Two kinds of Java Storage

➔ Dynamic: Allocated by new.
➔ (recycled by garbage collection)

➔ Local Lifetime:
➔ for method params and
➔ and locals ONLY.
➔ PUSHED on a stack when a
➔ method is called.
➔ POPPED when that call returns.
Chapter 8: Trees of Images
Chapter Objectives

The computer science goals for this chapter are:
- To create and use trees to represent collections of images.
- To construct and traverse trees.

The media learning goals for this chapter are:
- To introduce the structure of a scene graph, which is a common animation data structure.
- To use trees to create an animation.
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 \textit{cluster}.

• Is it the responsibility of the elements to know about layering and position?
  • Is that the right place to put that \textit{know how}? \\

• How do we structure \textit{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

• Linear sequences
  • of Pictures or their visible combinations
  • implemented by singly-linked lists

• Whole scene is described by a tree
  • Each Picture has no descendants but makes graphics appear (G&E call it a PictNode)
  • The immediate descendants of any one node are in a linear sequence (implemented by a singly linked list)
Blueprint for linear sequencing and painting

G&E's class DrawableNode {
    private DrawableNode next;
}
The Nodes of the Scene Tree: one instance of Branch at top..
First Level of the Scene

1. The 3 wolves (together)
2. The 8 trees (together)
3. The 6 houses (together)
Blueprint for linear sequencing

G&E's class DrawableNode {
    private DrawableNode next;
}
Blueprint for linear sequencing
AND FOR DRAWING

G&E's class DrawableNode {
    private DrawableNode next;
    public void drawOnOn(Picture bg) {
        Turtle turt=new Turtle(bg);
        this.drawWithWith(turt);
    }
    public abstract void drawWithWith(Turtle t);
    //BODY supplied ELSEWHERE!!
Turtle Graphics

Fond model for some computer graphics APIs.

The “turtle” has a location ((x,y) coordinates in a canvas, which could be a G&E Picture object.

The turtle has methods to change its location.

The turtle has a method to “drop” (copy in, as does the compose( ) method) a given Picture onto the canvas.
Blueprint for having a sequence of children
Blueprint for HAVING a linear sequence of children

class Branch extends DrawableNode{
    private DrawableNode first;
    public void drawWith(Turtle t)
    {
        DrawableNode finger=first;
        while(finger!=null) {
            finger.drawWith(t);
            finger=finger.getNext();
        }
        if(getNext()!=null)
            getNext.drawWith(t);
    }
}
DrawableNode
Can be a Node in a linked list, so it has a “next” pointer.
Has drawing behavior.

Branch extends DrawableNode
ALSO: Can have a list of children
Its drawing behavior draws its children too.
Blueprint for linear sequencing and painting

G&E's class DrawableNode {
    private DrawableNode next;
}
Blueprint for linear sequencing.......
Blueprint for having a sequence of children
More G&E Chapter 8 Software Architecture Concepts:

VBranch, HBranch extends Branch. Besides everything a Branch has, these have an int gap for the space between their children.

So, their drawWith methods use the value of gap to reposition the Turtle between drawing steps.
The Nodes of the Scene Tree:
one instance of Branch at top..
It’s a Tree

Branch (root)

- MoveBranch to (10,50)
  - VBranch with BlueScreenNode wolves
- MoveBranch to (10,400)
- MoveBranch to (300,450)
  - HBranch with 3 BSN houses and a HBranch with 3 BSN houses
  - HBranch with BSN trees
Scene Graphs: A tree to describe graphical scenes

- Modern graphics systems often build scene graphs.
  - Operations in branches.
  - Images in leaf nodes.
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 its next to draw.
- A **HBranch** draws its children by spacing them out horizontally.
- A **VBranch** draws its children by spacing them out vertically.
- A **MoveBranch** positions its children somewhere specific.
Note: This is *not* the same as the *scene (object)* structure!

**DrawableNode**
- Knows: next

**Branch**
- Knows: firstChild

**HBranch**
- Knows how to drawWith horizontally

**VBranch**
- Knows how to drawWith vertically

**MoveBranch**
- Knows: x,y
- Knows how to position then drawWith

**PictNode**
- Knows: myPict
- Knows how to drawWith

**BlueScreenNode**
- Knows how to drawWith as bluescreen
UML Class Structure Diagram

**DrawableNode**
- DrawableNode next;
- `drawWith(Turtle t)`

**Branch**
- DrawableNode `firstChild`;
- `drawWith(Turtle t)`

**HBranch**
- `drawWith(Turtle t)`
  - draws children horizontally

**VBranch**
- `drawWith(Turtle t)`
  - draws children vertically

**MoveBranch**
- `drawWith(Turtle t)`
  - draws children starting at this `x` and `y` position

**PictNode**
- `Picture myPict`;
- `drawWith(Turtle t)`

**BlueScreenNode**
- `drawWith(Turtle t)`
  - draw the non-blue parts of the picture
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

- MoveBranch to (10,50)
  - VBranch with BlueScreenNode wolves

- Branch (root)
  - MoveBranch to (10,400)
    - HBranch with BSN trees
  - MoveBranch to (300,450)
    - HBranch with 3 BSN houses and a
    - VBranch with 3 BSN houses
How delegate drawing?

