

Problem Solving with Data Structures: A Multimedia Approach

Mark Guzdial and Barbara Ericson

College of Computing/GVU

Georgia Institute of Technology

ALPHA VERSION OF TEXT



August 26, 2007

Copyright held by Mark Guzdial and Barbara Ericson, 2006.

Dedicated to TBD.

Contents

Contents	iii
List of Program Examples	vi
List of Figures	ix
I Introduction to Java: Object-Oriented Programming for Modeling the World	5
1 Objects for Modeling the World	7
1.1 Making Representations of the World	8
1.2 Why Java?	14
2 Introduction to Java	19
2.1 What's Java about?	19
2.2 Basic (Syntax) Rules of Java	20
2.3 Using Java to Model the World	24
2.4 Manipulating Pictures in Java	35
2.5 Exploring Sound in Java	41
2.6 Exploring Music in Java	42
3 Methods in Java: Manipulating Pictures	45
3.1 Reviewing Java Basics	45
3.2 Java is about Classes and Methods	49
3.3 Methods that return something: Compositing images	57
3.4 Creating classes that do something	66
4 Objects as Agents: Manipulating Turtles	69
4.1 Turtles: An Early Computational Object	69
4.2 Drawing with Turtles	70
4.3 Creating animations with turtles and frames	78
5 Arrays: A Static Data Structure for Sounds	85
5.1 Manipulating Sampled Sounds	85

5.2 Inserting and Deleting in an Array	91
II Introducing Linked Lists	95
6 Structuring Music using Linked Lists	97
6.1 JMusic and Imports	97
6.2 Starting out with JMusic	102
6.3 Making a Simple Song Object	102
6.4 Simple structuring of notes with an array	105
6.5 Making the Song Something to Explore	105
6.6 Making Any Song Something to Explore	112
6.7 Exploring Music	131
7 Structuring Images using Linked Lists	147
7.1 Simple arrays of pictures	148
7.2 Listing the Pictures, Left-to-Right	148
7.3 Listing the Pictures, Layering	154
7.4 Reversing a List	161
7.5 Animation	162
7.6 Lists with Two Kinds of Elements	165
8 Abstract Data Types: Separating the Meaning from the Implementation	179
8.1 Introducing the Stack	179
8.2 Introduction to Queues	192
III Trees: Hierarchical Structures for Media	199
9 Trees of Images	201
9.1 Representing scenes with trees	201
9.2 Our First Scene Graph: Attack of the Killer Wolves	203
9.3 The Classes in the SceneGraph	205
9.4 Building a scene graph	208
9.5 Implementing the Scene Graph	215
9.6 Exercises	229
10 Lists and Trees for Structuring Sounds	231
10.1 Composing with Sampled Sounds and Linked Lists: Recursive Traversals	231
10.2 Using Trees to Structure Sampled Sounds	245
11 Generalizing Lists and Trees	263
11.1 Refactoring a General Linked List Node Class	263
11.2 Making a New Kind of List	273
11.3 The Uses and Characteristics of Arrays, Lists, and Trees . .	275

11.4 Binary Search Trees: Trees that are fast to search	281
12 Circular Linked Lists and Graphs: Lists and Trees That Loop	293
12.1 Making Cell Animation with Circular Linked Lists	293
12.2 Generalizing a Circular Linked List	298
12.3 Graphs: Trees with Loops	300
13 User Interface Structures	307
IV Simulations: Problem Solving with Data Structures	309
14 Using an Existing Simulation Package	311
15 Introducing UML and Continuous Simulations	313
15.1 Introducing Simulations	313
15.2 Our First Model and Simulation: Wolves and Deer	316
15.3 Modelling in Objects	318
15.4 Implementing the Simulation Class	322
15.5 Implementing a Wolf	325
15.6 Implementing Deer	330
15.7 Implementing AgentNode	331
15.8 Extending the Simulation	332
16 Abstracting Simulations: Creating a Simulation Package	341
17 Discrete Event Simulation	343
17.1 Distributions and Events	343
A MIDI Instrument names in JMusic	345
B Whole Class Listings	349
Bibliography	367
Index	368

List of Program Examples

Example Program: An Example Program	2
Example Program: Person class, starting place	25
Example Program: Person, with a private name	26
Example Program: Person, with constructors	27
Example Program: <code>toString</code> method for Person	29
Example Program: Student class, initial version	29
Example Program: Class Student, with constructors	30
Example Program: <code>main()</code> method for Person	31
Example Program: <code>greet</code> method for Person	32
Example Program: <code>greet</code> method for Student	33
Example Program: Method to increase red in Picture	37
Example Program: Method to flip an image	40
Example Program: <code>decreaseRed</code> in a Picture	52
Example Program: <code>decreaseRed</code> with an input	55
Example Program: Method to compose this picture into a target	57
Example Program: Method for Picture to scale by a factor	60
Example Program: Methods for general chromakey and bluescreen	64
Example Program: A public static void main in a class	67
Example Program: Creating a hundred turtles	73
Example Program: Making a picture with dropped pictures	77
Example Program: An animation generated by a Turtle	79
Example Program: Increase the volume of a sound by a factor	86
Example Program: Reversing a sound	87
Example Program: Create an audio collage	88
Example Program: Append one sound with another	89
Example Program: Mix in part of one sound with another	90
Example Program: Scale a sound up or down in frequency	90
Example Program: Inserting into the middle of sounds	92
Example Program: <i>Amazing Grace</i> as a Song Object	102
Example Program: Amazing Grace with Multiple Voices	106
Example Program: Amazing Grace as Song Elements	108
Example Program: Amazing Grace as Song Elements, Take 2	113
Example Program: General Song Elements and Song Phrases	120
Example Program: More phrases to play with	123
Example Program: Computed Phrases	127

Example Program: 10 random notes SongPhrase	129
Example Program: 10 slightly less random notes	130
Example Program: SongNode class	132
Example Program: Repeating and weaving methods	135
Example Program: RepeatNextInserting	138
Example Program: SongPart class	141
Example Program: Song class—root of a tree-like music structure	142
Example Program: MySong class with a main method	143
Example Program: Elements of a scene in position order	149
Example Program: Methods to remove and insert elements in a list	152
Example Program: LayeredSceneElements	155
Example Program: Reverse a list	161
Example Program: Create a simple animation of a dog running	163
Example Program: Abstract method drawWith in abstract class SceneElement 167	
Example Program: SceneElement	168
Example Program: SceneElementPositioned	171
Example Program: SceneElementLayered	172
Example Program: MultiElementScene	173
Example Program: Modified drawFromMeOn in SceneElement	175
Example Program: Testing program for a PictureStack	182
Example Program: PictureStack	183
Example Program: PictureStack—push, peek, and pop	184
Example Program: PictureStack—size and empty	185
Example Program: Stack implementation with a linked list	186
Example Program: Stack implementation with an array	187
Example Program: Stack implementation with an array—methods	187
Example Program: Reverse a list—repeated	190
Example Program: Reverse with a stack	191
Example Program: Queue implemented as a linked list	194
Example Program: Queue implemented as an array	195
Example Program: The drawing part of DrawableNode	206
Example Program: Start of WolfAttackMovie class	208
Example Program: Start of setUp method in WolfAttackMovie	209
Example Program: Rest of setUp for WolfAttackMovie	210
Example Program: Rendering just the first scene in WolfAttackMovie	212
Example Program: renderAnimation in WolfAttackMovie	212
Example Program: DrawableNode	217
Example Program: PictNode	220
Example Program: BlueScreenNode	221
Example Program: Branch	222
Example Program: HBranch	225
Example Program: VBranch	226
Example Program: MoveBranch	227
Example Program: SoundElement	232

Example Program: SoundListText: Constructing a SoundElement	
list	238
Example Program: RepeatNext for SoundElement	239
Example Program: Weave for SoundElement	239
Example Program: copyNode for SoundElement	240
Example Program: De-gonging the list	241
Example Program: replace for SoundElement	242
Example Program: SoundTreeExample	246
Example Program: CollectableNode	250
Example Program: SoundNode	251
Example Program: SoundBranch	253
Example Program: ScaleBranch	254
Example Program: LLNode, a generalized linked list node class	264
Example Program: DrawableNode, with linked list code factored out	267
Example Program: CollectableNode, with linked list code factored out	270
Example Program: StudentNode class	273
Example Program: TreeNode, a simple binary tree	283
Example Program: insertInOrder for a binary search tree	285
Example Program: find, for a binary search tree	286
Example Program: traverse, a binary tree in-order	288
Example Program: addFirst and addLast, treating a tree as a list	290
Example Program: WalkingDoll	296
Example Program: CharNode, a class for representing characters in cell animations	299
Example Program: WolfDeerSimulation.java	352
Example Program: Wolf.java	355
Example Program: Deer.java	358
Example Program: AgentNode	361
Example Program: HungryWolf	363

List of Figures

1.1	Wildebeests in <i>The Lion King</i>	7
1.2	Parisian villagers in <i>The Hunchback of Notre Dame</i>	8
1.3	Katie's list of treasure hunt clues	10
1.4	An organization chart	11
1.5	A map of a town	11
1.6	Opening the DrJava Preferences	16
1.7	Adding the JMusic libraries to DrJava in Preferences	16
1.8	Adding <code>java-source</code> to DrJava	17
1.9	Parts of DrJava window	17
2.1	Showing a picture	36
2.2	Doubling the amount of red in a picture	38
2.3	Doubling the amount of red using our <code>increaseRed</code> method	39
2.4	Flipping our guy character—original (left) and flipped (right)	41
2.5	Just two notes	43
3.1	Structure of the Picture class defined in Picture.java	50
3.2	Part of the JavaDoc page for the Pixel class	56
3.3	Composing the guy into the jungle	58
3.4	Mini-collage created with scale and compose	63
3.5	Using the explore method to see the sizes of the guy and the jungle	64
3.6	Chromakeying the monster into the jungle using different levels of bluescreening	65
3.7	Run the main method from DrJava	68
4.1	Starting a Turtle in a new World	71
4.2	A drawing with a turtle	72
4.3	What you get with a hundred turtles starting from the same point, pointing in random directions, then moving forward the same amount	75
4.4	Dropping the monster character	76
4.5	Dropping the monster character after a rotation	76
4.6	An iterated turtle drop of a monster	77
4.7	Making a more complex pattern of dropped pictures	78

6.1	Playing all the notes in a score	98
6.2	Frequencies, keys, and MIDI notes—something I found on the Web that I need to recreate in a new way	99
6.3	Viewing a multipart score	101
6.4	JMusic documentation for the class Phrase	101
6.5	Playing all the notes in a score	102
6.6	Trying the Amazing Grace song object	104
6.7	A hundred random notes	105
6.8	Multi-voice <i>Amazing Grace</i> notation	107
6.9	AmazingGraceSongElements with 3 pieces	111
6.10	AmazingGraceSongElements with 3 pieces	112
6.11	Playing some different riffs in patterns	127
6.12	Sax line in the top part, rhythm in the bottom	129
6.13	We now have layers of software, where we deal with only one at a time	130
6.14	First score generated from ordered linked list	135
6.15	Javadoc for the class SongNode	135
6.16	Repeating a node several times	137
6.17	Weaving a new node among the old	138
6.18	Multi-part song using our classes	144
7.1	Array of pictures composed into a background	148
7.2	Elements to be used in our scenes	149
7.3	Our first scene	151
7.4	Our second scene	152
7.5	Removing the doggy from the scene	154
7.6	Inserting the doggy into the scene	154
7.7	First rendering of the layered scene	158
7.8	A doubly-linked list	159
7.9	Second rendering of the layered scene	160
7.10	A few frames from the AnimatedPositionedScene	165
7.11	The abstract class SceneElement, in terms of what it knows and can do	167
7.12	The abstract class SceneElement and its two subclasses	172
7.13	A scene rendered from a linked list with different kinds of scene elements	174
7.14	Same multi-element scene with pen traced	176
8.1	A pile of plates—only put on the top, never remove from the bottom	180
8.2	Later items are at the head (top) of stack	180
8.3	New items are inserted at the top (head) of the stack	181
8.4	Items are removed from the top (head) of a stack	181
8.5	An empty stack as an array	189
8.6	After pushing Matt onto the stack-as-an-array	189
8.7	After pushing Katie and Jenny, then popping Jenny	190

8.8 A basic queue	193
8.9 Elements are removed from the top or head of the queue	193
8.10 New elements are pushed onto the tail of the queue	193
8.11 When the queue-as-array starts out, head and tail are both zero	197
8.12 Pushing Matt onto the queue moves up the head to the next empty cell	197
8.13 Pushing Katie on moves the head further right	197
8.14 Popping Matt moves the tail up to Katie	198
9.1 A simple scene graph	202
9.2 A more sophisticated scene graph based on Java 3-D	203
9.3 The nasty wolfies sneak up on the unsuspecting village in the forest	203
9.4 Then, our hero appears!	204
9.5 And the nasty wolfies scamper away	204
9.6 Mapping the elements of the scene onto the scene graph	205
9.7 Stripping away the graphics—the scene graph is a tree	205
9.8 Hierarchy of classes used in our scene graph	206
9.9 The forest branch created in setUp—arrows point to children . .	210
9.10 Reserving more memory for the Interactions Pane in DrJava's Preferences pane	215
9.11 The implementation of the scene graph overlaid on the tree ab- straction	216
9.12 The actual implementation of the scene graph	216
9.13 How we actually got some of the bluescreen pictures in this book, such as our hero in WolfAttackMovie	221
10.1 The initial SoundElement list	236
10.2 As we start executing playFromMeOn()	236
10.3 Calling e2.collect()	237
10.4 Finally, we can return a sound	237
10.5 Ending e2.collect()	238
10.6 Starting out with e1.replace(croak,clap)	243
10.7 Checking the first node	243
10.8 Replacing from e2 on	244
10.9 Finally, replace on node e3	244
10.10 Our first sampled sound tree	245
10.11 The core classes in the CollectableNode class hierarchy	249
10.12 Extending the class hierarchy with ScaleBranch	250
10.13 Starting out with tree.root().collect()	257
10.14 Asking the root's children to collect()	257
10.15 Asking the first SoundNode to collect()	258
10.16 Asking the next SoundNode to collect()	258
10.17 Collecting from the next of the SoundBranch	259
10.18 Collecting from the last SoundBranch	259

11.1 An example organization chart	277
11.2 An equation represented as a tree	278
11.3 A tree of meanings	279
11.4 A sample sentence diagram	279
11.5 A query for a collection of sentence trees	280
11.6 A user interface is a tree	281
11.7 Simple binary tree	282
11.8 More complex binary tree	282
11.9 Tree formed by the names example	286
11.10 An unbalanced form of the last binary search tree	288
11.11 Rotating the right branch off “betty”	288
11.12 An equation tree for different kinds of traversals	290
12.1 Scenes from Super Mario Brothers	294
12.2 Three images to be used in a cell animation	294
12.3 A sequence of images arranged to give the appearance of walking	295
12.4 A circular linked list of images	295
12.5 Frames of the walking doll	296
12.6 A partial circular linked list	298
12.7 A map as a graph	301
12.8 Apply weights to a graph—distances on a map	302
12.9 Traversing a graph to create a spanning tree	303
12.10 Choosing the cheapest path out of Six Flags	304
12.11 Going to College Park	304
12.12 Backtracking to avoid re-visiting Six Flags	305
12.13 Adding Dunwoody, the obviously cheaper path	305
12.14 Finishing up in Charlotte	306
15.1 An execution of our wolves and deer simulation	316
15.2 The class relationships in the Wolves and Deer simulation . . .	317
15.3 A UML class diagram for the wolves and deer simulation . . .	320
15.4 One UML class	320
15.5 A Reference Relationship	321
15.6 A Gen-Spec (Generalization-Specialization) relationship . . .	322
15.7 The structure of the wolves linked list	324

Preface

The focus in this book is on teaching data structures as a way to solve problems in modeling the world and executing (simulating) the resultant model. We cover the standard data structures topics (e.g., arrays, linked lists, trees, graphs, stacks, and queues) but in the context of modeling situations then creating simulations (often generating animations).

The presumption is that the reader has had *some* previous programming experience. We expect that the reader can build programs that use iteration via **while** and **for**, and that the reader can assemble that program using functions that pass input via arguments. The reader should know what an array and matrix are. But we don't care what language that previous experience is in.

We use DrJava in examples in this text. *It is not necessary to use DrJava to use this book!* The advantage of DrJava is a simple interface and a powerful interactions pane, which allows us to manipulate objects without writing new methods or classes for each exploration. Rapid iteration allows students to explore and learn a wide space more quickly than they might if each exploration required a new Java file or method.

Typographical notations

Examples of Java code look like this: `x = x + 1`. Longer examples look like this:

```
public static void main(String[] args){  
    System.out.println("Hello, World!");  
}
```

When showing something that the user types in with DrJava's response, it will have a similar font and style, but the user's typing will appear after a prompt (`>`):

```
> int a = 5;  
> a + 7  
12
```

User interface components of DrJava will be specified using a smallcaps font, like SAVE menu item and the LOAD button.

There are several special kinds of sidebars that you'll find in the book.

Utility Program

Utility #1: An Example Utility

Utility programs are new pieces with which we will construct our models—not necessarily to be studied for themselves, but offered as something interesting to study and expand upon. They appear like this:

```

1  public class Greeter
2  {
3      public static void main(String[] argv)
4      {
5          // show the string "Hello World" on the console
6          System.out.println("Hello World");
7      }
8  }
```

Program Example #0

Example Java Code: An Example Program

A program creates a model of interest to us.

```

1  import jm.music.data.*;
2  import jm.JMC;
3  import jm.util.*;
4  import jm.music.tools.*;

6  public class Dot03 {
7      public static void main(String[] args) {
8          Note n = new Note(JMC.C4, JMC.QUARTERNOTE);
9          Phrase phr = new Phrase(0.0);
10         phr.addNote(n);
11         Mod.repeat(phr, 15);

12         Phrase phr2 = new Phrase(0.0);
13         Note r = new Note(JMC.REST, JMC.EIGHTHNOTE);
14         phr2.addNote(r);
15         Note n2 = new Note(JMC.E4, JMC.EIGHTHNOTE);
16         phr2.addNote(n2);
17         Note r2 = new Note(JMC.REST, JMC.QUARTERNOTE);
18         phr2.addNote(r2);
19         Mod.repeat(phr2, 7);

22         Part p = new Part();
23         p.addPhrase(phr);
24         p.addPhrase(phr2);

26         View.show(p);
    }
```

Computer Science Idea: An Example Idea

Powerful computer science concepts appear like this.

A Problem and Its Solution: The Problem that We're Solving

We use data structures to solve problems in modeling the world. In these side bars, we explicitly identify the problem and its solution.

Common Bug: An Example Common Bug

Common things that can cause your recipe to fail appear like this.

Debugging Tip: An Example Debugging Tip

If there's a good way to keep those bugs from creeping into your recipes in the first place, they're highlighted here.

Making It Work Tip: An Example How To Make It Work

Best practices or techniques that really help are highlighted like this.

Acknowledgements

My sincere thanks go out to the following:

- The National Science Foundation who gave us the initial grants that started the Media Computation project;
- Robert “Corky” Cartwright and the whole DrJava development team at Rice University;

- Andrew Sorensen and Andrew Brown, the developers of JMusic;
- Finally but most importantly, Barbara Ericson, and Matthew, Katherine, and Jennifer Guzdial, who allowed themselves to be photographed and recorded for Daddy's media project.

Part I

Introduction to Java: Object-Oriented Programming for Modeling the World

1 Objects for Modeling the World

In the 1994 Disney animated movie *The Lion King*, there is a scene when wildebeests charge over the ridge and stampede the lion king, Mufasa (Figure 1.1¹). Later, in the 1996 Disney animated movie *The Hunchback of Notre Dame*, Parisian villagers mill about, with a decidedly different look than the rest of the characters (see bottom of Figure 1.2²). These are actually related scenes. The wildebeests' stampede was one of the rare times that Disney broke away from their traditional hand-drawn cel animation. The wildebeests were not drawn by hand at all—rather, they were *modeled* and then brought to life in a *simulation*.

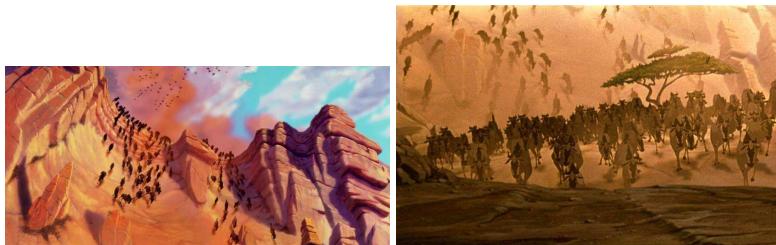


Figure 1.1: Wildebeests in *The Lion King*

A model is a detailed description of structure and behavior. The model of the wildebeests for *The Lion King* described what wildebeests looked like, how they moved, and what they did in a stampede. The villagers' model described what they did when milling about and how they reacted as a group to something noteworthy, like the entrance of Quasimodo. A simulation is execution of the model—simply let the wildebeests start responding to one another and to the obstacles on the ridge, according to the behavior defined in their model. Then, in a sense, simply “film” the screen.

