construct a branch with children and
     * next as null
     */
    public Branch()
    {
        super(); // Call superclass constructor
        children = null;
    }
}
```

Adding children to a Branch

```java
/**
 * Method to add nodes to children
 */
public void addChild(DrawableNode child)
{
    if (children != null)
        children.add(child);
    else
        children = child;
}
```

Drawing a Branch

```java
/**
 * Ask all our children to draw,
 * then let next draw.
 * @param pen Turtle to draw with
 */
public void drawWith(Turtle pen)
{
    DrawableNode current = children;
    // Tell the children to draw
    while (current != null)
    {
        current.drawWith(pen);
        pen.moveTo(pen.getXPos() + gap, pen.getYPos());
        current = current.getNext();
    }
    // Tell my next to draw
    if (this.getNext() != null)
        this.getNext().drawWith(pen);
}
```

HBranch: Horizontal Branches

```java
public class HBranch extends Branch
{
    /**
     * Horizontal gap between children
     */
    int gap;

    /*
     * Construct a branch with children and
     * next as null
     */
    public HBranch(int spacing)
    {
        super(); // Call superclass constructor
        gap = spacing;
    }
}
```

HBranch draws horizontal children

```java
/**
 * Ask all our children to draw,
 * then let next draw.
 * @param pen Turtle to draw with
 */
public void drawWith(Turtle pen)
{
    DrawableNode current = children;
    // Tell my children draw
    while (current != null)
    {
        current.drawWith(pen);
        pen.moveTo(pen.getXPos() + gap, pen.getYPos());
        current = current.getNext();
    }
    // Tell my next to draw
    if (this.getNext() != null)
        this.getNext().drawWith(pen);
}
```

But because they're DrawableNodes, too, they still know how to be linked lists. They reference things in two directions—as children and as next. Hence, they branch. Hence, a tree.
**VBranch is exactly the same, but vertically**

```java
public void drawWith(Turtle pen){
    DrawableNode current = children;
    // Have my children draw
    while (current != null){
        current.drawWith(pen);
        pen.moveTo(pen.getXPos(),pen.getYPos()+gap);
        current = current.getNext();
    }
    // Have my next draw
    if (this.getNext() != null) {
        this.getNext().drawWith(pen);
    }
}
```

**MoveBranch is different**

```java
public class MoveBranch extends Branch {
    /**
     * Position where to draw at
     */
    int x,y;
    /**
     * Construct a branch with children and
     * next as null
     */
    public MoveBranch(int x, int y){
        super(); // Call superclass constructor
        this.x = x;
        this.y = y;
    }
    /**
     * Set the location, then draw
     * @param pen Turtle to draw with
     */
    public void drawWith(Turtle pen){
        pen.moveTo(this.x,this.y);
        super.drawWith(pen); // Do a normal branch now
    }
}
```

**MoveBranch accessors, to make them movable**

```java
/**
 * Accessors
 */
public int getXPos() {return this.x;}
public int getYPos() {return this.y;}
public void moveTo(int x, int y){
    this.x = x; this.y = y;
}
```

**MoveBranch passes the buck on drawing**

```
```

**Doing the Branches...backwards**

- What if you processed *next before* the children?
- What if you did the move *after* you did the superclass drawing?
- What would the scene look like?
- Different kinds of tree traversals...

**Representing Structure and Behavior**

- Think about trees
  - Branches represent structure
  - HBranch, VBranch, and MoveBranch represent structure and behavior
- Think about objects
  - They know things, and they know how to do things.
  - They represent structure and behavior.
- Sophisticated programs represent both.
  - *The line between data and programs is very thin...*