• How does DrawableNode define drawing, but force the subclasses to handle it?
  • Abstract method
public abstract void drawWith(Turtle t);
// No body in an abstract method

/**
 * Draw on the given picture
 * @param bg the background picture to draw on
 */
public void drawOn(Picture bg) {
    Turtle t = new Turtle(bg);
    t.setPenDown(false);
    this.drawWith(t);
}
Making a Scene w/these Classes:
When dooggies go bad!

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

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

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

    // These are the nodes that change during the animation, so must be available outside the local method context
```
/**
 * Constructor that takes no arguments and 
 * sets up the movie 
 */

public WolfAttackMovie() {
    setUp();
}
/**
 * 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("mscary.jpg"));
}
/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);
Forest branch structure
// 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 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);
houses.addChild(house3);
houses.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!

```java
/**
 * Render just the first scene
 ***/

public void renderScene() {
    Picture bg = new Picture(500, 500);
    sceneRoot.drawOn(bg);
    bg.show();
}
```
Okay that works
public void renderAnimation() {
    frames = new FrameSequencer("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.

Where is sceneRoot declared? wolfEntry?
In the class.
If you declare them in this method, you will shadow (hide) the class level fields with local variables.
Our hero arrives! (In frame 26)

// Now, our hero arrives!
this.getSceneRoot().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
Branch root = this.getSceneRoot();  // creating an alias
root.getFirstChild().remove(wolfEntry);
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);
root.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);
    root.drawOn(bg);
    frames.addFrame(bg);

    // Tweak the data structure
    wolfretreat.moveTo(wolfretreat.getXPos()-10,
                       wolfretreat.getYPos()-20);
}
Welcome to DrJava.
> WolfAttackMovie wam = new WolfAttackMovie();
    wam.renderScene();
> wam.renderAnimation();
There are no frames to show yet. When you add a frame it will be shown
> wam.replay();

Can also put this in main():
public static void main (String[] args) {
    WolfAttackMovie movie = new WolfAttackMovie();
    movie.renderAnimation();
}
The Completed Movie
DrawableNode: The root of the class structure

/**
 * Stuff that all nodes and branches in the
 * scene tree know.
 ***/
abstract public class DrawableNode {
  /**
   * The next branch/node/whatever to process
   ***/
  privateDrawableNode next;

  /**
   * Constructor for DrawableNode just sets
   * next to null
   ***/
  public DrawableNode(){
    next = null;
  }
}
How does this work?

- Key insight: This is a list of lists.
  - Branches have a **children** link that point to the first node in the children list.
  - The siblings are linked together as a linked list, through **next** links.
The three MoveBranches aren’t actually all linked from the root. Rather, the first child is linked as `firstChild` then rest are linked as `next`.
Just the implementation tree

firstChild

link

Branch (root)

MoveBranch to (10,50)

MoveBranch to (10,400)

MoveBranch to (300,450)

HBranch with BSN trees

HBranch with 3 BSN houses and a...

VBranch with BlueScreenNode wolves

Wolf

Wolf

Wolf

house

house

house

VBranch with 3 BSN houses

...
DrawableNodes know how to be linked lists

```java
/**
 * 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);
    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

/*
 * 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;
}
```

If you want to call the superclass’s constructor, you must do it first.
/**
 * Use the given turtle to draw oneself
 * @param pen the Turtle to draw with
 ***/
public void drawWithWith(Turtle pen){
    pen.drop(myPict);
}
BlueScreenNodes know nothing new

/*
 * 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

/*
 * 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());
}
public class Branch extends DrawableNode {

  /*
   * A list of children to draw
   */
  private DrawableNode firstChild;

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

  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*. 

public void addChild(DrawableNode child) {
    // if we have no child then this is the first
    if (firstChild == null) {
        firstChild = child;
    }
    // else add to the end of the list of children
    else {
        firstChild.add(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 = firstChild;

    // Tell the children to draw
    while (current != null){
        current.drawWith(pen);
        current = current.getNext();
    }

    // Tell my next to draw
    if (this.getNext() != null)
    {this.getNext().drawWith(pen);}
}
```
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

/*
 * Ask all our children to draw,
 * then let next draw.
 * @param pen Turtle to draw with
 **/

public void drawWith(Turtle pen){
    DrawableNode current = this.getFirstChild();

    // Have my children draw
    while (current != null){
        current.drawWith(pen);
        pen.moveTo(pen.getXPos() + gap, pen.getYPos());
        current = current.getNext();
    }

    // Have my next draw
    if (this.getNext() != null)
        {this.getNext().drawWith(pen);}
}
VBranch is *exactly* the same, but vertically

```java
public void drawWith(Turtle pen){
    DrawableNode current = this.getFirstChild();

    // 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);
    }
}
```

So the gap gets added to the Y position, not the X.
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;
}
MoveBranch accessors, to make them movable

/**
 * 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

```java
/**
 * Set the location, then draw
 * @param pen Turtle to draw with
 **/

public void drawWithWith(Turtle pen){
    pen.moveTo(this.x,this.y);
    super.drawWith(pen); // Do a normal branch now
}
```
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...