This is a different process than when Pixar created *Toy Story*. There is a model for Woody, which describes how Woody looks and what parts of him move together when he smiles or walks. But *Toy Story* wasn't a

¹Images copyright ©1994, 1995 The Walt Disney Company.

²Images copyright ©1995, 1996 The Walt Disney Company.



Figure 1.2: Parisian villagers in *The Hunchback of Notre Dame*

simulation. The movements and character responses of *Toy Story* were carefully scripted. In the wildebeest or villagers simulations, each character is simply following a set of rules, usually with some random element (e.g., Should the wildebeest move left or right when coming up against the rock? When should the villagers shuffle or look right?) If you run a simulation a second time, depending on the model and the random variables you used, you may get a different result than you did the first time.

This book is about understanding these situations. The driving questions of this book are “***How did the wildebeests stampede over the ridge? How did the villagers move and wave?***”. The process of answering those questions will require us to cover a lot of important computer science concepts, like how to choose different kinds of *data structures* to model different kinds of structures, and how to define behavior and even combine structure and behavior in a single model. We will also develop a powerful set of tools and concepts that will help us understand how to use modelling and simulation to answer important questions in history or business.

1.1 Making Representations of the World

What we’re doing when we model is to construct a representation of the world. Think about our job as being the job of an artist—specifically, let’s consider a painter. Our canvas and paints are what we make our world out of. That’s what we’ll be using *Java* for.

Is there more than one way to model the world? Can you imagine two different paintings, perhaps *radically* different paintings, of the same thing? Part of what we have to do is to pick the software structures that best represents the structure and behavior that we want to model. Making those choices is solving a *representation problem*.

You already know about mathematics as a way to model the world,

though you may not have thought about it that way. An equation like $F = ma$ is saying something about how the world works. It says that the amount of force (F) in a collision (for example) is equal to the amount of mass (m) of the moving object times its acceleration (a). You might be able to imagine a world where that's not true—perhaps a cartoon world where a slow-moving punch packs a huge wallop. In that world, you'd want to use a different equation for force F .

The powerful thing about software representations is that they are executable—they have *behavior*. They can move, speak, and take action within the simulation that we can interpret as complex behavior, such as traversing a scene and accessing resources. A computer model, then, has a *structure* to it (the pieces of the model and how they relate) and a *behavior* to it (the actions of these pieces and how they interact).

Are there better and worse *physical structures*? Sure, but it depends on what you're going to use them for. A skyscraper and a duplex home each organize space differently. You probably don't want a skyscraper for a nuclear family with 2.5 children, and you're not going to fit the headquarters of a large multinational corporation into a duplex. Consider how different the physical space of a tree is from a snail—each has its own strengths for the contexts in which they're embedded.

Are there better and worse information structures, *data structures*? Imagine that you have a representation that lists all the people in your department, some 50–100 of them sorted by last names. Now imagine that you have a list of all the people in your work or academic department, but grouped by role, e.g., teachers vs. writers vs. administrative staff vs. artists vs. management, or whatever the roles are in your department. Which representation is *better*? Depends on what you're going to do with it.

- If you need to look up the phone number of someone whose name you know, the first representation is probably better.
- If the artistic staff gets a new person, the second representation makes it easier to write the new person's name in at the right place.

Computer Science Idea: Better or worse structures depend on use

A structure is better or worse depending on how it's going to be used – both for access (looking things up) and for change. How will the structure be changed in the future? The best structures are fast to use and easy to change in the ways that you need them to change.

Structuring our data is *not* something new that appeared when we started using computers. There are lots of examples of data structuring and the use of representations in your daily life.

- Consider the stock listing tables that appear in your paper. For each stock (arranged vertically into rows), there is information such as the closing price and the difference from the day before (in columns). A *table* appears in the computer as a *matrix*.
- My daughter, Katie, likes to create treasure hunts for the family, where she hides notes in various rooms (Figure 1.3). Each note references the next note in the list. This is an example of a *linked list*. Each note is a link in a chain, where the note tells you (links to) the next link in the chain. Think about some of the advantages of this structure: the pieces define a single structure, even though each piece is physically separate from the others; and changing the order of the notes or inserting a new note only requires changing the neighbor lists (the ones before or after the notes affected).

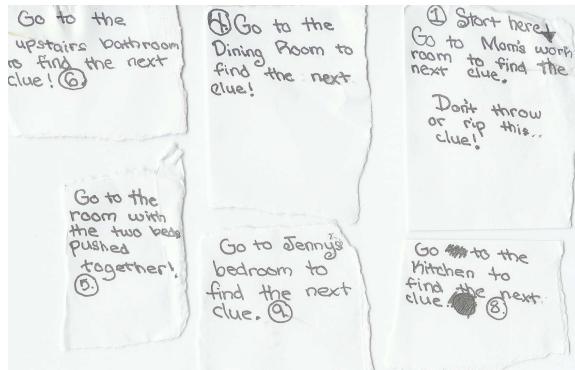


Figure 1.3: Katie’s list of treasure hunt clues

- An organization chart (Figure 1.4) describes the relationships between roles in an organization. It’s just a representation—there aren’t really lines extending from the feet of the CEO into the heads of the Presidents of a company. This particular representation is quite common—it’s called a *tree*. It’s a common structure for representing *hierarchy*.
- A map (Figure 1.5) is another common representation that we use. The real town actually doesn’t look like that map. The real streets have other buildings and things on them—they’re wonderfully rich and complex. When you’re trying to get around in the town, you don’t want a satellite picture of the town. That’s too much detail. What you really want is an *abstraction* of the real town, one that just shows you what you need to know to get from one place to another. We think about Interstate I-75 passing through Atlanta, Chattanooga, Knoxville, Cincinnati, Toledo, and Detroit, and Interstate I-94 goes

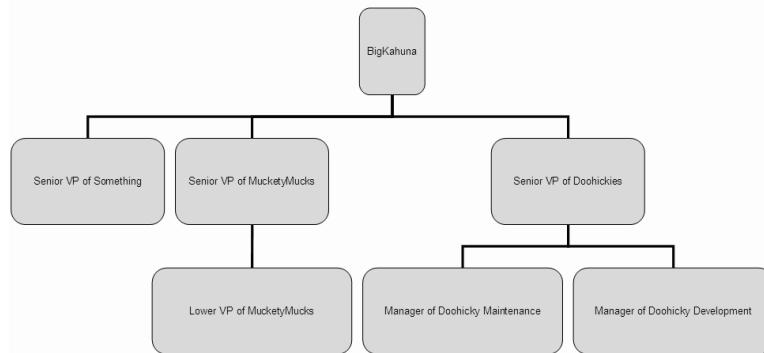


Figure 1.4: An organization chart

from Detroit through Chicago. We can think about a map as *edges* or *connections* (streets) between points (or *nodes*) that might be cities, intersections, buildings, or places of interest. This kind of a structure is called a *graph*.

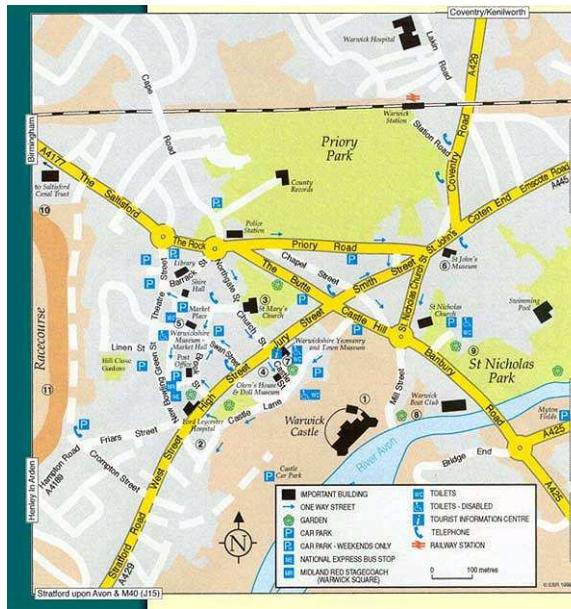


Figure 1.5: A map of a town

Each of these data structures have particular *properties* that make them good for some purposes and bad for others. A table or matrix is re-

ally easy for looking things up (especially if it's ordered in some way). But if you have to insert something into the middle of the table, everything else has to move down. When we're talking about space in the computer (*memory*), we're literally talking about moving each element in memory separately. On the other hand, inserting a new element into a linked list or into a graph is easy—just add edges in the right places.

How does it matter what kind of structure that you're using? It matters because of the way that computer memory works. Remember that you can think of memory as being a whole bunch of mailboxes in a row, each with its own address. Each mailbox stores exactly one thing. In reality, that one thing is a binary pattern, but we can interpret it any way we want, depending on the encoding. Maybe it's a number or maybe it's a character.

A table (a matrix or an array) is stored in consecutive mailboxes. So, if you have to put something into the middle of a table, you have to move the things already in there somewhere else. If you put something new where something old used to be, you end up over-writing the something old.

To make it clear, let's imagine that we have a table that looks something like this:

Name	Age	Weight
Arnold	12	220
Kermit	47	3
Ms. Piggy	42	54

Let's say that we want to add "Fozzie" to the list, who's 38 and weighs 125 pounds. He would go below Arnold and above Kermit, but if just put him after Arnold, we would over-write Kermit. So, the first thing we have to do is to make room for Fozzie at the *bottom* of the table. (We can simply annex the next few mailboxes after the table.)

Name	Age	Weight
Arnold	12	220
Kermit	47	3
Ms. Piggy	42	54

Now we have to copy everything down into the new space, opening up a spot for Fozzie. We move Ms. Piggy and her values into the bottom space, then Kermit into the space where Ms. Piggy was. That's two *sets* of data that we have to change, with three values in each set.

Notice that that leaves us with Kermit's data duplicated. That's okay—we're about to overwrite them.

Name	Age	Weight
Arnold	12	220
Fozzie	38	125
Kermit	47	3
Ms. Piggy	42	54

Now let's compare that to a different structure, one that's like the treasure trail of notes that Katie created. We call that a linked list representation. Consider a note (found in a bedroom) like:

“The next note is in the room where we prepare food.”

Let's think about that as a note *in* the bedroom that *references* (says to go to) the kitchen. We'll draw that like this:

In terms of memory mailboxes, think about each note as having two parts: a current location, and where *next* is. Each note would be represented as two memory mailboxes—something like this:

Current location: Bedroom	Where to go next: Kitchen
------------------------------	------------------------------

So let's imagine that Katie has set up a trail that looks like this:

Katie's bedroom

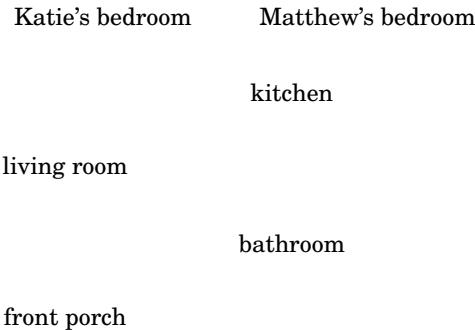
kitchen

living room

bathroom

front porch

Now, she changes her mind. Katie's bedroom shouldn't refer to the kitchen; her bedroom should point to Matthew's bedroom. How do we change that? Unlike the table, we don't have to move any data anywhere. We simply make Matthew's bedroom (anywhere), then point Katie's bedroom's note to Matthew's bedroom, and point Matthew's bedroom's note to the kitchen (where Katie's bedroom used to point).



In terms of memory mailboxes, we only changed the *next* part of Katie's bedroom note, and the location and next parts of the (new) Matthew's bedroom note. No copying of data was necessary.

Adding to a linked list representation is much easier than adding to a table, especially when you're adding to the middle of the table. But there are advantages to tables, too. They can be faster for looking up particular pieces of information.

Much of this book is about these trade-offs between different data structures. Each data structure has strengths that solve some sets of problems, but the same data structure probably has weaknesses in other areas. Each choice of data structure is a trade-off between these strengths and weaknesses, and the choices can only be made in the context of a particular problem.

These data structures have a *lot* to do with our wildebeests and villagers.

- The visual structure of villagers and wildebeests (e.g., how legs and arms attach to bodies) is typically described as a tree or graph.
- Tracking which villager is next to do something (e.g., move around) is a queue.
- Tracking all of the wildebeests to stampede is often done in a *list* (like a linked list).
- The images to be used in making the villagers wave or wildebeests run are usually stored in a list.

1.2 Why Java?

Why is this class taught in Java?

- Overall, Java is faster than Python (and definitely faster than Jython). We can do more complex things faster in Java than in Python.

- Java is a good language for exploring and learning about data structures. It makes it explicit how you're connecting data through *references*.
- More computer science classes are taught in Java than Python. So if you go on beyond this class in data structures, knowing Java is important.
- Java has “resume-value.” It’s a well-known language, so it’s worth it to be able to say, even to people who don’t really know computer science, that you know Java. This is important—you’ll learn the content better if you have good reason for learning it.

Getting Java Set-Up

You can start out with Java by simply downloading a *JDK* (*Java Development Kit*) from <http://www.java.sun.com> for your computer. With that, you have enough to get started programming Java. However, that’s not the easiest way to *learn* Java. In this book, we use *DrJava* which is a useful *IDE* (*Integrated Development Environment*)—a program that combines facilities for editing, compiling, debugging, and running programs. DrJava is excellent for learning Java because it provides an Interactions Pane where you can simply type in Java code and try it out. No files or compilers necessary to get started.

If you’d like to use DrJava, follow these steps:

- Download and install DrJava from <http://www.drjava.org>.
- Download and install *JMusic* from <http://jmusic.ci.qut.edu.au/>.
- You’ll need to tell DrJava about JMusic in order to access it. You use the Preferences in DrJava (see Figure 1.6) to add in the JMusic *jar file* and the instruments (Figure 1.7).
- Make sure that you grab the `MediaSources` and `java-source` from the CD or the website.
- Just as you added JMusic to your DrJava preferences, add the `java-source` folder to your preferences, too.

Making It Work Tip: Keep all your Java files in your `java-source` directory

Once you put `java-source` in your Preferences, you will have added it to Java’s *classpath*. That means that everything you create will be immediately accessible and easy to build upon. (Figure 1.8).

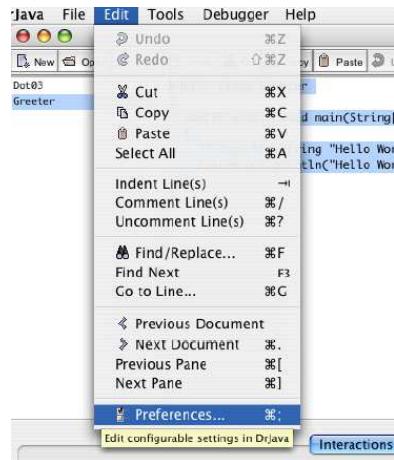


Figure 1.6: Opening the DrJava Preferences

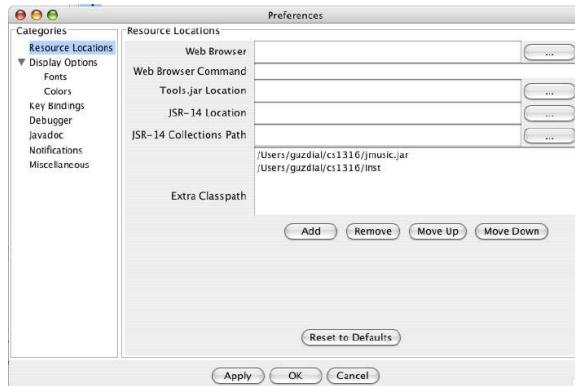


Figure 1.7: Adding the JMusic libraries to DrJava in Preferences

* * *

Once you start DrJava, you'll have a screen that looks like Figure 1.9. If you choose *not* to use DrJava, that's fine. Set up your IDE as best you wish, but be sure to install JMusic and set up your classpath to access JMusic and `java-source` directory. This book will assume that you're using DrJava and will describe using classes from the Interactions Pane, but you can easily create a class with a main method (as we'll start talking

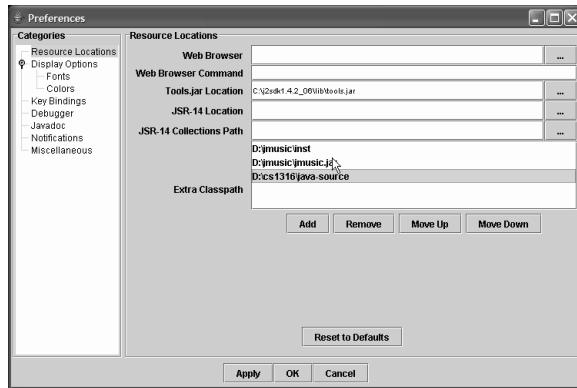
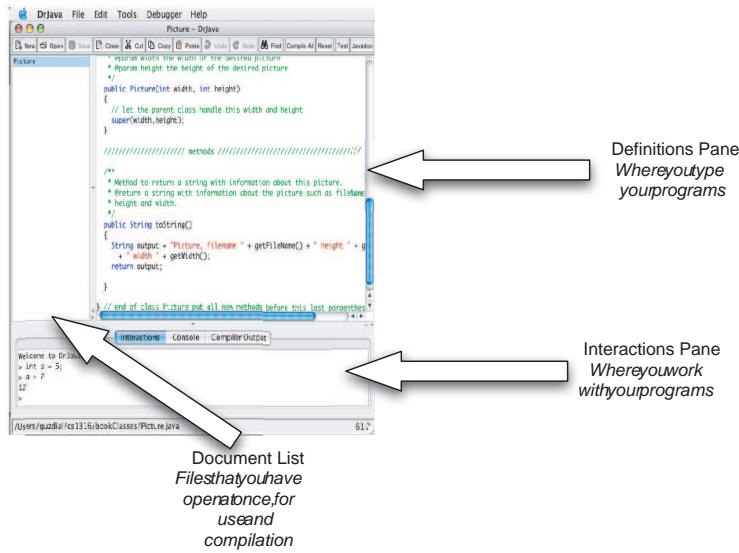
Figure 1.8: Adding `java-source` to DrJava

Figure 1.9: Parts of DrJava window

2 Introduction to Java

Chapter Learning Objectives

- Introducing Java with an explanation of why it's relevant to modelling and simulation.
- Brief taste of media manipulation of pictures, sounds, and music.

2.1 What's Java about?

Nearly everything in Java is an *object*. In *object-oriented programming*, the programmer cares about more than just *specifying* a process. In other languages, like Python or Visual Basic, you mostly tell the computer “First you do this, then you do that.” In object-oriented programming (which you have to do in Java, since it’s almost all objects), you care about who (or what) does the process, and how the overall process *emerges* from the interaction of different objects. The software engineering term for this is *responsibility-driven design*—we don’t just care about how the process happens, we care about who (which object) does which part of the process.

Object-oriented programming dates back to a programming language called *Simula*, which was a programming language for creating simulations of the world. The idea was to describe the world that you cared about in the Simula language, e.g. how customers worked their way through a store floor, how material flowed through a factory, how deer and wolves balanced each other ecologically in the ecosystem. That description is called a *model*. When Alan Kay discovered Simula in the late 1960’s, he realized that all programs can be thought of as modelling some world (real or imaginary) and all programming is about simulation. It was that insight that led to his programming language *Smalltalk* and our current understanding of object-oriented programming, which is what leads us to Java.

Thinking about programming as modelling and simulation means that you have to do this responsibility-driven design—you have to share control over what happens in the overall process across many objects. That’s the way that the real world works. Setting aside theological arguments, there is no great big **for** loop telling everything in the real world to take another time step. You don’t write one big master program in Java—your program arises out of the interaction of lots of objects, just like the real world. Most

importantly, in the real world, no *one* object **knows** everything and can **do** everything. Instead, in the real world and in Java, each object has things that it *knows* and things that it can *do* (or knows how to do).

2.2 Basic (Syntax) Rules of Java

Here are the basic rules for doing things in Java. We'll not say much about classes and methods here—we'll introduce the syntax for those as we need them. These are the things that you've probably already seen in other languages.

Declarations and Types

If your past experience programming was in a language like Python, Visual Basic, or Scheme, the trickiest part of learning Java will probably be its *types*. All variables and values (including what you get back from *functions*—except that there are no functions, only *methods*) are typed. We must declare the type of a variable before we use it. The types Picture, Sound, and Sample are already created in the base classes for this course for you. Other types are built-in for Java.

Many of these types are actually the names of classes names. A class specifies what all the objects of that class *know* and can *do*. The Picture class specifies what pictures can do (e.g., show() themselves) and what they know (e.g., they know their pixels). We *declare* variables to only hold objects of particular classes.

Java, unlike those other languages, is *compiled*. The Java compiler actually takes your Java program code and turns it into another program in another language—something close to *machine language*, the bytes that the computer understands natively. It does that to make the program run faster and more efficiently.

Part of that efficiency is making it run in as little memory as possible—as few bytes, or to use a popular metaphor for memory, mailboxes. If the compiler knows just how many bytes each variable will need, it can make sure that everything runs as tightly packed into memory as possible. How will the compiler know which variables are integers and which are floating point numbers and which are pictures and which are sounds? We'll tell it by *declaring* the type of the variable.

```
> int a = 5;
> a + 7
12
```

In the below java, we'll see that we can only declare a variable once, and a floating point number must have an “f” after it.

```
> float f;
> f = 13.2;
```

```
Error: Bad types in assignment
> float f = 13.2f;
Error: Redefinition of 'f'
> f = 13.2f
13.2
```

The type **double** is also a floating point number, but doesn't require anything special.

```
> double d;
> d = 13.231;
> d
13.231
> d + f
26.43099980926514
```

There are strings, too.

```
> String s = "This is a test";
> s
"This is a test"
```

Assignment

VARIABLE = EXPRESSION

The equals sign (=) is assignment. The left VARIABLE should be replaced with a declared variable, or (if this is the first time you're using the variable) you can declare it in the same assignment, e.g., **int a = 12;**. If you want to create an object (not a *literal* like the numbers and strings in the last section, you use the term **new** with the name of the class (maybe with an input for use in constructing the object).

```
> Picture p = new Picture(FileChooser.pickAFile());
> p.show();
```

All statements are separated by semi-colons. If you have only one statement in a *block* (the body of a conditional or a loop or a method), you don't have to end the statement with a semi-colon.

Conditionals

```
if (EXPRESSION)
    STATEMENT
```

An expression in Java is pretty similar to a logical expression in any other language. One difference is that a logical *and* is written as **&&**, and an *or* is written as **||**.

STATEMENT above can be replaced with a single statement (like **a=12;**) or it can be *any number* of statements set up inside of *curly braces*—**{** and **}**.

```
if (EXPRESSION)
    THEN-STATEMENT
else
    ELSE-STATEMENT
```

Iteration

```
while (EXPRESSION)
    STATEMENT
```

There is a **break** statement for ending loops.

Probably the most confusing iteration structure in Java is the **for** loop. It really combines a specialized form of a **while** loop into a single statement.

```
for (INITIAL-EXPRESSION ; CONTINUING-CONDITION;
ITERATION-EXPRESSION)
    STATEMENT
```

A concrete example will help to make this structure make sense.

```
> for (int num = 1 ; num <= 10 ; num = num + 1)
    System.out.println(num);
1
2
3
4
5
6
7
8
9
10
```

The first thing that gets executed *before anything inside the loop* is the INITIAL-EXPRESSION. In our example, we're creating an integer variable num and setting it equal to 1. We'll then execute the loop, testing the CONTINUING-CONDITION before each time through the loop. In our example, we keep going as long as the variable num is less than or equal to 10. Finally, there's something that happens *after* each time through the loop – the ITERATION-EXPRESSION. In this example, we add one to num. The result is that we print out (using System.out.println, which is the same as print in many languages) the numbers 1 through 10. The expressions in the **for** loop can actually be several statements, separated by commas.

The phrase VARIABLE = VARIABLE + 1 is so common in Java that a short form has been created.

```
> for (int num = 1 ; num <= 10 ; num++)
    System.out.println(num);
```

Arrays

To declare an array, you specify the *type* of the elements of the array, then open and close *square brackets*. (In Java, all elements of an array have the same type.) Picture [] declares an array of type Picture. So Picture [] myarray; declares myarray to be a variable that can hold an array of Pictures.

To actually create the array, we might say something like **new Picture[5]**. This declares an array of five pictures. This does *not* create the pictures, though! Each of those have to be created separately. The indices will be 0 to 4 in this example. Java indices start with zero, so if an array has five elements, the maximum index is four. The fifth array reference (as seen below) will result in an error—the frequently visited **ArrayIndexOutOfBoundsException**.

```
> Picture [] myarray = new Picture[5];
> Picture background = new Picture(800,800);
> FileChooser.setMediaPath("D:/cs1316/mEDIAsources/");
> //Can load in any order
> myarray[1]=new Picture(FileChooser.getMediaPath("jungle.jpg"));
> myarray[0]=new Picture(FileChooser.getMediaPath("katie.jpg"));
> myarray[2]=new Picture(FileChooser.getMediaPath("barbara.jpg"));
> myarray[3]=new Picture(FileChooser.getMediaPath("flower1.jpg"));
> myarray[4]=new Picture(FileChooser.getMediaPath("flower2.jpg"));
> myarray[5]=new Picture(FileChooser.getMediaPath("butterfly.jpg"));
ArrayIndexOutOfBoundsException:
    at java.lang.reflect.Array.get(Native Method)
```

Strings

A *string* is not strictly an array, but it is similar to one. You can create a string from an array of characters. Characters are defined with single quotes (e.g., 'a') as opposed to double quotes (e.g., "a") which define strings.

```
> char characters[]={'B','a','r','b'}
> characters
[C@1ca209e
> String wife = new String(characters)
> wife
"Barb"
```

You cannot index a string with square brackets like an array. You can use the *substring* method on a string to retrieve individual characters (*substrings*) within the string. The *substring* method takes a starting and ending position in the string, starting with zero.

```
> String name = "Mark Guzdial";
> name
"Mark Guzdial"
> name[0]
```

```
Error: 'java.lang.String' is not an array
> name.substring(0,0)
"""
> name.substring(0,1)
"M"
> name.substring(1,1)
"""
> name.substring(1,2)
"a"
```

2.3 Using Java to Model the World

We have talked about the value of objects in modelling the world, and about the value of Java. In this section, we use Java to model some objects from our world. Let's consider the world of a student.

A Problem and Its Solution: What should I model?

When you model something in the world (real or virtual) as an object, you are asking yourself “What’s important about the thing that I am trying to model?” Typically, that question is answered by considering what you want from the model. Why are you doing this model? What’s important about it? What questions are you trying to answer?

If you are creating a model of students, then the question that you want to answer determines what you model and how you model it. If you want to create a model in order to create a course registration system, then you care about the students as people with a particular role. If you wanted to model students to answer questions about their health (e.g., how dormitory food impacts their liver, or how lack of sleep impacts their brains), then you want to model students as biological organisms with organs like livers.

For our purposes in this chapter, let’s imagine that we are modeling students in order to explore registration behavior. Given the previous, that means that we care about students as people. At the very least then, we want to define a class `Student` and a class `Person`.

A Problem and Its Solution: How much should I model?

Always try for the minimal model. Model as little as possible to answer your question. Models grow in complexity rapidly. The more attributes and variables that you model, the more that you have to worry about later. When you model, you are always asking yourself “Did I deal with all the relationships between all the variables in this model?” The more variables you model, the more relationships there are to consider. Model as little as

you need to answer your question.

We are going to define the class `Student` as a *subclass* of `Person`. There are several implications of that statement.

- Class `Person` is a *superclass* of `Student`.
- We also say that `Person` is the *parent class* of `Student`, and `Student` is the *child class* of `Person`.
- All *fields* or *instance variables* of the class `Person` are automatically in instances of the class `Student`. That does not mean that all those variables are accessible—some of them may be *private* which means that they are defined in the parent class, they are accessible in the parent class, but they are not accessible in the methods of the subclass.
- All *methods* in the superclass, `Person` are automatically usable in the subclass `Student`.
- We should be able to think about the subclass, `Student`, as being “a kind of” (sometimes shortened to *kind-of*) the superclass. A student is a kind of person—that’s true, so it’s a reasonable superclass-subclass relationship to set up.

Here’s an initial definition of the class `Person`.

Example Java Code: **Person class, starting place**

*Program
Example #1*

```
public class Person {
    public String name;
}
```

That’s enough to create an instance of class `Person` and use it. Here’s how it will look in DrJava:

```
> Person fred = new Person();
> fred.name
null
> fred.name = "Fred";
> fred.name
"Fred"
```

There’s an implication of this implementation of `Person`—we can change the name of the person. Should we be able to do this?

```
> fred.name = "Mabel";
> fred.name
"Mabel"
```

If we want to control name changes, we can make the field name private. To change or get the name, we can use *accessor* methods (sometimes called *getter* and *setter* methods because they let us get and set instance variables). These methods could check whether it's appropriate to set (or get) the variable.

Program Example #2

Example Java Code: Person, with a private name

```
public class Person {
    private String name;

    public void setName(String somename)
    {this.name = somename;}

    public String getName()
    {return this.name;}
}
```

How it works: The field name is now **private** meaning that only methods in this class can directly manipulate this variable. The method setName takes a String as input, then sets the name to that input. Since setName doesn't return any value, its return value is **void**—literally, nothing. The method getName does return the value of the variable name so it has a return value of String. Notice that both of these methods refer to **this.name**. This is a special variable meaning the object that has been told to getName or setName. The phrase **this.name** means “The variable name that is within the object **this**, the one that was told to execute this method.”

Now, we manipulate the field name using these methods—because we can't access the variable directly anymore. Within getName and setName in this example, the variable **this** means fred — they are two variable names referencing the exact same object.

```
> Person fred = new Person();
> fred.setName("Fred");
> fred.getName()
"Fred"
```

This works well for getting and setting the name. Let's consider what happens when we first create a new Person instance.

```
> Person barney = new Person();
> barney.getName()
null
```

Should “barney” have a **null** name? The value **null** means that this variable does not yet have a value. How do we define an instance of Person

such that it automatically has a value for its name? That would be the way that the real world works, that baby people get names at birth.

We can make that happen by defining a *constructor*. A constructor is a method that gets called when a new instance is created. A constructor has the same name as the class itself. It can take an input, so that we can create an object with certain values. Constructors don't *have* to take inputs—it's okay to just create an instance and have pre-defined values for variables.

Example Java Code: Person, with constructors

*Program
Example #3*

```
public class Person {
    private String name;

    public Person(){
        this.setName("Not-yet-named");
    }

    public Person(String name){
        this.setName(name);
    }
    public void setName(String somename)
    {this.name = somename;}

    public String getName()
    {return this.name;}
}
```

How it works: This version of class Person has two different constructors. One of them takes no inputs, and gives the name field a predefined, default value. The other takes one input—a new name to be given to the new object. It's okay to have multiple constructors as long as they can be distinguished by the inputs. Since one of these takes nothing as input, and the other takes a String, it's pretty clear which one we're calling when.

```
> Person barney = new Person("Barney")
// Here, we call the constructor that takes a string.
> barney.getName()
"Barney"
> Person wilma = new Person()
// Here, we call the constructor that doesn't take an input
> wilma.getName()
"Not-yet-named"
> wilma.setName("Wilma")
> wilma.getName()
```

```
"Wilma"
```

If we give Java some inputs to the constructor, but those inputs don't match any existing constructor, we get an error.

```
> Person agent99 = new Person(99)
java.lang.NoSuchMethodException: Person constructor
```

Discourse Rules for Java

In an American "Western" novel or movie, there are certain expectations. The hero carries a gun and rides a horse. The hero never uses a bazooka, and never flies on a unicorn. Yes, you can make a Western that has a hero with a bazooka or a unicorn, but it's considered a weird Western—you've broken some rules.

Those are *discourse rules*—the rules about how we interact in a certain genre or setting. They are not laws or rules that are enforced by some outside entity. They are about expectations.

There are discourse rules in all programming languages. Here are some for Java:

- Class names start with capital letters. All regular variables and method names start with lowercase letters. There are a couple special cases of variables (those that belong to a class, and those that are constant or `final`) that are also capitalized—those are rather rare.
- Class names are never plurals. If you want more than one instance of a class, you use an array or a list. The class name is not plural.
- Class names are typically nouns, not verbs.
- Instance variables and methods always start with lowercase letters.
- Methods should describe verbs—what objects know how to do.
- Accessors are typically named "set-" and "get-" the name of the field. To separate the word "set" or "get" from the variable name, the first letter of the variable name is capitalized.

Defining `toString`

What is an instance of Person? What do we get if we try to print its value?

```
> Person barney = new Person("Barney")
> barney
Person@63a721
```

That looks like Barney is a Person followed by some expletive or code. We can make the display of Barney look a little more reasonable using the method `toString`. When we try to print an object (by simply displaying its

value in the Interactions Pane in DrJava, or when printed from a program using `System.out.println`), the object is converted to a string. The method `toString` does that conversion. By providing a good `toString` method, we can make debugging easier—we can simply print the variable at important points in the code to see what the object's important values are.

Example Java Code: **toString method for Person**

*Program
Example #4*

```
public String toString(){
    return "Person named "+this.name;
}
```

How it works: The method `toString` returns a `String`, so we declare the return type of `toString` to be `String`. We can stick the words “Person named” before the actual name using the `+` operator—we call that *string concatenation*.

Now, we can immediately print Barney after creating him, and his value is reasonable and useful.

```
> Person barney = new Person("Barney")
> barney
Person named Barney
```

Defining Student as subclass of Person

Now, let's define our class `Student`. A student is a kind of person—we established that earlier. In Java, we say that `Student extends Person`—that means that `Student` is a subclass of `Person`. It means that a `Student` is everything that a `Person` is, with an extension in some way.

How should `Student` instances be different? Let's say that a `Student` has an identification number. That's not particularly insightful, but it is likely true.

Example Java Code: **Student class, initial version**

*Program
Example #5*

```
public class Student extends Person {
    private int idnum;

    public int getID(){ return idnum;}
    public void setID(int id) {idnum = id;}
}
```

* * *

How it works: Assuming that the identification number, idnum, will fit within the bounds of an integer **int**, this isn't a bad way to go. If the ID number might be too large, or if we would want to record dashes or spaces within it, using an **int** would be a bad model for the real identification number. Notice that we are also creating a getter and a setter for idnum.

From here, we can create instances of Student, but only by name—an inherited constructor from Person. There is no method Student(), so we can't create an instance without any input yet.

```
> Student betsy=new Student ("Betsy")
java.lang.NoSuchMethodException: Student constructor
> Student betsy = new Student()
> betsy.getName()
"Not-yet-named"
> betsy.setName ("Betsy")
> betsy.getName()
"Betsy"
> betsy.setID(999)
> betsy.getID()
999
```

If we print out Betsy, she'll tell us that she's a Person, not a Student. That's because the class Student inherits the `toString` method from Person.

```
> betsy
Person named Betsy
```

Let's define the class Student with reasonable constructors.

*Program
Example #6*

Example Java Code: **Class Student, with constructors**

```
public class Student extends Person {
    private int idnum;

    // Constructors
    public Student(){
        super(); //Call the parent's constructor
        idnum = -1;
    }
    public Student(String name){
        super(name);
        idnum = -1;
    }
    public int getID(){ return idnum;}
    public void setID(int id) {idnum = id;}
}
```

How it works: We create a default value for the idnum as `-1`. We also create accessors, `getID()` and `setID()`, for manipulating the identification number. Note that we still want to do whatever the Person constructors would do, so we call them with `super`.

Debugging Tip: Make the call to super first in a method

If you want to extend a method from a superclass, call `super` first. Let the superclass do whatever it would do. Then, extend the superclass's version with what you want the subclass to do.

Creating a `main()` method

How do we test our Java code? For those of us using DrJava¹, it's easy—we can simply type code in the Interactions Pane. What if you don't have an Interactions Pane? Then, you can test code by providing a `main()` method. There can be at most one main method in a class. It is often used to try out the class.

A main method is declared as `public static void main(String[] args)`, which is a real mouthful. That means (in brief—there is a better explanation later in the book):

- **public** means that this method can be accessed by any other class.
- **static** means that this method is known to the *class*, not just the instances. We can execute it without any instances having been created.
- **void** means that this method does not return anything.
- `String[] args` means that this method *could* be executed by running it from a command line, and any words on that line (e.g., filenames or options) would be passed on to the main method as elements in an array of strings named `args` (for “arguments”).

Example Java Code: **main() method for Person**

*Program
Example #7*

```
public static void main(String [] args){
    Person fred = new Person("Fred");
```

¹Or BlueJ, <http://www.bluej.org>

```

Person barney = new Person("Barney");

System.out.println("Fred is "+fred);
System.out.println("Barney is "+barney);
}

```

How it works: When this method is in the class Person, and we run the class (however your Java environment allows you to do this) or execute it from the command line, this method is executed. Two instances of class Person are created, and then both are printed to the console (or Interactions Pane, for DrJava). System.out.println prints the input to the console. When what is to be printed includes objects, like fred and barney above, they are converted to the type String (using the object's `toString` method) and displayed to the console.

When we execute this using the RUN button, we see:

```

Welcome to DrJava.
> java Person
Fred is Person named Fred
Barney is Person named Barney

```

Exploring Inheritance

A subclass inherits all the methods in the superclass—*unless* the subclass overrides it. If the subclass defines its own version of a method in the superclass, then the subclass version will be executed from an instance of the subclass. (An instance of the superclass will, of course, still execute the method of the superclass.)

Let's add a method to class Person that allows instances of Person to be friendly and greet people.

*Program
Example #8*

Example Java Code: **greet** method for Person

```

public void greet(){
    System.out.println("Hi! I am "+this.name);
}

```

Executing this method looks like this:

```

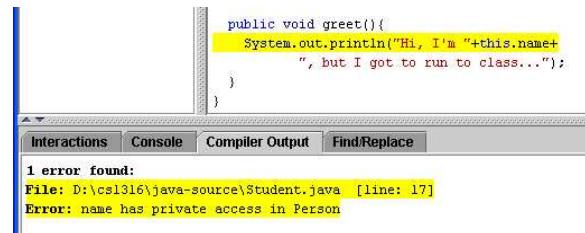
> Person bruce = new Person("Bruce")
> bruce.greet()
Hi! I am Bruce
> Person george = new Person("George W.")
> george.greet()

```

```
Hi! I am George W.
> Student krista = new Student("Krista")
> krista.greet()
Hi! I am Krista
```

Let's imagine that we create a greet method for class Student, too. We might try something like this:

```
public void greet(){
    System.out.println("Hi! I'm "+this.name+
        " but I got to run to class...");
```



Unfortunately, that will generate an error:

That error occurred because name is a **private** variable. The subclass, Student can't access the variable. Only the class that defined it, Person can access it. Instances of the class Student certainly *have* an name variable—they just can't access it directly. If we want to access the variable in Student, we have to use the accessors.

Example Java Code: **greet method for Student**

*Program
Example #9*

```
public void greet(){
    System.out.println("Hi, I'm "+this.getName()+
        " but I got to run to class...");
```

Now we get different effects when we ask a Person or Student to greet().
Welcome to DrJava.
> Person george = new Person("George W.")
> george.greet()
Hi! I am George W.
> Student krista = new Student("Krista")
> krista.greet()
Hi, I'm Krista, but I got to run to class...

It's worth thinking this through—what exactly happened when we executed krista.greet()?

- Krista knows that she is a Student—an instance of the class Student. Krista knows that if she can't greet, she can ask her parent to do it for her.
- She does know how to greet(), so she executes that method.
- But midway, “Uh-oh. I don’t know how to getName()!” So Krista asks her parent (who might have asked his parent, and so on, as necessary), who does know how to getName().

Let’s try one more experiment:

```
> Person fred = new Student ("Fred")
> Student mabel = new Person ("Mabel")
ClassCastException: mabel
```

Why did the first statement work, but the second one generated an error? Variables in Java always have a particular *type*. They can hold values that match that type. A Person variable fred can hold a Person instance. A variable of a given type can also hold instances of any *subclasses*. Thus, Person variable fred can hold an instance of class Student. However, a variable of a given type can *not* hold instances of any *superclasses*. Thus, the Student variable mabel cannot hold an instance of the class Person.

There are some subtle implications of putting an object of one type in a variable of another type. Consider the following example:

```
> fred.greet()
Hi, I'm Fred, but I got to run to class...
> fred.setID(999)
Error: No setID' method in 'Person'
> ((Student) fred).setID(999)
> ((Student) fred).getID()
999
```

Our Person variable fred knows how to greet()—of course it does, since both Person and Student classes know how to greet. Note that fred executes the *Student* method greet()—that also makes sense, since fred contains an instance of the class Student.

Here’s the tricky part: fred can’t setID(). You may be thinking, “But Fred’s a Student! Students know how to set their ID!” Yes, that’s true. The variable fred, though, is declared to be type Person. That means that Java is checking that fred is only asked to do things that a Person might be asked to do. A Person does now know how to set its ID. We can tell Java, “It’s okay—fred contains a Student” by *casting*. When we say ((Student) fred), we are telling Java to treat fred like a Student and let it execute Student methods. Then it works.

2.4 Manipulating Pictures in Java

We can get *file paths* using FileChooser and its method pickAFile(). FileChooser is a **class** in Java. The method pickAFile() is special in that it's known to the class, not to objects created from that class (*instances*). It's called a **static** or *class method*. To access that method in that class, we use *dot notation*: *Classname.methodname()*.

```
> FileChooser.pickAFile()  
"/Users/guzdial/cs1316/MediaSources/beach-smaller.jpg"
```

In the array example at the end of the last section, we see the use of FileChooser.setMediaPath and FileChooser.getMediaPath. The method setMediaPath takes as input a directory—note that it *must* end in a slash. (You can always use forward slashes here—it'll work right on any platform.) The method getMediaPath takes a filename, then returns the directory concatenated in front of it. So FileChooser.getMediaPath("jungle.jpg") actually returns "D:/cs1316/mEDIAsources/jungle.jpg". You *only need to use setMediaPath once!* It gets stored in a file on your computer, so that all your code that accesses getMediaPath will just work. This makes it easier to move your code around. New pictures don't have any value – they're **null**.

```
> Picture p;  
> p  
null
```

Debugging Tip: Did you get your Picture?

If you got an error as soon as you typed Picture p; there are two main possibilities.

- All the Java files we provide you are in source form. You need to compile them to use them. Open Picture.java and click COMPILE ALL. If you get additional errors about classes not found, open those files and compile them, too.
- You might not have your PREFERENCES set up correctly. If Java cant find Picture, you cant use it.

Debugging Tip: Semi-colons or not?

In the DrJava Interactions Pane, you don't have to end your lines with a semi-colon (;). If you don't, you're saying to DrJava "Evaluate this, and show me the result." If you do, you're saying "Treat this like a line of

code, just as if it were in the Code Pane.” Leaving it off is a useful debugging technique—it shows you what Java thinks that variable or expression means. But be careful—you *must* have semi-colons in your Code Pane!

To make a new picture, we use the code (you might guess this one) **new Picture()**. Then we’ll have the picture show itself by telling it (using dot notation) to **show()** (Figure 2.1).

```
> p = new Picture("/Users/guzdial/cs1316/MediaSources/beach-smaller.jpg");
> p
Picture, filename /Users/guzdial/cs1316/MediaSources/beach-smaller.jpg height 360 width 480
> p.show()
```

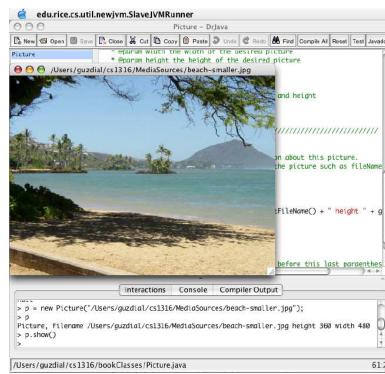


Figure 2.1: Showing a picture

The variable **p** in this example has the type **Picture**. That means that it can only hold pictures. We can assign it to new pictures, but we can’t assign it to a **Sound** or an **int**. We also can’t re-declare **p**.

Common Bug: One declaration per scope

Within a given *scope* (e.g., any set of curly braces, such as a single method, or the Interactions Pane in DrJava between compilations or reset), a variable can be declared once and only once. Another declaration is an error. You can *use* the variable as much as you might like after declaration, but you can only *declare* it once.

After the scope in which it was declared, the variable ceases to exist. So, if you declare a variable inside the curly braces of a **for** or **while** loop, it will not be available *after* the end curly brace.

* * *

Common Bug: Java may be hidden on Macintosh

When you open windows or pop-up file choosers on a Macintosh, they will appear in a separate “Java” application. You may have to find it from the Dock to see it.

The downside of types is that, if you need a variable, you need to create it. In general, that's not a big deal. In specific cases, it means that you have to plan ahead. Let's say that you want a variable to be a pixel (class Pixel) that you're going to assign inside a loop to each pixel in a list of pixels. In that case, the declaration of the variable *has* to be *before* the loop. If the declaration were inside the loop, you'd be re-creating the variable, which Java doesn't allow.

To create an array of pixels, we use the notation `Pixels []`. The square brackets are used in Java to index an array. In this notation, the open-close brackets means “an array of indeterminate size.”

Here's an example of increasing the red in each pixel of a picture by doubling (Figure 2.2).

```
> Pixel px;
> int index = 0;
> Pixel [] mypixels = p.getPixels();
> while (index < mypixels.length)
{
    px = mypixels[index];
    px.setRed(px.getRed() * 2);
    index = index + 1;
}
> p.show()
```

How would we put this process in a file, something that we could use for *any* picture? If we want *any* picture to be able to increase the amount of red, we need to edit the class Picture in the file `Picture.java` and add a new method, maybe named `increaseRed`.

Here's what we would want to type in. The special variable `this` will represent the Picture instance that is being asked to increase red. (In Python or Smalltalk, `this` is typically called `self`.)

Example Java Code: **Method to increase red in Picture**

*Program
Example #10*

```
2  /**
 * Method to increase the red in a picture.
 */
```



Figure 2.2: Doubling the amount of red in a picture

```

4   public void increaseRed()
5   {
6       Pixel px;
7       int index = 0;
8       Pixel [] mypixels = this.getPixels();
9       while (index < mypixels.length)
10      {
11          px = mypixels[index];
12          px.setRed(px.getRed()*2);
13          index = index + 1;
14      }
15  }

```

How it works:

- The notation `/*` begins a comment in Java – stuff that the compiler will ignore. The notation `*/` ends the comment.
- We have to declare methods just as we do variables! The term **public** means that anyone can use this method. (Why would we do otherwise? Why would we want a method to be **private**? We'll start explaining that next chapter.) The term **void** means “this is a method that doesn't return anything—don't expect the return value to have any particular type, then.”

Once we type this method into the bottom of class Picture, we can press the **COMPILE ALL** button. If there are no errors, we can test our new method. When you compile your code, the objects and variables you had in the Interactions Pane disappear. You'll have to recreate the objects you want.

* * *

Making It Work Tip: The command history isn't reset!

Though you lose the variables and objects after a compilation, the history of all commands you typed in DrJava is still there. Just hit up-arrow to get to previous commands, then hit return to execute them again.

You can see how this works in Figure 2.3.

```
> Picture p = new Picture(FileChooser.pickAFile());
> p.increaseRed()
> p.show()
```

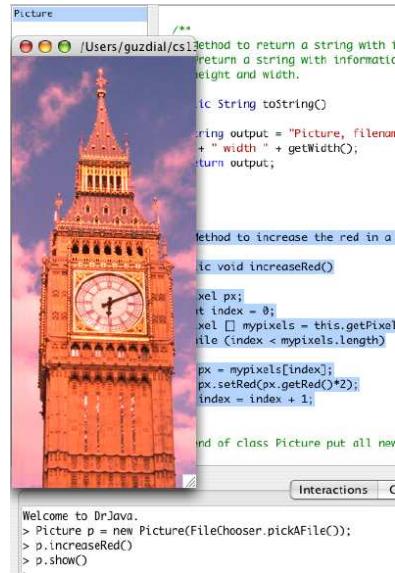


Figure 2.3: Doubling the amount of red using our increaseRed method

Later on, we're going to want to have characters moving to the left or to the right. We'll probably only want to create one of these (left or right), then flip it for the other side. Let's create the method for doing that. Notice that this method returns a *new* picture, not modifying the original one. Instead of being declared **void**, the flip method is declared **Picture**. It returns a picture. At the bottom of the method, you'll see that it does actually use **return** to return the target picture that we create inside the method. We'll see later that that's pretty useful, to create a new image rather than change the target picture. (Figure 2.4).

*Program
Example #11*

* * *

Example Java Code: Method to flip an image

```

1  /**
2  * Method to flip an image left-to-right
3  */
4  public Picture flip() {
5
6      Pixel currPixel;
7      Picture target = new Picture(this.getWidth(),this.getHeight());
8
9      for (int srcx = 0, trgx = getWidth()-1; srcx < getWidth();
10           srcx++, trgx--)
11     {
12         for (int srcy = 0, trgy = 0; srcy < getHeight();
13              srcy++, trgy++)
14         {
15             // get the current pixel
16             currPixel = this.getPixel(srcx,srcy);
17
18             /* copy the color of currPixel into target
19             */
20             target.getPixel(trgx,trgy).setColor(currPixel.getColor());
21         }
22     };
23     return target;
24 }
```

```

> Picture p = new Picture(FileChooser.pickAFile());
> p
Picture, filename D:\cs1316\MediaSources\guy1-left.jpg height 200
width 84
> Picture flipp = p.flip();
> flipp.show();
```

Common Bug: Width is the size, not the coordinate

Why did we subtract one from getWidth() (which defaults to `this.getWidth()`) to set the target X coordinate (`trgx`)? `getWidth()` returns the *number of pixels* across the picture. But the last *coordinate* in the row is one less than that, because Java starts all arrays at *zero*. Normal everyday counting starts with one, and that's what `getWidth()` reports.

* * *



Figure 2.4: Flipping our guy character—original (left) and flipped (right)

2.5 Exploring Sound in Java

We can create sounds in an analogous way to how we’re creating pictures.

```
> Sound s = new Sound(FileChooser.pickAFile());
> s.play();
```

How it works: Just as with pictures, we can create sounds as we declare them. FileChooser is an object that knows how to pickAFile(). That method puts up a file picker, then returns a string (or `null`, if the user hits CANCEL). Instances of the class Sound know how to play().

But what if we get it wrong?

```
> s.play()
> s.show()
Error: No 'show' method in 'Sound'
> Picture.play()
Error: No 'play' method in 'Picture'
> anotherpicture.play()
Error: Undefined class 'anotherpicture'
```

You can’t ask a Sound object to show()—it doesn’t know how to do that. Picture doesn’t know how to play() nor how to show()—it’s the *instances* (objects of that type or class) that know how to show(). The point of this example isn’t to show you Java’s barking messages, but to show you that there is no bite there. Type the wrong object name? Oh well—try again.

2.6 Exploring Music in Java

We will be working a lot with *MIDI* in this class. MIDI is a standard representation of musical information. It doesn't record sound. It records notes—when they're pressed, when they're released, how hard they're pressed, and what instrument is being pressed upon.

To use MIDI, we have to **import** some additional libraries. We're going to be using *JMusic* which is a wonderful Java music library that is excellent for manipulating MIDI.

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

How it works: First, you'll see a couple of **import** statements to bring in the basics of JMusic. Note is the name of the class that represents a musical note object. We're declaring a note variable named n1. We then create a Note instance. We don't need a filename—we're not reading a JPEG or WAV file. Instead, we simply need to know *which* note and for what duration (0.5 is an eighth note). That last line looks surprisingly like English, because it is. Any line starting with “//” is considered a *comment* and is ignored by Java. Table 2.1 summarizes the relationships between note numbers and more traditional keys and octaves.

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

But this isn't actually enough to play our note yet. A note isn't music, at least not to JMusic.

```
> Note n2=new Note(64,1.0);
```

```

> 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 -----

```



Figure 2.5: Just two notes

How it works: You'll see that we can't notate() a single note. We can, however, create a phrase that can take two notes (with different durations) Figure 2.5. A Phrase of JMusic knows how to addNote(). A View object knows how to notate() a phrase of music in standard Western music notation. From this window, we can actually play our music, change parameters (like the speed at which it plays), and shift instruments (e.g., to accordion or wind chimes or steel drums). We'll do more with the window later.

JMusic is a terrific example of using objects to *model*. JMusic is really modelling music.

- Note objects have tones and durations.
- Musical Phrase objects are collections of notes.
- A View object can present a musical phrase to us.

We can break this down in terms of what objects *know* and what they *do*.

	What instances of this class <i>know</i>	What instances of this class <i>do</i>
Note	A musical pitch and its duration.	(Nothing we've seen yet.)
Phrase	Notes in the phrase.	addNote(aNote)

Ex. 1 — Create a constructor for Student that takes a name and an ID.

3 Methods in Java: Manipulating Pictures

Chapter Learning Objectives

To make wildebeests charge over the ridge or villagers move around in the town square, we need to be able to manipulate pictures. We manipulate pictures using *methods*, the bits of behavior that classes store for use on their instances. That's the focus of this chapter.

The computer science goals for this chapter are:

- To be able to use Java with control over details, like expressions, understanding what **public** means, and being able to use correctly the ubiquitous **public static void main(String[] args)**.
- To compile and execute your code.
- To create a variety of different kinds of methods, including those that return values.
- To use JavaDoc.
- To chain method calls for compact, powerful expressions.
- To start our discussion of data structures with arrays.

The media learning goals for this chapter are:

- To extend what we can do with pictures.
- To combine methods for powerful picture manipulation.

3.1 Reviewing Java Basics

Assignment

As we saw in the last chapter, assignments come in form of
`CLASSNAME VARIABLE = EXPRESSION;`
or simply (if the variable has already been declared)
`VARIABLE = EXPRESSION;`

As mentioned, we can't declare variables twice in the same scope, and you can't use a variable of one type (or class) with an expression that results in an incompatible type. You can't assign a string to an `int` variable, for example,

Making It Work Tip: DrJava will declare for you—maybe not a good thing

If `snd` is an undeclared variable, DrJava will actually allow you to execute `snd = new Sound("D:/myfile.wav");`. DrJava is smart enough to figure out that you must mean for `snd` to be of type `Sound`. My suggestion: Don't do it. Be explicit in your type declarations. It will be too easy and forget when you're in Java Code Pane.

There are rules about *Java programming style* that you should know about. These aren't rules that, if broken, will result in a compiler error (usually). These are rules about how you write your code so that other Java programmers will understand what you're doing. We might call them *discourse rules*—they're the standard style or ways of talking in Java that Java programmers use.

- Always capitalize your class names.
- Never capitalize instance names, instance variable (sometimes called *fields*) names, or method names.
- You can use `mixedCaseToShowWordBreaks` in a long name.

All Java statements end with a semi-colon. You can insert as many spaces or returns (press the ENTER key) as you want in an expression—it's the semi-colon that Java uses to indicate the end of the line. Indentation doesn't matter at all in Java, unlike in Python. You can have no indentation at all in Java! Of course, no one, including you, will be able to make out what's going on in your program if you use no indentation at all. You probably should indent as we do here, where the body of a loop is indented deeper than the loop statement itself. DrJava will take care of that for you to make it easier to read, as will some other Java Integrated Development Environments (*IDEs*).

What goes in an expression? We can use `+`, `-`, `*` and `/` exactly as you used them in whatever your first programming language was. An expression that you've seen several times already is `new CLASSNAME()`, sometimes with inputs like `new Picture("C:/mypicture.jpg")`. You may have already noted that sometimes you create a new class with inputs, and sometimes you don't. Whether or not you need inputs depends on the *constructors* for the given class—those are methods that take inputs (optionally) and do something to set up the new object. For example, when we created

a new Note with a pitch and a duration, we passed the specification of the pitch and duration in as input to the class Note and they were assembled into the new object.

Java has a handful of shortcuts that you will see frequently. Because the phrase `x = x + 1` (where `x` could be any integer variable) occurs so often, we can abbreviate it `x++`. Because the phrase `y=y-1` occurs so often, it can be abbreviated as `y--`. There's a general form, too. The phrase `x = x + y` can be shortened to `x += y`. There are similar abbreviations `x *= y`, `x /= y`, and `x -= y`.

Arrays

An array is a homogenous (all the same type) linear collection of objects which are compacted together in memory. An array of integers, then, is a whole bunch of numbers (each without a decimal place), one right after another in memory. Being all scrunched together in memory is about as efficient in terms of memory space as they can be, and they can be accessed very quickly—going from one to the other is like leaving your house and going to the one next door.

An array is declared with square brackets `[]`. It turns out that the square brackets can come before or after the variable name in a declaration, so both of the below are correct Java statements (though clearly not both in the same scope!).

```
Pixel[] myPixels;  
Pixel myPixels[];
```

To access an array, we'll use square brackets again, e.g., `myPixels[0]`, which gets the first element in the array. Java begins numbering its indices at zero.

Conditionals

You've seen already that conditionals look like this:

```
if (LOGICAL-EXPRESSION)  
    then-statement;
```

As you would expect, the logical expression can be made up of the same logical operators you've used before: `<,>,<=,>=,==`. Note that `==` is the test for equivalence—it is not the assignment operator `=`. Depending on other languages you've learned, you may have used the words `and` and `or` for chaining together logical statements, but not in Java. In Java, a logical *and* is `&&`. A logical *or* is `||`.

The `then-statement` part can be one of two things. It could just be a simple statement ending in a semi-colon (e.g., `pixel.setRed(0);`). Or it could be any number of statements (each separated by semi-colons) inside of curly braces, like this:

```
if (pixel.getRed()<25)
{
    pixel.setRed(0);
    pixel.setBlue(120);
}
```

Do you need a semi-colon after the last curly brace? No, you don't have to, but if you do, it's not wrong. All of the below are correct conditionals in Java.

```
if (thisColor == myColor)
    setColor(thisPixel,newColor);
if (thisColor == myColor)
    {setColor(thisPixel,newColor);}
if (thisColor == myColor)
    {x = 12;
     setColor(thisPixel,newColor);};
```

We call those curly braces a *block*. A block is a single statement to Java. All the statements in the block together are considered just one statement. Thus, we can think of a Java statement ending in a semi-colon or a right curly brace (like an English sentence can end in “.” or “!” or “?”).

After the block for the *then* part (the part that gets executed “if” the logical expression is true, as in “if this, then that”) of the **if**, you can have the keyword **else**. The **else** keyword can be followed with another statement (or another block of statements) that will be executed if the logical expression is *false*. You can't use the **else** statement if you end the *then* block with a semi-colon though (like in the last **if** in the example above). Java gets confused if you do that, and thinks that you're trying to have an **else** without an **if**.

Iteration: While and For

A **while** loop looks like an **if**:

```
while (LOGICAL-EXPRESSION)
    statement;
```

But they're not at all similar. An **if** tests the expression *once* then possibly executes the *then* statement. A **while** tests, and if true, executes the statement—then tests again, and again executes the statement, and repeats until the statement is no longer true. That is, a **while** statement *iterates*.

We can use a **while** for addressing all the pixels in an image and setting all the red values to zero.

```
> p
Picture, filename D:/cs1316/MediaSources/Swan.jpg height 360 width
480
> Pixel [] mypixels = p.getPixels();
```

```
> int index = 0;
> while (index < mypixels.length)
    {mypixels[index].setRed(0);
     index++;};
```

How it works: Notice the reference to `mypixels.length` above. This is the standard way of getting an array's length. The expression `.length` isn't referring to a method. Instead it's referring to an *instance variable* or *field*. Every array knows an instance variable that provides its length, but each length is an instance variable unique to that instance. It's all the same name, but it's the right value for each array.

Recall that **for** loop is unusual. It lists *initialization* (something to be done before the loop starts), *continuing condition* (something to test at the top of each loop), and *an incrementing condition* (what to do after testing at the top of each loop). It's actually the same structure as in the programming languages C and C++. We can use a **for** loop to count from one value to another, just as you might use a **for** loop in Basic or Python. But you can also use the Java **for** loop to do a lot more, like walk through both the `x` and `y` values at once.

Here's the same example as the **while** loop above, but with a **for** loop:

```
> for (int i=0; i < mypixels.length ; i++)
    { mypixels[i].setRed(0);};
```

How it works: Our *initialization* part is declaring an integer (**int**) variable `i` and setting it equal to zero. Notice that `i` will *ONLY* exist within the **for** loop. On the line afterward, `i` won't exist—Java will complain about an undeclared variable. The *continuing condition* is `i < mypixels.length`. We keep going until `i` is equal to the length, and we *don't* execute when `i` is equal to the length. The *incrementing condition* is `i++`—increment `i` by one. What this does is to make `i` take on every value from 0 to `mypixels.length-1` (minus one because we stop when `i` *IS* the length), and execute the body of the loop—which sets red of the pixel at `i` equal to zero.

3.2 Java is about Classes and Methods

In Java, nearly everything is an object. Objects *know* things and *know how to do* things. It's the objects that do and know things—there aren't globally accessible functions like there are in many other languages like Python, C, or Java.

Programming in Java (and object-oriented languages, in general) is about defining what these objects know and what they know how to do. Each object belongs to a *class*. The class defines what the object knows (its *instance variables* or *fields*) and what it knows how to do (its *methods*).

For the `Picture` object, there is a file named `Picture.java` that defines the fields and methods that all `Picture` objects know (Figure 3.1). That file starts out with the line `public class Picture`. That starts out the definition

of the class Picture. Everything inside the curly braces after that line is the definition of the class Picture.

Typically, we define at the top of the file the instance variables that *all* objects of that class know. We define the methods that all the instance variables of that class below that, but still within the curly braces of the class definition. Each object (e.g., each Picture instance) has the same instance variables, but different values for those variables—e.g., each picture has a filename where it read its file from, but they can all be different files. But all objects know the same methods—they *know how to do* the same things.

Picture.java

```
public class Picture {
    Definitions for data in
    each Picture object go
    here.

    Each method goes
    inside here.

}
```

Figure 3.1: Structure of the Picture class defined in Picture.java

Debugging Tip: You change Picture.java

There should be one and only one `Picture.java` file. This means that you *have to* modify the file that we give you. If you rename it (say, `Picture-v2.java`), Java will just complain about the filename being incorrect. If you save `Picture.java` somewhere else, Java will get confused about two versions. Save a backup copy somewhere, and trust that it will be okay—you won’t damage the file too severely.

* * *

So what's this **public** statement about the class Picture? You might be wondering if there are other options, like discreet or celebrity. The statement **public** means that the class Picture is available for access by any other class. In general, *every* field or method can be **public**, or **protected** or **private**.

- **public** means that the class, field, or method is accessible by anyone. If there was a class with **public** fields, any other object could read those fields or change the values in them. Is that a good thing? Think about it—if objects represent (model) the real world, can you read any value in the world or change it? Not usually.
- **private** means that the field or method can *only* be accessed by the class containing that field or method. That's probably the best option for fields. Some methods might be **private**, but probably most would be **public**.
- **protected** is a middle ground that doesn't generally work. It means that the field or method is accessible by any class or its subclass—or any class belonging to the same *package*. “Package?” you say. “I haven't seen anything about packages!” Exactly—and if you don't deal with packages, **protected** data and methods are essentially **public**. Makes sense? Not to me either.

Pictures are about arrays and pixels

A picture is a two-dimensional array of pixels. An array is typically one-dimensional—there's just a collection of values, one right after the other. In a one-dimensional array, each element has a number associated with it called a *index variable*—think of it as the mailbox address for each element. A two-dimensional array is called a *matrix*—it has both height and width. We usually number the columns and the rows. With pictures, upper left hand corner is row number 0 (which we will refer to as the *y* index) and column number 0 (which we will refer to as the *x* index). The *y* values increase going down, and the *x* values increase going to the right.

What is in those columns and rows is *pixels*. A pixel is a picture element—a small dot of color in the picture or on the screen. Each dot is actually made up of a red component, a green component, and a blue component. Each of those components can have a value between 0 and 255. A red value of 0 is no red at all, and a red value of 255 is the maximum red. All the colors that we can make in a picture are made up of combinations of those red, green, and blue values, and each of those pixels sits at a specific (*x*, *y*) location in the picture.

A method for decreasing red

Let's explore a method in the class Picture to see how this all works. Here's the method to decrease the red in a picture, which would be inserted in Picture.java within the curly braces defining the class.

*Program
Example #12*

Example Java Code: **decreaseRed** in a Picture

```

1  /**
2   * Method to decrease the red by half in the current picture
3   */
4  public void decreaseRed()
5  {
6      Pixel pixel = null; // the current pixel
7      int redValue = 0;           // the amount of red
8
9      // get the array of pixels for this picture object
10     Pixel[] pixels = this.getPixels();
11
12     // start the index at 0
13     int index = 0;
14
15     // loop while the index is less than the length of the pixels array
16     while (index < pixels.length)
17     {
18         // get the current pixel at this index
19         pixel = pixels[index];
20         // get the red value at the pixel
21         redValue = pixel.getRed();
22         // set the red value to half what it was
23         redValue = (int) (redValue * 0.5);
24         // set the red for this pixel to the new value
25         pixel.setRed(redValue);
26         // increment the index
27         index++;
28     }
29 }
```

We would use this method like this:

```

> Picture mypic = new Picture("C:/intro-prog-java/mediasources/Barbara.jpg");
> mypic.show(); // Show the picture
> mypic.decreaseRed();
> mypic.repaint(); // Update the picture to show the changes
```

The first line creates a picture (**new Picture...**), declares a variable mypic to be a Picture, and assigns mypic to the new picture. We then show the

picture to get it to appear on the screen. The third line *calls* (or *invokes*) the method decreaseRed on the picture in mypic. The fourth line, mypic.repaint();, tells the picture to update itself on the screen. decreaseRed changes the pixels within memory, but repaint tells the window on the screen to update from memory.

How it works: Note that this method is declared as returning **void**—that means that this method doesn't return anything. It simply changes the object that it has been invoked upon. It's **public** so anyone (any object) can invoke it.

At the beginning of the method, we typically declare the variables that we will be using. We will be using a variable pixel to hold each and every pixel in the picture. It's okay to have a variable pixel whose class is Pixel—Java is case sensitive so it can figure out the variable names from the case names. We will use a variable named redValue to store the red value for each pixel before we change it (decrease it by 50%). It will be an integer (no decimal part) number, so we declare it **int**.

Making It Work Tip: Give variables values as you declare them

It's considered good practice to give initial values to the variables as you declare them. That way you know what is in there when you start, because you put it in there.

We give redValue an initial value of zero. We give pixel an initial value of **null**. **null** is the value that says, "This variable holds an object, but right now, it's holding nothing at all." **null** is the nothing-object.

The next line in decreaseRed is the statement that both declares the array of pixels pixels and assigns the name to an array. The array that we assign it to is what the method getPixels() returns. (That's the method getPixels which takes no inputs, but we still have to type () to tell Java that we want to call the method.) getPixels is a really useful method that returns all those pixels that are in the picture, but in linear, single-dimension array. It converts them from the matrix form to an array form, to make them easier to process. Of course, as an array, we lose the ability to know what row or column a pixel is in, but if we're doing the same thing to all pixels, we really don't care.

Notice that the object that we invoke getPixels() on is **this**. What's **this**? The object that we invoked decreaseRed on. In our example, we are calling decreaseRed on the picture in mypic (*barbara.jpg*). So **this** is the picture in *barbara.jpg*.

We are going to use a variable named index to keep track of which pixel we are currently working on in this method. The first index in any array is 0, so we start out index as 0. We next use a **while** loop to process each pixel in the picture. We want to keep going as long as index is less than the

number of pixels in the array. The number of elements in the array `pixels` is `pixels.length`.

`length` is not a method—it's a *field* or an *instance variable*. It's a value, not a behavior. We access it to get the number of elements in an array. Every array has the field `length`.

Common Bug: The maximum index is $length - 1$

A common mistake when working with arrays is to make the index go until `length`. The length is the *number* of elements in the array. The index numbers (the addresses on the array elements) start at 0, so the maximum index value is `length - 1`. If you try to access the element at index `length` you will get an error that says that you have an `OutOfBoundsException`—you've gone beyond the bounds of the array.

The body of the `while` loop in `decreaseRed` will execute repeatedly as long as `index` is less than `pixels.length`. Within the loop, we:

- Get the pixel at address `index` in the array `pixels` and make variable `pixel` refer to that pixel.
- Get the redness out of the pixel in variable `pixel` by calling the method `getRed()` and store it in `redValue`.
- We make `redValue` 50% smaller by setting it to 0.5 times itself. Notice that multiplying `redValue` by 0.5 could result in a value with a decimal point (think about odd values). But `redValue` can only be an integer, a value with no decimal point. So, we have to force the return value into an integer. We do that we *cast* the value into an integer, i.e., `int`. By saying “`(int)`” before the value we are casting (“`(redValue * 0.5)`”), we turn it into an integer. There's no rounding involved—any decimal part simply gets hacked off.
- We then store the new `redValue` back into the pixel with `setRed(redValue)`. That is, we invoke the method `setRed` on the pixel in the variable `pixel` with the input `redValue`: `pixel.setRed(redValue);`
- Finally, we increment `index` with `index++`. That makes `index` point toward the next value in the array, all set for the test at the top of the `while` and the next iteration through the body of the array.

Method with an input

What if we wanted to decrease red by an *amount*, not always 50%? We could do that by calling `decreaseRed` with an input value. Now, the code we just walked through cannot take an input. Here's one that can.

* * *

Example Java Code: **decreaseRed with an input**

*Program
Example #13*

```
/**  
 * Method to decrease the red by an amount  
 * @param amount the amount to change the red by  
 */  
public void decreaseRed(double amount)  
{  
    Pixel[] pixels = this.getPixels();  
    Pixel p = null;  
    int value = 0;  
  
    // loop through all the pixels  
    for (int i = 0; i < pixels.length; i++)  
    {  
        // get the current pixel  
        p = pixels[i];  
        // get the value  
        value = p.getRed();  
        // set the red value the passed amount time what it was  
        p.setRed((int) (value * amount));  
    }  
}
```

How it works: This version of decreaseRed has something within the parentheses after the method name—it's the amount to multiply each pixel value by. We have to tell Java the type of the input value. It's **double**, meaning that it's a double-precision value, i.e., it can have a decimal point.

In this method, we use a **for** loop. A **for** loop has three parts to it:

- An initialization part, a part that occurs before the loop begins. Here, that's **int** i = 0. There are semi-colons before each part.
- A test part—what has to be true for the loop to continue. Here, it's that we have more pixels left, i.e., i < pixels.length.
- An iteration part, something to do each time through the loop. Here, it's i++, go to the next index value (in variable i).

The rest of this loop is much the same as the other. *It's perfectly okay with Java to have both versions of decreaseRed in the Picture class at once.* Java can tell the difference between the version that takes no inputs and the one that takes one numeric input. We call the name, the number of inputs, and the types of the inputs as the *method signature*. As long as

two methods had different method signatures, it's okay for them to both have the same name.

Notice the odd comment at the start of the method, the one with the @param notation in it. That specialized form of comments is what's used to produce the documentation called *JavaDoc*. The JavaDoc for the media classes provided with this book is in the `doc` folder inside the `media-sources` folder. These Web pages explain all the methods, their inputs, how they're used, and so on (Figure 3.2). We sometimes refer to this information as the *API* or *Application Program Interface*. The content comes from these specialized comments in the Java files.

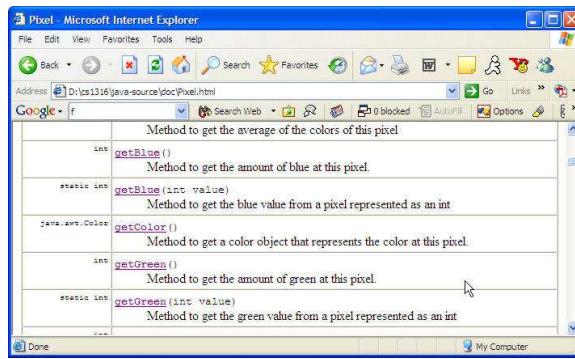


Figure 3.2: Part of the JavaDoc page for the Pixel class

Now, if you look at the class `Picture`, you may be surprised to see that it doesn't know very much at all. Certainly, important methods like `show` and `repaint` are missing. Where are they? If you edit the class `Picture` (in the file `Picture.java`), you'll see that it says:

```
public class Picture extends SimplePicture
```

That means that some of what the class `Picture` understands is actually defined in the class `SimplePicture`. Class `Picture` *extends* class `SimplePicture`. `Picture` is a subclass of `SimplePicture`, which means that class `Picture` *inherits* everything that `SimplePicture` has and knows. It's `SimplePicture` that actually knows about pixels and how pictures are stored, and it's `SimplePicture` that knows how to show and repaint pictures. `Picture` inherits all of that by being a *subclass* of `SimplePicture`.

Why do that? Why make `Picture` so relatively dumb? There are lots of reasons for using inheritance. The one we're using here is *information hiding*. Open up `SimplePicture.java` and take a peek at it. It's pretty technical and sophisticated code, filled with `BufferedImage` objects and references to `Graphics` contexts. We *want* you to edit the `Picture` class, to change methods and add new methods. We want that code to be understandable, so we hide the stuff that is hard to understand in `SimplePicture`.

3.3 Methods that return something: Compositing images

If we're going to make wildebeests or villagers, we need some way of getting those images onto a frame. Here are some methods to do it. Along the way, we will create methods that return new pictures—a very useful feature for creating more complex pictures.

Example Java Code: **Method to compose this picture into a target** *Program Example #14*

```
/*
 * Method to compose this picture onto target
 * at a given point.
 * @param target the picture onto which we chromakey this picture
 * @param targetx target X position to start at
 * @param targety target Y position to start at
 */
public void compose(Picture target, int targetx, int targety)
{
    Pixel currPixel = null;
    Pixel newPixel = null;

    // loop through the columns
    for (int srcx=0, trgx = targetx; srcx < this.getWidth();
         srcx++, trgx++)
    {
        // loop through the rows
        for (int srcy=0, trgy=targety; srcy < this.getHeight();
             srcy++, trgy++)
        {
            // get the current pixel
            currPixel = this.getPixel(srcx, srcy);

            /* copy the color of currPixel into target,
             * but only if it'll fit.
             */
            if (trgx < target.getWidth() && trgy < target.getHeight())
            {
                newPixel = target.getPixel(trgx, trgy);
                newPixel.setColor(currPixel.getColor());
            }
        }
    }
}
```

* * *

Using this method, we can then compose the guy into the jungle like this (Figure 3.3).

```
> Picture p = new Picture(FileChooser.getMediaPath("guy1-left.jpg"));
> Picture bg = new Picture(FileChooser.getMediaPath("jungle.jpg"));
> p.compose(bg, 65, 250);
> bg.show();
> bg.write("D:\\cs1316\\jungle-composed-with-guy.jpg")
```



Figure 3.3: Composing the guy into the jungle

How it works: Basically what happens in this method is that we copy the colors out of the source picture, **this**, and set the pixels in the target to that color. That makes this picture appear in the target.

The `compose` method takes three inputs. The first one is a picture onto which the **this** picture (the one that the method is being invoked upon) will be composed. Think of the input as a canvas onto which we paint this picture. The other two inputs are the *x* and *y* position where we start painting the picture—the variables `targetx` and `targety` are integers that define where the upper left hand corner of this picture appears.

We don't have the luxury of using `getPixels` this time. We need to know which rows and columns are which, so that we make sure that we copy them all into the right places. We don't want this picture (our source) to show up as one long line of pixels—we want the rows and columns in the source to show up as rows and columns in the target.

We are going to need two **for** loops. One is going to take care of the *x* indices, and the other will take care of the *y* indices. We use two variables for keeping track of the current pixel in **this** picture, our source. Those variables are named `srcx` and `srcy`. We use two other variables for keeping track of the current location in the target, `trgx` and `trgy`. The trick to a composition is to always increment `srcx` and `trgx` together (so that we're talking about columns in the source and the target at the same time), and `srcy` and `trgy` together (so that the rows are also in sync). You don't want

to start a new row in the source but not the target, else the picture won't look right when composed.

To keep them in synch, we use a **for** loop where we move a couple of expressions in each part. Let's look at the first one in detail.

```
for (int srcx=0, trgx = targetx; srcx < this.getWidth();  
     srcx++, trgx++)
```

- In the initialization part, we declare srcx and set it equal to zero, then declare trgx and have it start out as the input targetx. Notice that declaring variables here is *the same as* (for the **for** loop) declaring them inside the curly braces of the **for** loop's block. This means that those variables *only* exist in this block—you can't access them after the class ends.
- In the testing part, we keep going as long as we have more columns of pixels to process in the source—that is, as long as srcx is less than the maximum width of this picture, `this.getWidth()`.
- In the increment part, we increment srcx and trgx together.

Common Bug: Don't try to change the input variables

You might be wondering why we copied targetx into trgx in the compose method. While it's perfectly okay to use methods on input objects (as we do in `compose()` when we get pixels from the target), and maybe change the object that way, don't try to add or subtract the values passed in. It's complicated why it doesn't work, or how it does work in some ways. It's best just to use them as variables you can *read* and *call methods on*, but not *change*.

The body of the loop essentially gets the pixel from the source, gets the pixel from the target, and sets the color of the target pixel to the color of the source pixel. There is one other interesting statement to look at:

```
if (trgx < target.getWidth() && trgy < target.getHeight())
```

What happens if you have a really wide source picture and you try to compose it at the far right edge of the target? You can't fit all the pixels, of course. But if you write code that *tries* to access pixels beyond the edge of the target picture, you will get an error about `OutOfBoundsException`. This statement prevents that.

The conditional says that we *only* get the target pixel and set its color, if the *x* and *y* values that we're going to access are less than the maximum

width of the target and the the maximum height of the target. We stay well inside the boundary of the picture that way¹.

So far, we've only seen methods that return **void**. We get some amazing expressive power by combining methods that return other objects. Below is an example of how we use the methods in class Picture to scale a picture larger (or smaller).

```
> // Make a picture from a file selected by the user
> Picture doll = new Picture(FileChooser.pickAFile());
> Picture bigdoll = doll.scale(2.0);
> bigdoll.show();
> bigdoll.write("bigdoll.jpg"); //Store the new picture to a new
file
```

Program
Example #15

Example Java Code: **Method for Picture to scale by a factor**

```

1   /**
2    * Method to scale the picture by a factor, and return the result
3    * @param scale factor to scale by (1.0 stays the same,
4    *             0.5 decreases each side by 0.5, 2.0 doubles each side)
5    * @return the scaled picture
6    */
7   public Picture scale(double factor)
8   {
9       Pixel sourcePixel, targetPixel;
10      Picture canvas = new Picture(
11          (int) (factor*this.getWidth())+1,
12          (int) (factor*this.getHeight())+1);
13      // loop through the columns
14      for (double sourceX = 0, targetX=0;
15           sourceX < this.getWidth();
16           sourceX+=(1/factor), targetX++)
17      {
18          // loop through the rows
19          for (double sourceY=0, targetY=0;
20               sourceY < this.getHeight();
21               sourceY+=(1/factor), targetY++)
22          {
23              sourcePixel = this.getPixel((int) sourceX,(int) sourceY);
24              targetPixel = canvas.getPixel((int) targetX, (int) targetY);
25              targetPixel.setColor(sourcePixel.getColor());
26          }
27      }
28      return canvas;
}
```

¹Of course, if you try to compose to the *left* of the picture, or *above* it, by using negative starting index values, you will get an exception still.

How it works: The method scale takes as input the amount to scale the picture `this`. This method is declared type Picture, instead of `void`—scale returns a picture.

The basic process of scaling isn't too complicated. If we have a picture and want it to fit into a smaller space, we have to lose some pixels—we simply can't fit all the pixels in. (All pixels are basically the same size, for our purposes.) One way of doing that is to skip, say, every other pixel, by skipping every other row and column. We do that by adding two to each index instead of incrementing by one each time through the loop. That reduces the size of the picture by 50% in each dimension.

What if we want to scale *up* a picture to fill a large space? Well, we have to duplicate some pixels. Think about what happens if we add 0.5 to the index variable (either for *x* or *y*) each time through the loop. The values that the index variable will take will be 0, 0.5, 1.0, 1.5, 2.0, 2.5, and so on. But the index variable can only be an integer, so we'd chop off the decimal. The result is 0, 0, 1, 1, 2, 2, and so on. By adding 0.5 to the index variable, we end up taking each position twice, thus doubling the size of the picture.

Now, what if we want a different sizing—increase by 30% or decrease by 25%? That's where the factor comes in as the input to scale. If you want a factor of 0.25, you want the new picture to be 1/4 of the original picture in each dimension. So what do you add to the index variable? It turns out that $1/factor$ works quite nice. $1/0.25$ is 4, which is a good index increment to get 0.25 of the size.

The scale method starts out by *creating* a target picture. The picture is sized to be the scaling factor times the height and width—so the target will be bigger if the scaling factor is over 1.0, and smaller if it is less. As we can see here, new instances of the class Picture can be created by filename *or* by specifying the height and width of the picture. The returned picture is blank. We add one to deal with off-by-one errors on oddly sized pictures.

The tricky part of this method is the `for` loops.

```
for (double sourceX = 0, targetX=0;  
     sourceX < this.getWidth();  
     sourceX+=(1/factor), targetX++)
```

Like in compose, we're manipulation two variables at once in this `for` loop. We're using `double` variables to store the indices, so that we can add a $1/factor$ to them and have them work, even if $1/factor$ isn't an integer. Again, we start out at zero, and keep going as long as there are columns (or rows, for the *y* variable) to process. The increment part has us adding one to `targetX` but doing `sourceX += (1/factor)` for the `sourceX` variable. This is a shortcut that is equivalent to `sourceX = sourceX + (1/factor)`.

When we use the index variables, we cast them to integers, which removes the floating point part.

```
sourcePixel = this.getPixel((int) sourceX, (int) sourceY);
```

At the very end of this method, we **return** the newly created picture. The power of returning a new picture is that we can now do a lot of manipulation of pictures with opening up only a few pictures and *never changing those original pictures*. Consider the below which creates a mini-collage by creating a new blank picture (by asking for a **new** Picture with a height and width as inputs to the constructor, instead of a filename) then composing pictures onto it, scaled at various amounts (Figure 3.4).

```
> Picture blank = new Picture(600, 600);
> Picture swan = new Picture("C:/cs1316/MediaSources/swan.jpg");
> Picture rose = new Picture("C:/cs1316/MediaSources/rose.jpg");
> rose.scale(0.5).compose(blank, 10, 10);
> rose.scale(0.75).compose(blank, 300, 300);
> swan.scale(1.25).compose(blank, 0, 400);
> blank.show();
```

What's going on here? How can we *cascade* methods like this? It's because *all* pictures understand the same methods, whether they were created from a file or created from nothing. So, the *scaled* rose understands compose just as well as the rose itself does.

This is a method that's understood by Pictures. Why, that's what *scale* returns!

This returns a Picture—and *rose* is not changed!

rose.scale(0.5).compose(blank,10,10);

Sometimes you don't want to show the result. You may prefer to explore it, which allows you to check colors and get exact *x* and *y* coordinates for parts of the picture. We can explore pictures to figure out their sizes and where we want to compose them (Figure 3.5).

We also see in this example that we can use `setMediaPath` and `getMediaPath` to make it easier to get the pieces by basename instead of typing out the whole file path each time. `FileChooser.setMediaPath` remembers the path that you specify as the location of your media². `FileChooser.getMediaPath` then recalls that path and sticks it *before* the base file name that you provide as input.

²The path is actually stored as a file on your disk, so you should only have to do `setMediaPath` once on a single computer

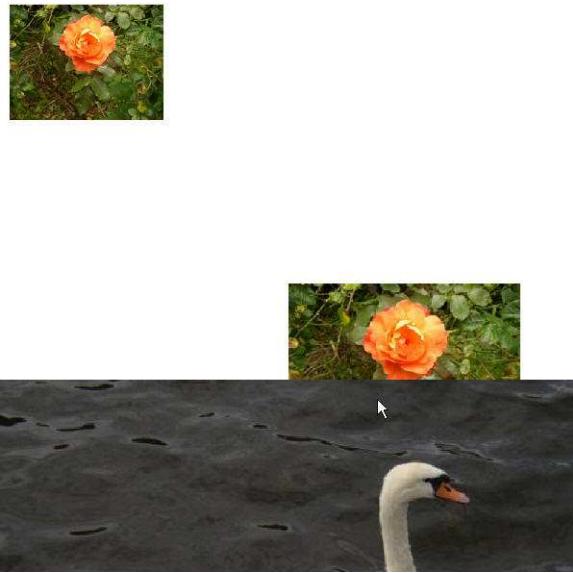


Figure 3.4: Mini-collage created with scale and compose

```
> FileChooser.setMediaPath("C:/cs1316/mediasources/");
> Picture bg = new Picture(FileChooser.getMediaPath("jungle.jpg"));
> bg.explore();
> p.explore();
```

Composing by Chromakey

Chromakey is the video technique by which a meteorologist on our television screen gestures to show a storm coming in from the East, and we see the meteorologist in front of a map (perhaps moving) where the storm is clearly visible in the East next to the meteorologist's hand. The reality is that the meteorologist is standing in front of a blue or green screen. The chromakey algorithm replaces all the blue or green pixels in the picture with pixels of a different background, effectively changing where it looks like the meteorologist is standing. Since the background pixels won't be all the *exact* same blue or green (due to lighting and other factors), we usually use a *threshold* value. If the blue or green is "close enough" (that is, within a threshold distance from our comparison blue or green color), we swap the background color.

There are a couple of different chromakey methods in Picture. `chromakey()` lets you input the color for the background and a threshold for how close you want the color to be. `bluescreen()` assumes that the background is blue, and looks for more blue than red or green (Figure 3.6. If there's a lot of blue

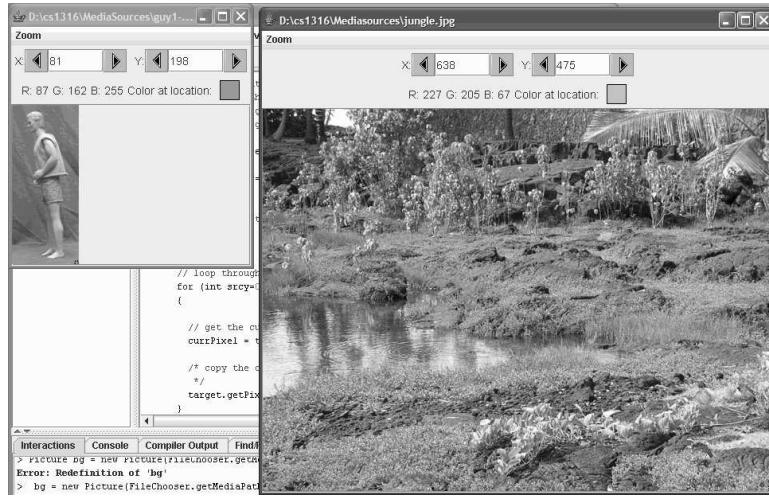


Figure 3.5: Using the explore method to see the sizes of the guy and the jungle

in the character, it's hard to get a threshold to work right. It's the same reason that the meteorologist doesn't wear blue or green clothes—we'd see right through them!

```

> Picture p = new Picture(FileChooser.getMediaPath("monster-right1.jpg"));
> Picture bg = new Picture(FileChooser.getMediaPath("jungle.jpg"));
> p.bluescreen(bg,65,250);
> import java.awt.*;
> p.chromakey(bg,Color.blue,100,165,200);
> p.chromakey(bg,Color.blue,200,26,250);
> bg.show();
> bg.write("D:/cs1316/jungle-with-monster.jpg");

```

Program Example #16

Example Java Code: Methods for general chromakey and bluescreen

```

/*
 * Method to do chromakey using an input color for background
 * at a given point.
 * @param target the picture onto which we chromakey this picture
 * @param bgcolor the color to make transparent
 * @param threshold within this distance from bgcolor, make transparent
 * @param targetx target X position to start at
 * @param targety target Y position to start at
*/

```



Figure 3.6: Chromakeying the monster into the jungle using different levels of bluescreening

```

10  public void chromakey(Picture target, Color bgcolor, int threshold,
11      int targetx, int targety)
12  {
13      Pixel currPixel = null;
14      Pixel newPixel = null;
15
16      // loop through the columns
17      for (int srcx=0, trgx=targetx;
18          srcx<getWidth() && trgx<target.getWidth();
19          srcx++, trgx++)
20      {
21
22          // loop through the rows
23          for (int srcy=0, trgy=targety;
24              srcy<getHeight() && trgy<target.getHeight();
25              srcy++, trgy++)
26          {
27
28              // get the current pixel
29              currPixel = this.getPixel(srcx,srcy);
30
31              /* if the color at the current pixel is within threshold of
32              * the input color, then don't copy the pixel
33              */
34              if (currPixel.colorDistance(bgcolor)>threshold)
35              {
36                  target.getPixel(trgx,trgy).setColor(currPixel.getColor());
37              }
38          }

```

```

        }
    }

42   /**
43   * Method to do chromakey assuming blue background for background
44   * at a given point.
45   * @param target the picture onto which we chromakey this picture
46   * @param targetx target X position to start at
47   * @param targety target Y position to start at
48   */
49   public void bluescreen(Picture target,
50                         int targetx, int targety)
51   {
52     Pixel currPixel = null;
53     Pixel newPixel = null;
54
55     // loop through the columns
56     for (int srcx=0, trgx=targetx;
57          srcx<getWidth() && trgx<target.getWidth();
58          srcx++, trgx++)
59     {
60
61       // loop through the rows
62       for (int srcy=0, trgy=targety;
63            srcy<getHeight() && trgy<target.getHeight();
64            srcy++, trgy++)
65     {
66
67       // get the current pixel
68       currPixel = this.getPixel(srcx, srcy);
69
70       /* if the color at the current pixel mostly blue (blue value is
71      * greater than red and green combined), then don't copy pixel
72      */
73       if (currPixel.getRed() + currPixel.getGreen() > currPixel.getBlue())
74     {
75         target.getPixel(trgx, trgy).setColor(currPixel.getColor());
76     }
77   }
78 }
}

```

3.4 Creating classes that do something

So far, we have created methods in the class Picture that know *how* to do something, but we actually *do* things with statements in the Interactions Pane. How do we get a Java class to *do* something? We use a particular

method that declares itself to be the main thing that this class *does*. You declare a method like this:

```
public static void main(String [] args){
    //code goes here
}
```

The code that goes inside a main method is exactly like what goes in an Interactions Pane. For example, here's a class that the *only* thing it does is to create a mini-collage.

Example Java Code: A **public static void main** in a class

*Program
Example #17*

```
1 public class MyPicture {
2     public static void main(String args[]){
3         Picture canvas = new Picture(600,600);
4         Picture swan = new Picture("C:/cs1316/MediaSources/swan.jpg");
5         Picture rose = new Picture("C:/cs1316/MediaSources/rose.jpg");
6         Picture turtle = new Picture("C:/cs1316/MediaSources/turtle.jpg");
7
8         swan.scale(0.5).compose(canvas,10,10);
9         swan.scale(0.5).compose(canvas,350,350);
10        swan.flip().scale(0.5).compose(canvas,10,350);
11        swan.flip().scale(0.5).compose(canvas,350,10);
12        rose.scale(0.25).compose(canvas,200,200);
13        turtle.scale(2.0).compose(canvas,10,200);
14        canvas.show();
15    }
16 }
17 }
```

The seemingly-magical incantation **public static void** main(**String** [] args) will be explained more later, but we can talk about it briefly now.

- **public** means that it's a method that any other class can access.
- **static** means that this is a method accessible from the *class*. We don't need to create instances of this class in order to run the main method.
- **void** means that the main method doesn't return anything.
- **String[] args** means that the main method can actually take inputs *from the command line*. You can run a main method from the command line by typing the command java and the class name, e.g. java MyPicture (presuming that you have Java installed!).

To run a main method from within DrJava, use function key F2. That's the same as using RUN DOCUMENT'S MAIN METHOD from the TOOLS menu (Figure 3.7).



Figure 3.7: Run the main method from DrJava

A main method is not very object-oriented – it's not about defining what an object knows or what it can do. But it is pretty useful.

4 Objects as Agents: Manipulating Turtles

Chapter Learning Objectives

We are going to model our wildebeests and villagers as *agents*—objects that behave independent of each other, seemingly simultaneously, with a graphical (visible) representation. Turtles are an old computational idea that are useful for understanding agents behavior. They are also a powerful tool for understanding object-oriented programming. In this chapter, we learn about turtles in order to simply animations and simulations later.

The computer science goals for this chapter are:

- To introduce some of the history of object-oriented programming, from Logo (and turtles) to Smalltalk.
- To generalize an understanding of objects, from Pictures to Turtles.
- To understand better cascading methods.
- To introduce some basic list manipulation ideas, e.g., that nodes are different *objects*.

The media learning goals for this chapter are:

- To create animations using a FrameSequence.
- To find another technique for composing pictures.
- To use a simple technique for rotating pictures.

4.1 Turtles: An Early Computational Object

In the mid-1960's, Seymour Papert at MIT and Wally Feurzeig and Danny Bobrow at BBN Labs were exploring educational programming by children. That was a radical idea at the time. Computers were large, expensive devices which were shared by multiple people at once. Some found the thought of giving up precious computing time for use by 10 or 11 year old children to be ludicrous. But Papert, Feurzeig, and Bobrow believed

that the activity of programming was a great context for learning all kinds of things, including learning about higher-order thinking skills, like planning and debugging. They created the programming language *Logo* as a simplified tool for children.

The most common interaction with computers in those days was through teletypes—large machines with big clunky keys that printed all output to a roll of paper, like a big cash register receipt paper roll. That worked reasonably well for textual interactions, and much of the early use of Logo was for playing with language (e.g., writing a pig-Latin generator). But the Logo team realized that they really needed some graphical interaction to attract the kids with whom they were working. They created a robot *turtle* with an attached pen to give the students something to control to make drawings.

The simple robot turtle sat on a large piece of paper, and when it moved (and if its pen was “down” and touching the paper) it would draw a line behind it. The Logo team literally invented a new kind of mathematics to make Logo work (*XXX Cite Abelson and diSessa*), where the turtle didn’t know Cartesian coordinates ((x, y) points) but instead knew its heading (which direction it was facing), and could turn and go forward. This relative positioning (as opposed to global, Cartesian coordinates) was actually enough to do quite a bit of mathematics, including biological simulations and an exploration of Einstein’s Special Theory of Relativity (*XXX Cite Abelson and diSessa*).

As we will see in the next section, the Logo turtle is very clearly a computational object, in our sense of object. The turtle *knows* some things (like its heading and whether its pen is down) and it can *do* some things (like turn and forward). But even more directly, the Logo turtle influenced the creation of object-oriented programming. Alan Kay (*XXX Cite "An Early History of Smalltalk*") modeled his *Smalltalk* programming language on Logo—and Smalltalk is considered to be the very first object-oriented programming language (a direct influence on Java), and Alan Kay is considered to be the inventor of object-oriented programming.

The Logo turtle today still exists in many implementations of many languages, but has multiplied. Seymour Papert’s student, Mitchel Resnick, developed a version of Logo, *StarLogo* with thousands of turtles that can interact with one another. Through this interaction, they can simulate scenarios like ants in an anthill, or termites, slime mold, or vehicle traffic[Resnick, 1997].

4.2 Drawing with Turtles

We’re going to use turtles to draw on our pictures in interesting and flexible ways, and to simplify animation. (See the Appendix for what the Turtle class looks like.) Our Turtle class instances can be created on a Picture or on a World. Think of a World as a constantly updating picture that repaints automatically. We create a World by simply creating a **new** one. We create

a Turtle on this world by passing the World instance in as input to the Turtle constructor (Figure 4.1).

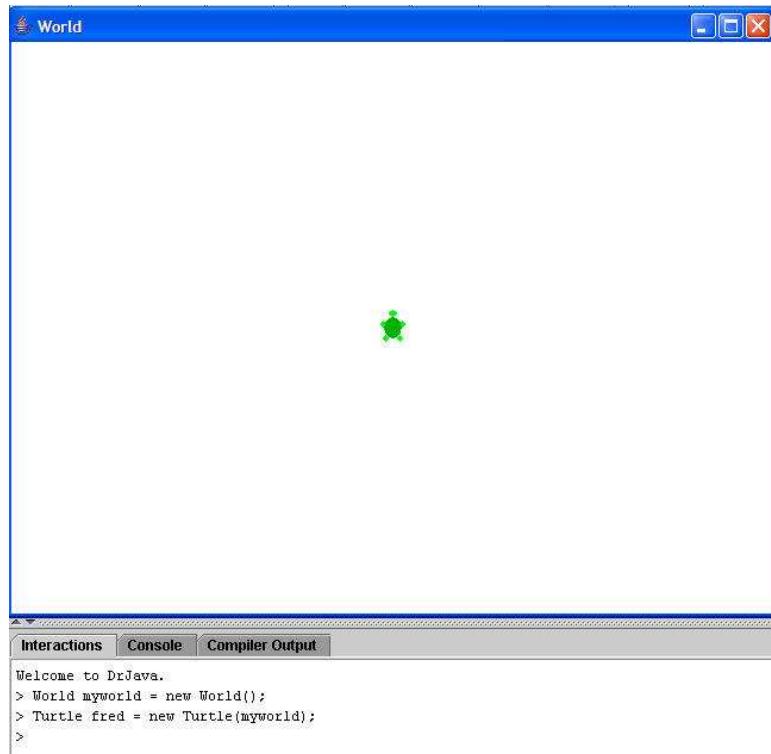


Figure 4.1: Starting a Turtle in a new World

Here's an example of opening a turtle on a Picture instead (Figure 4.2). Turtles can be created on blank Picture instances (which start out white) in the middle of the picture with pen down and with black ink. When a turtle is told to go forward, it moves forward the input number of *turtle steps* (think "pixels," which isn't *exactly* correct, but is close enough most of the time—the actual unit is computed by Java depending on your screen resolution) in whatever direction the turtle is currently facing. You can change the direction in which the turtle is facing by using the turn method which takes as input the number of degrees to turn. Positive degrees are clockwise, and negative ones are counter-clockwise.

```
> Picture blank = new Picture(200,200);
> Turtle fred = new Turtle(blank);
> fred
Unknown at 100, 100 heading 0
> fred.turn(-45);
```

72 CHAPTER 4. OBJECTS AS AGENTS: MANIPULATING TURTLES

```
> fred.forward(100);
> fred.turn(90);
> fred.forward(200);
> blank.show();
> blank.write("D:/cs1316/turtleexample.jpg")
```



Figure 4.2: A drawing with a turtle

Turtles know their position (unlike the original robot turtles) and their heading. The heading is 0 when straight up (how they're first created), and 90 when pointed to the right (due east), and 180 when pointed straight down. Clearly, turtles know things (e.g., their heading, *x* and *y* position) and can do things (e.g., like moving forward and turning).

```
> fred.forward(100);
> fred.turn(90);
> fred.getHeading()
90
> fred.getXPos()
320
> fred.getYPos()
140
```

Turtles can pick up their pen (stop drawing) using either the method `penUp()` or the code `setPenDown(false)`. We can set the pen down using `penDown()` or `setPenUp(true)`. Java does know about truth, or at least, about *Boolean* values: **true** and **false**.

To draw more complex shapes, we tell the turtle to do its basic steps repeatedly. Telling a turtle to go forward a certain number of steps and to turn 90 degrees makes a square.

```
> for (int sides=0; sides <= 3 ; sides++)
{fred.forward(100); fred.turn(90);}
```

When cascades don't work

Here's a thought experiment: will this work?

```
> World earth = new World();
> Turtle turtle = new Turtle(earth);
> turtle.forward(100).right(90);
```

The answer is “no,” but can you figure out why? Here’s a hint: The error you get in the Interactions Pane is Error: No ‘right’ method in ‘void’ with arguments: (**int**). While the error message is actually telling you exactly what the problem is, it’s written in *Javanese*—it presumes that you understand Java and can thus interpret the message.

The problem is that forward does not return anything, and certainly not a *turtle*. The method forward returns **void**. When we cascade methods like this, we are telling Java to invoke right(90) on what turtle.forward(100) returns. Since forward returns **void**, Java checks if instances of class **void** understand right. Of course not—**void** is nothing at all. So Java tells us that it checked for us, and the class **void** has no method right that takes an integer (**int**) input (e.g., 90 in our example). (Of course, **void** doesn’t know anything about right with *any* inputs, but just in case we only got the inputs wrong, Java lets us know what it looked for.) Thus: Error: No ‘right’ method in ‘void’ with arguments: (**int**).

You can only use a cascade of method calls if the *previous* method call returns an object that has a method defined for the *next* method call. Since forward returns nothing (**void**), you can’t cascade anything after it. Sure, we could create forward so that it does return the turtle **this**, the one it was invoked on, but one may ask if that makes any sense. Should forward return something?

Making lots of turtles

Using Mitchel Resnick’s StarLogo as inspiration, we may want to create something with *lots* of turtles. For example, consider what this program draws on the screen.

Example Java Code: **Creating a hundred turtles**

Program
Example #18

```
1  public class LotsOfTurtles {
2      public static void main(String[] args){
3          // Create a world
4          World myWorld = new World();
5          // A flotilla of turtles
6          Turtle [] myTurtles = new Turtle[100];
7
8      }
```

74 CHAPTER 4. OBJECTS AS AGENTS: MANIPULATING TURTLES

```

10    // Make a hundred turtles
11    for (int i=0; i < 100; i++) {
12        myTurtles[i] = new Turtle(myWorld);
13    }
14
15    // Tell them all what to do
16    for (int i=0; i < 100; i++) {
17        // Turn a random amount between 0 and 360
18        myTurtles[i].turn((int) (360 * Math.random()));
19        // Go 100 pixels
20        myTurtles[i].forward(100);
21    }
22}

```

How it works: Study the program and think about it before you look at Figure 4.3.

- First we create a World and name it myWorld.
- Next, we create an array to store 100 Turtle instances. Notice that `Turtle [] myTurtles = new Turtle[100];` creates no turtles!. That 100 is enclosed in square brackets—we’re not calling the Turtle constructor yet. Instead, we’re simply asking for 100 slots in an array myTurtles that will each be a Turtle.
- Inside a **for** loop that goes 100 times, we see `myTurtles[i] = new Turtle(myWorld);`. Here’s where we’re creating each of the 100 turtles in myWorld and putting each of them in their own slot of the array myTurtles.
- Finally, we tell each of the turtles to turn a random amount and go forward 100 steps. `Math.random()` returns a number between 0 and 1.0 where all numbers (e.g., 0.2341534) in that range are equally likely. Since that will be a **double**, we have to cast the result to **int** to use it as an input to `forward`.

Figured it out yet? It makes a circle of radius 100! This is an example from Mitchel Resnick’s book that introduced StarLogo[Resnick, 1997].

Obviously, we can have more than one Turtle in a World at once. Instances of class Turtle know some methods that allow them to interact with one another. They know how to `turnToFace(anotherTurtle)` which changes the one heading to match the other. They also know how to compute `getDistance(x,y)` which is the distance from this turtle (the one that `getDistance` was invoked upon) to the point x, y . Thus, in this example, r2d2 goes off someplace random on tatooine, but c3po turns to face him and moves forward exactly the right distance to catch r2d2.

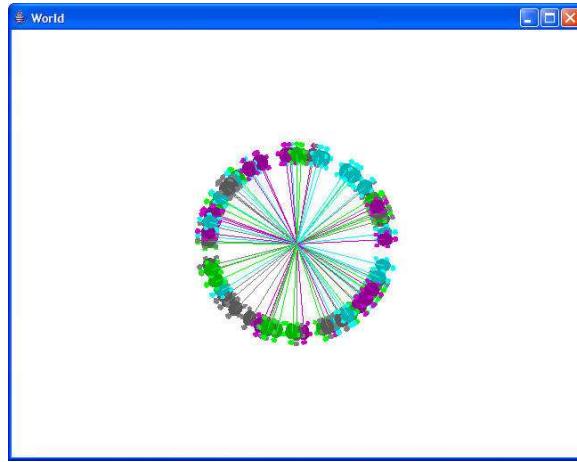


Figure 4.3: What you get with a hundred turtles starting from the same point, pointing in random directions, then moving forward the same amount

```
> World tattoine = new World();
> Turtle r2d2 = new Turtle(tattoine);
> r2d2.turn((int)
    (360 * Math.random()));
> r2d2.forward((int)
    (Math.random() * 100));
> Turtle c3po = new Turtle(tattoine);
> c3po.turnToFace(r2d2);
> c3po.forward((int)
    (c3po.getDistance(
        r2d2.getXPos(), r2d2.getYPos())));
```

Composing pictures with turtles

We saw earlier that we can place turtles on instance of class Picture, not just instances of class World. We can also use turtles to compose pictures into other pictures, through use of the drop method. Pictures get “dropped” *behind* (and to the right of) the turtle. If it’s facing down (heading of 180.0), then the picture shows up upside down (Figure 4.4).

```
> Picture monster = new Picture(FileChooser.getMediaPath("monster-right1.jpg"));
> Picture newbg = new Picture(400, 400);
> Turtle myturt = new Turtle(newbg);
> myturt.drop(monster);
> newbg.show();
```

We’ll rotate the turtle and drop again (Figure 4.5).



Figure 4.4: Dropping the monster character

```
> myturt.turn(180);
> myturt.drop(monster);
> newbg.repaint();
```



Figure 4.5: Dropping the monster character after a rotation

We can drop using loops and patterns, too (Figure 4.6). Why don't we see 12 monsters here? Maybe some are blocking the others?

```
> Picture frame = new Picture(600,600);
> Turtle mabel = new Turtle(frame);
> for (int i = 0; i < 12; i++)
    {mabel.drop(monster); mabel.turn(30);}
```

We can combine these in a main method to create a more complex image (Figure 4.7).

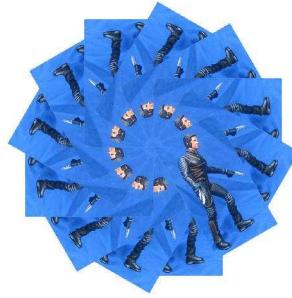


Figure 4.6: An iterated turtle drop of a monster

* * *

Example Java Code: **Making a picture with dropped pictures**

*Program
Example #19*

```

1   public class MyTurtlePicture {
2
3       public static void main(String [] args) {
4           Picture canvas = new Picture(600,600);
5           Turtle jenny = new Turtle(canvas);
6           Picture lilTurtle =
7               new Picture(
8                   FileChooser.getMediaPath("Turtle.jpg"));
9
10      for (int i=0; i <=40; i++)
11      {
12          if (i < 20)
13              {jenny.turn(20);}
14          else
15              {jenny.turn(-20);}
16          jenny.forward(40);
17          jenny.drop(lilTurtle.scale(0.5));
18      }
19      canvas.show();
20  }

```

* * *

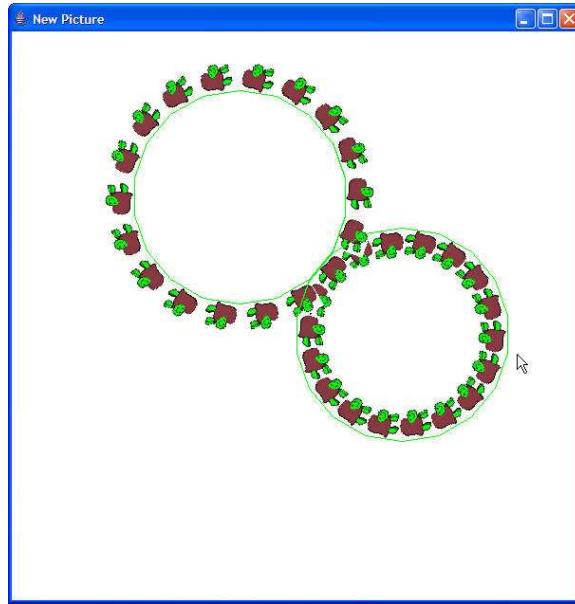


Figure 4.7: Making a more complex pattern of dropped pictures

4.3 Creating animations with turtles and frames

Our eyes tend to present an image to our brain, even for a few moments after the image has disappeared from sight. That's one of the reasons why we don't panic when we naturally blink (many times a minute without noticing)—the world doesn't go away and we don't see blackness. Rather, our eyes persist in showing the image in the brief interval when we blink—we call that *persistence of vision*.

A movie is a series of images, one shown right after the other. If we can show at least 16 images (*frames*) in a logical sequence in a second, our eye merges them through persistence of vision, and we perceive continuous motion. Fewer frames per second may be viewed as continuous, but it will probably be choppy. If we show frames that are not in a logical sequence, we perceive a montage, not continuous motion. Typical theater movies present at 22 frames per second, and video is typically 30 frames per second.

If we want to create an animation, then, we need to store a bunch of instances of Picture as frames, then play them back faster than 16 frames per second. We have a class for doing this, FrameSequence.

- The constructor for FrameSequence takes a path to a directory where frames will be stored as JPEG images, so that you might be able to reassemble them into a movie using some other tool (e.g., Windows

Movie Maker, Apple Quicktime Player, ImageMagick).

- A FrameSequence knows how to show(). Once shown, a FrameSequence will show each frame as it is added to the FrameSequence. When you first tell a FrameSequence to show, it will warn you that there's nothing to see until a frame is added.
- It knows how to addFrame(aPicture) to add another frame to a FrameSequence.
- It knows how to replay(delay) to show a sequence of frames back again. The delay is an integer for the number of milliseconds to wait between each frame. A delay of 62 is roughly 16 frames per second—anything around there or less will be perceived as continuous motion.

Here's a silly example of how you might use a FrameSequence. We're adding three (unrelated) pictures to a FrameSequence via addFrame. We can then replay the sequence back, one frame per second (or strictly, one frame per 1000 milliseconds).

```
> FrameSequence f = new FrameSequence("D:/Temp");
> f.show()
There are no frames to show yet. When you add a frame it will be
shown
> Picture t = new
    Picture("C:/cs1316/MediaSources/Turtle.jpg");
> f.addFrame(t);
> Picture barb = new
    Picture("C:/cs1316/MediaSources/Barbara.jpg");
> f.addFrame(barb);
> Picture katie = new
    Picture("C:/cs1316/MediaSources/Katie.jpg");
> f.addFrame(katie);
> f.replay(1000); // Delay one frame per second
```

Let's combine turtles and a FrameSequence to make an animation of frames.

Example Java Code: An animation generated by a Turtle

*Program
Example #20*

```
2   public class MyTurtleAnimation {
3       private Picture canvas;
4       private Turtle jenny;
5       private FrameSequence f;
6
7       public MyTurtleAnimation() {
8
```

80 CHAPTER 4. OBJECTS AS AGENTS: MANIPULATING TURTLES

```

10    canvas = new Picture(600,600);
11    jenny = new Turtle(canvas);
12    f = new FrameSequence("C:/Temp/");
13 }
14 public void next(){
15     Picture lilTurtle = new Picture(FileChooser.getMediaPath("Turtle.jpg"));
16     jenny.turn(-20);
17     jenny.forward(30);
18     jenny.turn(30);
19     jenny.forward(-5);
20     jenny.drop(lilTurtle.scale(0.5));
21     f.addFrame(canvas.copy()); // Try this as
22                           // f.addFrame(canvas);
23 }
24
25 public void next(int numTimes){
26     for (int i=0; i < numTimes; i++)
27     {this.next();}
28 }
29
30 public void show(){
31     f.show();
32 }
33
34 public void replay(int delay){
35     f.show();
36     f.replay(delay);
37 }
38 }
```

We run this program like this:

```
> MyTurtleAnimation anim = new MyTurtleAnimation();
> anim.next(20); // Generate 20 frames
> anim.replay(500); // Play them back, two per second
```

How it works: An instance of MyTurtleAnimation has three instance variables associated with it: A Picture onto which the turtle will draw named `canvas`, a Turtle named `jenny`, and a `FrameSequence`. The constructor for `MyTurtleAnimation` creates the original objects for each of these three names.

Common Bug: Don't declare the instance variables

There's a real temptation to put "Picture" in front of that line in the constructor `canvas = new Picture(600,600);`. But resist it. Will it compile? Absolutely. Will it work? Not at all. If you declare `canvas` a Picture inside

of the constructor `MyTurtleAnimation()`, it *only* exists in the constructor. If you want it canvas to exist outside that method, you have to access the instance variable, the field created in the object. If you don't declare it in the constructor, Java figures out that you mean the instance variable.

There are two different next methods in `MyTurtleAnimation`. The one that we called in the example (where we told it to go for 20 steps) is this one:

```
public void next(int numTimes){
    for (int i=0; i < numTimes; i++)
        {this.next();}
}
```

All that `next(int numTimes)` does is to call `next()` the input `numTimes` number of times. So the real activity in `MyTurtleAnimation` occurs in `next()` with no inputs.

```
public void next(){
    Picture lilTurtle = new Picture(
        FileChooser.getMediaPath("Turtle.jpg"));

    jenny.turn(-20);
    jenny.forward(30);
    jenny.turn(30);
    jenny.forward(-5);
    jenny.drop(lilTurtle.scale(0.5));
    f.addFrame(canvas.copy()); // Try this as
                               // f.addFrame(canvas);
}
```

The method `next()` does a bit of movement, a drop of a picture, and an addition of a frame to the `FrameSequence`. Basically, it generates the next frame in the animation sequence.

The `show()` and `replay()` methods *delegate* their definition to the `FrameSequence` instance. Delegation is where one class accomplishes what it needs to do by asking an instance of another class to do the task. It's perfectly reasonable to ask an animation to `show()` or `codereplay()`, but the way that it would the animation accomplishes those tasks is by asking the `FrameAnimation` to `show` or `replay`.

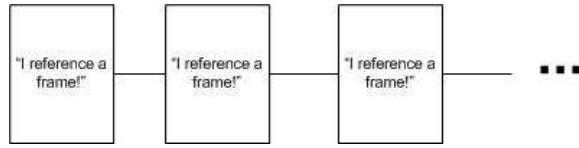
Data structure within FrameSequence

Did you try this same animation with the last line of `next` changed to `f.addFrame(canvas);`? What happened? If you did try it, you may have thought you made a mistake. When you ran the next animation, you saw the animation play out as normal. But when you executed `replay`, you saw only the final frame appear —never any other frame. Go check the temporary directory where the `FrameSequence` wrote out its frames. You'll find a

82 CHAPTER 4. OBJECTS AS AGENTS: MANIPULATING TURTLES

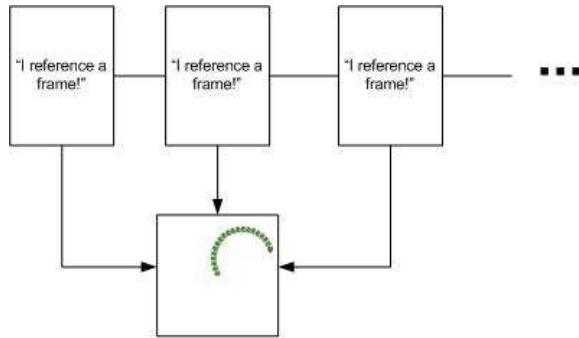
bunch of JPEG images there: `frame0001.jpg`, `frame0002.jpg`, and so on. So the animation did work and the `FrameSequence` did write out the frames. But why isn't it replaying correctly?

What we are seeing helps us to understand how `FrameSequence` works and what its internal data structure is. As you might imagine, a `FrameSequence` is a series of frames—but that's not correct in the details. The `FrameSequence` is actually a list of *references* to `Picture` objects. Each element in the `FrameSequence` refers to some `Picture`.

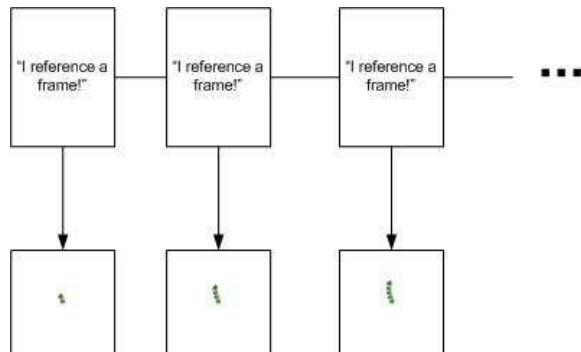


If a `FrameSequence` does not actually have any frames in it, where do the frames come from? From the `Picture` that you input to `addFrame!` That's what the `FrameSequence` frame references point to.

Without the `.copy()` method call on `canvas`, all the references in the code-`FrameSequence` point to *one* `Picture`, `canvas`. There is only *one* picture in the `FrameSequence`, and since, at the end, the `canvas` is in its final state, then all the references in `FrameSeqence` point to that same `canvas Picture` in its final state.



With the `.copy()` method call on `canvas`, you create *different* `Picture` instances for *each* reference in the `FrameSequence`. When we tell the `FrameSequence` to replay, the pictures referenced by the `FrameSequence` play out on the screen.



Exercises

1. Using sampled sounds and turtles, create a musical dance. Play music (sounds at least?) while the turtles move in patterns.
 - Your piece must last at least 10 seconds.
 - You must have at least five turtles. They can drop pictures, they can leave trails (or not), they can spin slowly, whatever you want them to do. Your turtles must move.
 - You must have at least four different sounds.
 - There must be some interweaving between turtle motion and sounds. In other words, you can't move the five turtles a little, then play 10 seconds of music. Hint: Use blockingPlay() instead of play() to play the sounds. If you use play(), the turtle movement and the sounds will go in parallel, which is nice, but you'll have no way to synchronize playing and moving. The method blockingPlay() will keep anything else from happening (e.g., turtle movement) during the playing of the sound.

Implement your dance and sounds in a `TurtleDance` class in a main method.

For extra credit, use MIDI (JMusic) instead of sampled sounds. Look up the `Play` object in the JMusic docs. `Play.midi(score, false)` will play a score in the background (`false` keeps it from quitting Java after playing). `Play.waitcycle(score)` will block (wait) anything else from happening until the score is done playing essentially, letting you block like `blockingPlay()`.

5 Arrays: A Static Data Structure for Sounds

Chapter Learning Objectives

The last media type that we will need to create animations like those segments of the wildebeests and villagers is sampled sound. Sampled sound has an advantage for our purposes here—it's naturally an array. Sounds are arrays of samples. We will use sampled sounds to talk about the strengths and weaknesses of arrays as a data structure.

The computer science goals for this chapter are:

- To use and manipulate arrays, including insertions into the middle (shifting the rest towards the end) and deletions from the middle (shifting the end back and padding).

The media learning goals for this chapter are:

- To understand how sounds are sampled and stored on a computer.
- To learn methods for manipulating sounds.

5.1 Manipulating Sampled Sounds

We can work with sounds that come from WAV files. We sometimes call these *sampled sounds* because they are sounds made up of *samples* (thousands per second), in comparison with *MIDI music* which encodes music (notes, durations, instrument selections) but not the sounds themselves.

A sampled sound is a series of numbers (samples) that represent the air pressure at a given moment in time. As you probably know from your physics classes, sound is the result of vibrations in the air molecules around our ears. When the molecules bunch up, there is an increase in air pressure called *compression*; when the molecules then space back up, there is a *rarefaction* (a drop) in the air pressure. We can plot air pressure over time to see the cycles of sounds.

Each of these samples is a number that goes both positive (for increases in air pressure) and negative (for rarefactions). Typically, two *bytes* (8 bits each, for a total of 16 bits) are used to store each sample. Given that we

86CHAPTER 5. ARRAYS: A STATIC DATA STRUCTURE FOR SOUNDS

have to represent both negative and positive numbers in those 16 bits, we have plus or minus 32,000 (roughly) as values in our samples. To capture all the frequencies of a sound that humans can hear, CD-quality sound requires that we capture 44,100 samples every second.

A WAV file stores samples, albeit in a compressed form. MIDI actually stores specifications of music. MIDI files contain encoded commands of the form “Press down on this key now” and later “release that key now.” What a “key” sounds like is determined by the MIDI synthesizer when the file is played. A WAV file, though, stores the original samples—the sound itself, as encoded on a computer.

Here’s a simple example of creating a Sound instance from a file, and playing it.

```
> Sound s = new Sound("C:/cs1316/MediaSources/thisisatest.wav");
> s.play();
> s.increaseVolume(2.0);
> s.play();
```

How it works: Here we see us creating a new Sound instance (by saying **new** Sound). The *constructor* for the Sound class (the method that constructs and initializes a new instance of a class) takes a WAV filename as input, then creates a Sound instance from those samples. We play the sound using the play() method. We then increase the volume by 2.0 and play it again.

Increasing the volume is a matter of increasing the amplitude of the sound. Here’s what that method looks like.

Program

Example #21

Example Java Code: Increase the volume of a sound by a factor

```
/*
 * Increase the volume of a sound
 */
4  public void increaseVolume(double factor){
5      SoundSample [] samples = this.getSamples();
6      SoundSample current = null;
7
8      for (int i=0; i < samples.length; i++) {
9          current = samples[i];
10         current.setValue((int) (factor * current.getValue()));
11     }
12 }
```

How it works: The first thing we do is to get all the samples out of the sound. getSamples() returns an array of SoundSample objects with all the

samples in the sound. We then use a **for** loop for the length of the samples. We get each sample, then multiply the factor times the current `getValue()` of the sample. We set the sample to the product. If the factor is less than 1.0, this reduces the volume, because the amplitude shrinks. If the factor is greater than 1.0, the sound increases in volume because the amplitude grows.

Just like with Picture instances, methods that *return* a new Sound are particularly powerful. Consider the following example:

```
> Sound s = new Sound("D:/cs1316/MediaSources/thisisatest.wav");
> s.play();
> s.reverse()
Sound number of samples: 64513
```

Why do you think we see this printout after `reverse()`? Because `reverse` doesn't *change* the Sound instance that it's called on—it returns a new one. If you were to execute `s.play()` right now, the sound would be the same. If you want to hear the reversed sound, you'd need to execute `s.reverse().play();`.

Here's how we reverse a sound.

Example Java Code: Reversing a sound

*Program
Example #22*

```
/*
 * Method to reverse a sound.
 */
public Sound reverse()
{
    Sound target = new Sound(getLength());
    int sampleValue;

    for (int srcIndex=0,trgIndex=getLength()-1;
        srcIndex < getLength();
        srcIndex++,trgIndex--)
    {
        sampleValue = this.getSampleValueAt(srcIndex);
        target.setSampleValueAt(trgIndex ,sampleValue);
    };
    return target;
}
```

How it works: The `reverse()` method first creates a target sound instance. It has the same length as the sound that `reverse()` is called upon. (Notice that a reference to `getLength()` without a specified object default to being references to `this`.) We use a **for** loop that manipulates two variables at

88CHAPTER 5. ARRAYS: A STATIC DATA STRUCTURE FOR SOUNDS

once. The source index, srcIndex goes up from 0 (the start of the array). The target index, trgIndex starts at the end of the list. Each time through the list, we add one to the source index and subtract one from the target index. The effect is to copy the front of the source to the back of the target, and to keep going until the whole sound is copied—reversing the sound. The methods getSampleValueAt and setSampleValueAt allow you to get and set the number in the sample. Most importantly, the last thing in the method is **return** target; which lets us meet the requirement of our method reverse() declaration, that it returns a Sound.

Debugging Tip: Beware the length as an index

Why did we subtract one from getLength to start out the target index (trgIndex)? Because getLength is the *number* of elements (samples) in the array, but the last *index* is getLength()–1. Computer scientists stubbornly insist on starting counting indices from zero, not one, so the last value is one less than the number of elements there.

Methods that return a new sound can then be used to create all kinds of interesting effects, without modifying the source sound.

```
> Sound s = new Sound(FileChooser.getMediaPath("gonga-2.wav"));
> Sound s2 = new Sound(FileChooser.getMediaPath("gongb-2.wav"));
> s.play();
> s2.play();
> s.reverse().play(); // Play first sound in reverse
> s.append(s2).play(); // Play first then second sound
> // Mix in the second sound, so you can hear part of each
> s.mix(s2,0.25).play();
> // Mix in the second sound sped up
> s.mix(s2.scale(0.5),0.25).play();
> s2.scale(0.5).play(); // Play the second sound sped up
> s2.scale(2.0).play(); // Play the second sound slowed down
> s.mix(s2.scale(2.0),0.25).play();
```

Given all of these, we can create a *collage* of sounds pretty easily.

Program
Example #23

Example Java Code: Create an audio collage

```
1  public class MySoundCollage {
2      public static void main(String [] args){
3          Sound snap = new Sound(
4              FileChooser.getMediaPath("snap-tenth.wav"));
5      }
6  }
```

```

8   Sound drum = new Sound(
9     FileChooser.getMediaPath("drumroll-1.wav"));
10  Sound clink = new Sound(
11    FileChooser.getMediaPath("clink-tenth.wav"));
12  Sound clap = new Sound(
13    FileChooser.getMediaPath("clap-q.wav"));

14  Sound drumRev = drum.reverse().scale(0.5);
15  Sound soundA = snap.append(clink).
16    append(clink).append(clap).append(drumRev);
17  Sound soundB = clink.append(clap).
18    append(clap).append(drum).append(snap).append(snap);

20  Sound collage = soundA.append(soundB).
21    append(soundB).append(soundA).
22    append(soundA).append(soundB);
23  collage.play();
24 }
}

```

Here is how some of these additional methods are coded.

Example Java Code: Append one sound with another

*Program
Example #24*

```

1 /**
2  * Return this sound appended with the input sound
3  * @param appendSound sound to append to this
4  */
5 public Sound append(Sound appendSound) {
6   Sound target = new Sound(getLength()+appendSound.getLength());
7   int sampleValue;
8
9   // Copy this sound in
10  for (int srcIndex=0,trgIndex=0;
11      srcIndex < getLength();
12      srcIndex++,trgIndex++)
13  {
14    sampleValue = this.getSampleValueAt(srcIndex);
15    target.setSampleValueAt(trgIndex, sampleValue);
16  };
17
18  // Copy appendSound in to target
19  for (int srcIndex=0,trgIndex=getLength();
20      srcIndex < appendSound.getLength();
21      srcIndex++,trgIndex++)

```

90CHAPTER 5. ARRAYS: A STATIC DATA STRUCTURE FOR SOUNDS

```
22     {
23         sampleValue = appendSound.getSampleValueAt(srcIndex);
24         target.setSampleValueAt(trgIndex, sampleValue);
25     };
26
27     return target;
28 }
```

Program
Example #25

Example Java Code: Mix in part of one sound with another

```
1 /**
2  * Mix the input sound with this sound, with percent ratio of input.
3  * Use mixIn sound up to length of this sound.
4  * Return mixed sound.
5  * @param mixIn sound to mix in
6  * @param ratio how much of input mixIn to mix in
7 */
8 public Sound mix(Sound mixIn, double ratio){
9     Sound target = new Sound(getLength());
10
11     int sampleValue, mixValue, newValue;
12
13     // Copy this sound in
14     for (int srcIndex=0,trgIndex=0;
15          srcIndex < getLength() && srcIndex < mixIn.getLength();
16          srcIndex++,trgIndex++)
17     {
18         sampleValue = this.getSampleValueAt(srcIndex);
19         mixValue = mixIn.getSampleValueAt(srcIndex);
20         newValue = (int)(ratio*mixValue) + (int)((1.0 - ratio)*sampleValue);
21         target.setSampleValueAt(trgIndex, newValue);
22     };
23     return target;
24 }
```

Program
Example #26

Example Java Code: Scale a sound up or down in frequency

```
1 /**
2  * Scale up or down a sound by the given factor
3  * (1.0 returns the same, 2.0 doubles the length, and 0.5 halves the length)
4 */
```

```

4      * @param factor ratio to increase or decrease
5      */
6  public Sound scale(double factor){
7      Sound target = new Sound((int) (factor * (1+getLength())));
8      int sampleValue;
9
10     // Copy this sound in
11     for (double srcIndex=0.0,trgIndex=0;
12         srcIndex < getLength();
13         srcIndex+=(1/factor),trgIndex++)
14     {
15         sampleValue = this.getSampleValueAt((int)srcIndex);
16         target.setSampleValueAt((int) trgIndex ,sampleValue);
17     };
18     return target;
19 }
```

How it works: There are several tricky things going on in these methods, but not *too* many. Most of them are just copy loops with some tweak.

- The class Sound has a *constructor* that takes the number of *samples*.
- You'll notice in reverse that we can use `--` as well as `++`. `variable--` is the same as `variable = variable - 1`.
- In `scale` you'll see another shorthand that Java allows: `srcIndex+=(1/factor)` is the same as `srcIndex = srcIndex + (1/factor)`.
- A **double** is a floating point number. These can't be automatically converted to integers. To use the results as integers where we need integers, we *cast* the result. We do that by putting the name of the class in parentheses before the result, e.g. `(int) srcIndex`.

5.2 Inserting and Deleting in an Array

A sound is naturally an array. It's a long list of sample values, one right after the other. Manipulation of sounds, then, gives us a sense of the trade-offs of working with an array.

Imagine that you want to insert one sound *into* another—not overwriting parts of the original sound, but pushing the end further down. So if you had “This is a test” and you wanted to insert a *clink* sound after the word “is,” you’d want to hear “This is*clink* a test.” It might look like this:

```

> Sound test = new Sound("D:/cs1316/MediaSources/thisisatest.wav");
> test.getLength()
64513
> Sound clink = new Sound("D:/cs1316/MediaSources/clink-tenth.wav");
> clink.getLength()
```

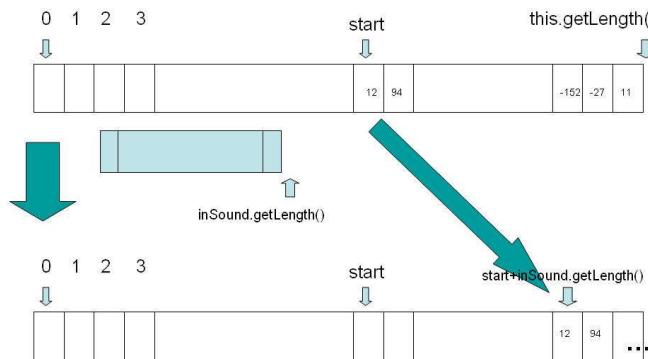
92CHAPTER 5. ARRAYS: A STATIC DATA STRUCTURE FOR SOUNDS

```
2184  
> test.insertAfter(clink, 40000)  
> test.play()
```

How would you do this? Think about doing it physically. If you had a line of objects in particular spots (think about a line of mailboxes in an office) and you had to insert something in the middle, how would you do it? First thing you'd have to do is to make space for the new ones. You'd move the ones from the end further down. You would move the last ones first, and then the ones just before the old last, and then the ones before that—moving backwards towards the front. There are some error conditions to consider, e.g., what if there's not enough mailboxes? Assuming that you *have to* put in the new ones, you have to lose some of the old content. Maybe trim off the end?

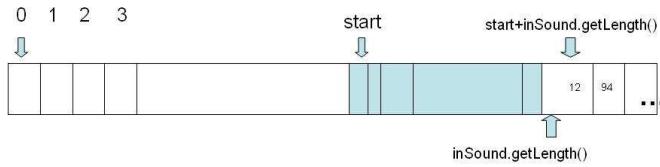
In any case, your first step would look something like this:

First, making room



And then, you would insert the new things in. That's the easy part.

Second, copying in



That's essentially what it takes to do it with sounds, too.

Program
Example #27

Example Java Code: **Inserting into the middle of sounds**

```

1  /**
2   * insert the input Sound after the nth sample (input integer).
3   * Modifies the given sound
4   * @param insound Sound to insert
5   * @param start index where to start inserting the new sound
6   */
7  public void insertAfter(Sound inSound, int start){
8
9      SoundSample current=null;
10     // Find how long insound is
11     int amtToCopy = inSound.getLength();
12     int endOfThis = this.getLength()-1;
13
14     if (start + amtToCopy > endOfThis)
15         {// If too long, copy only as much as will fit
16             amtToCopy = endOfThis-start-1;}
17     else {
18         // If short enough, need to clear out room.
19         // Copy from endOfThis-amtToCopy;, moving backwards
20         // (toward front of list) to start,
21         // moving UP (toward back) to endOfThis
22         // KEY INSIGHT: How much gets lost off the end of the
23         // array? Same size as what we're inserting — amtToCopy
24         for (int source=endOfThis-amtToCopy; source >= start ; source--)
25         {
26             // current is the TARGET — where we're copying to
27             current = this.getSample(source+amtToCopy);
28             current.setValue(this.getSampleValueAt(source));
29         }
30     }
31
32     // NOW, copy in inSound up to amtToCopy
33     for (int target=start,source=0;
34         source < amtToCopy;
35         target++, source++) {
36         current = this.getSample(target);
37         current.setValue(inSound.getSampleValueAt(source));
38     }
39 }
```

Think for a minute how long this takes to do. There are two loops here, and each one basically involves moving n elements, where n is the number of elements in the inserted array (sound). We would say that this algorithm is $O(2n)$ or $O(n)$. The number of operations grows linearly with the growth of the data.

Part II

Introducing Linked Lists

6 Structuring Music using Linked Lists

Chapter Learning Objectives

Media manipulators are often artists and always creative. They need flexibility in manipulating data, in inserting some here and deleting some there. Arrays, as we saw in the last chapter, are *not* particularly flexible. We will introduce *linked lists* in this chapter as a way of handling data more flexibly. In this chapter, the problem driving our exploration will be, “How do we make it easy for composers to creatively define music?”

The computer science goals for this chapter are:

- To create linked lists of various kinds.
- To understand and use operations on linked lists, including traversals, insertion, deletion, repetition, and weaving.
- To understand the tradeoffs between linked lists and arrays.
- To see a need for tree structures.

The media learning goals for this chapter are:

- To manipulate data flexibly.
- To develop different structures for composing MIDI creatively.
- To use rudimentary forms automated composition of music.

6.1 JMusic and Imports

Before you can use special features, those not built into the basic Java language, you have to **import** them.

Here’s what it looks like when you run with the JMusic libraries installed (Figure 6.1):

```
Welcome to DrJava.  
> import jm.music.data.*;  
> import jm.JMC;
```

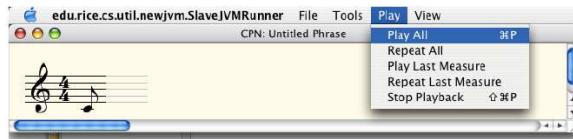


Figure 6.1: Playing all the notes in a score

```
> import jm.util.*;
> Note n = new Note(60,101);
> Note n = new Note(60,0.5); // Can't do this
Error: Redefinition of 'n'
> n=new Note(60,0.5);
> Phrase phr = new Phrase();
> phr.addNote(n);
> View.notate(phr);
```

The first argument to the *constructor* (the call to the class to create a new instance) for class Note is the *MIDI note*. Figure 6.2 shows the relation between frequencies, keys, and MIDI notes¹.

Here's another java that uses a different Phrase constructor to specify a starting time and an *instrument* which is also known as a *MIDI program*.

```
> import jm.music.data.*;
> import jm.JMC;
> import jm.util.*;
> Note n = new Note(60,0.5)
> Note n2 = new Note(JMC.C4,JMC.QN)
> Phrase phr = new Phrase(0.0,JMC.FLUTE);
> phr.addNote(n);
> phr.addNote(n2);
> View.notate(phr);
```

How it works:

- We import the pieces we need for Jmusic.
- We create a note using constants, then using named constants. JMC.C4 means “C in the 4th octave.” JMC.QN means “quarter note.” JMC is the class *Java Music Constants*, and it holds many important constants. The constant JMC.C4 means 60, like in the Table 2.1. A sharp would be noted like JMC.CS5 (C-sharp in the 5th octave). Eighth note is JMC.EN and half note is JMC.HN. A dotted eighth would be JMC.DEN.
- We create a Phrase object that starts at time 0.0 and uses the *instrument* JMC.FLUTE. JMC.FLUTE is a constant that corresponds to the correct instrument from Table 6.1.

¹Taken from <http://www.phys.unsw.edu.au/~jw/notes.html>

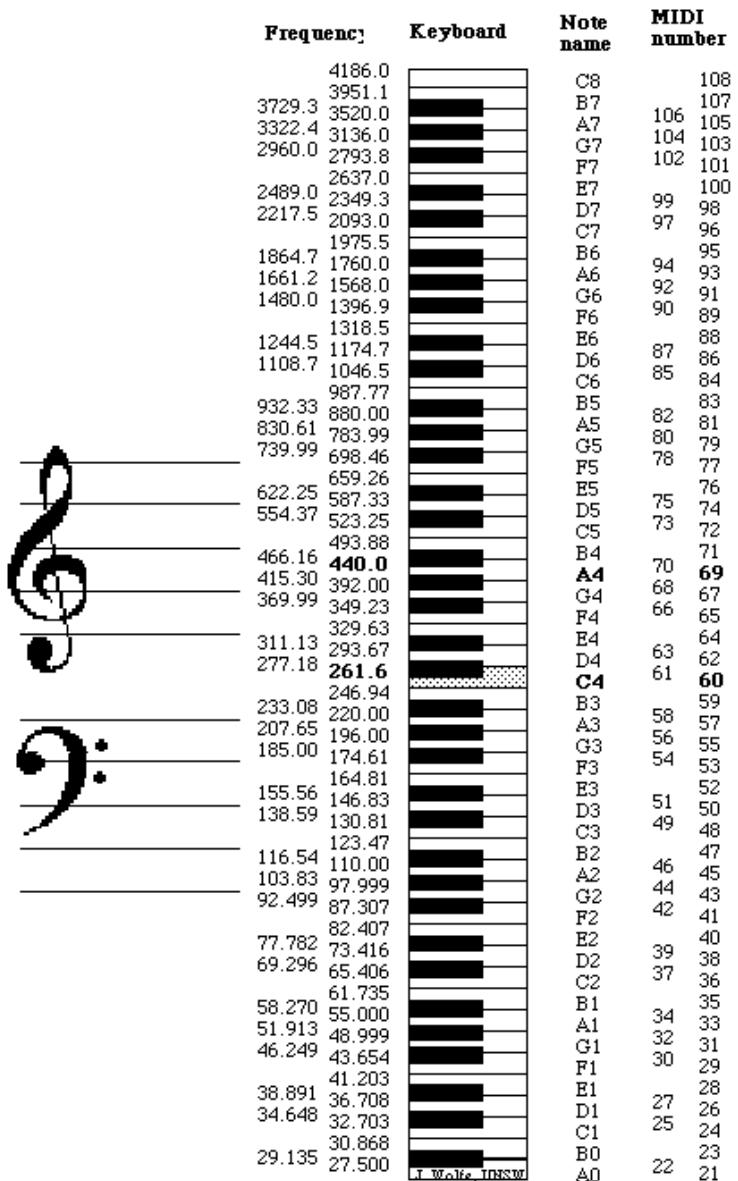


Figure 6.2: Frequencies, keys, and MIDI notes—something I found on the Web that I need to recreate in a new way

- We put the notes into the Phrase instance, and then notate and view the whole phrase.

Piano	Bass	Reed	Synth Effects
0 — Acoustic Grand Piano	32 — Acoustic Bass	64 — Soprano Sax	96 — FX 1 (rain)
1 — Bright Acoustic Piano	33 — Electric Bass (finger)	65 — Alto Sax	97 — FX 2 (soundtrack)
2 — Electric Grand Piano	34 — Electric Bass (pick)	66 — Tenor Sax	98 — FX 3 (crystal)
3 — Honky-tonk Piano	35 — Fretless Bass	67 — Baritone Sax	99 — FX 4 (atmosphere)
4 — Rhodes Piano	36 — Slap Bass 1	68 — Oboe	100 — FX 5 (brightness)
5 — Chorused Piano	37 — Slap Bass 2	69 — English Horn	101 — FX 6 (goblins)
6 — Harpsichord	38 — Synth Bass 1	70 — Bassoon	102 — FX 7 (echoes)
7 — Clavinet	39 — Synth Bass 2	71 — Clarinet	103 — FX 8 (sci-fi)
Chromatic Percussion	Strings	Pipe	Ethnic
8 — Celesta	40 — Violin	72 — Piccolo	104 — Sitar
9 — Glockenspiel	41 — Viola	73 — Flute	105 — Banjo
10 — Music box	42 — Cello	74 — Recorder	106 — Shamisen
11 — Vibraphone	43 — Contrabass	75 — Pan Flute	107 — Koto
12 — Marimba	44 — Tremolo Strings	76 — Bottle Blow	108 — Kalimba
13 — Xylophone	45 — Pizzicato Strings	77 — Shakuhachi	109 — Bagpipe
14 — Tubular Bells	46 — Orchestral Harp	78 — Whistle	110 — Fiddle
15 — Dulcimer	47 — Timpani	79 — Ocarina	111 — Shanai
Organ	Ensemble	Synth Lead	Percussive
16 — Hammond Organ	48 — String Ensemble 1	80 — Lead 1 (square)	112 — Tinkle Bell
17 — Percussive Organ	49 — String Ensemble 2	81 — Lead 2 (sawtooth)	113 — Agogo
18 — Rock Organ	50 — Synth Strings 1	82 — Lead 3 (caliope lead)	114 — Steel Drums
19 — Church Organ	51 — Synth Strings 2	83 — Lead 4 (chiff lead)	115 — Woodblock
20 — Reed Organ	52 — Choir Aahs	84 — Lead 5 (charang)	116 — Taiko Drum
21 — Accordion	53 — Voice Ooohs	85 — Lead 6 (voice)	117 — Melodic Tom
22 — Harmonica	54 — Synth Voice	86 — Lead 7 (fifths)	118 — Synth Drum
23 — Tango Accordion	55 — Orchestra Hit	87 — Lead 8 (brass + lead)	119 — Reverse Cymbal
Guitar	Brass	Synth Pad	Sound Effects
24 — Acoustic Guitar (nylon)	56 — Trumpet	88 — Pad 1 (new age)	120 — Guitar Fret Noise
25 — Acoustic Guitar (steel)	57 — Trombone	89 — Pad 2 (warm)	121 — Breath Noise
26 — Electric Guitar (jazz)	58 — Tuba	90 — Pad 3 (polysynth)	122 — Seashore
27 — Electric Guitar (clean)	59 — Muted Trumpet	91 — Pad 4 (choir)	123 — Bird Tweet
28 — Electric Guitar (muted)	60 — French Horn	92 — Pad 5 (bowed)	124 — Telephone Ring
29 — Overdriven Guitar	61 — Brass Section	93 — Pad 6 (metallic)	125 — Helicopter
30 — Distortion Guitar	62 — Synth Brass 1	94 — Pad 7 (halo)	126 — Applause
31 — Guitar Harmonics	63 — Synth Brass 2	95 — Pad 8 (sweep)	127 — Gunshot

Table 6.1: MIDI Program numbers

We can create multiple parts with different start times and instruments. We want the different parts to map onto different *MIDI channels* if we want different start times and instruments (Figure 6.3). We'll need to combine the different parts into a Score object, which can then be viewed and notated the same way as we have with phrases and parts.

```

> Note n3=new Note(JMC.E4, JMC.EN)
> Note n4=new Note(JMC.G4, JMC.HN)
> Phrase phr2= new Phrase(0.5, JMC.PIANO);
> phr2.addNote(n3)
> phr2.addNote(n4)
> phr2
----- jMusic PHRASE: 'Untitled Phrase' contains 2 notes. Start time: 0.0 -----
jMusic NOTE: [Pitch = 60] [RhythmValue = 0.5] [Dynamic = 85] [Pan =
0.5] [Duration = 0.45] jMusic NOTE: [Pitch = 60] [RhythmValue =
1.0] [Dynamic = 85] [Pan = 0.5] [Duration = 0.9]

> phr2

```

```
----- jMusic PHRASE: 'Untitled Phrase' contains 2 notes. Start time: 0.5 -----
jMusic NOTE: [Pitch = 64] [RhythmValue = 0.5] [Dynamic = 85] [Pan =
0.5] [Duration = 0.45] jMusic NOTE: [Pitch = 67] [RhythmValue =
2.0] [Dynamic = 85] [Pan = 0.5] [Duration = 1.8]

> Part partA = new Part(phr,"Part A",JMC.FLUTE,1)
> Part partB = new Part(phr2,"Part B",JMC.PIANO,2)
> Phrase phraseAB = new Phrase()
> Score scoreAB = new Score()
> scoreAB.addPart(partA)
> scoreAB.addPart(partB)
> View.notate(scoreAB)
```



Figure 6.3: Viewing a multipart score

How do you figure out what JMusic can do, what the classes are, and how to use them? There is a standard way of documenting Java classes called *Javadoc* which produces really useful documentation (Figure 6.4). JMusic is documented in this way. You can get to the JMusic Javadoc at <http://jmusic.ci.qut.edu.au/jmDocumentation/index.html>, or you can download it onto your own computer <http://jmusic.ci.qut.edu.au/GetJMusic.html>.



Figure 6.4: JMusic documentation for the class Phrase

Table A.1 in the Appendix lists the constant names in JMC for accessing instrument names.

6.2 Starting out with JMusic

Here's what it looks like when you run:

```
Welcome to DrJava.
> import jm.music.data.*;
> import jm.JMC;
> import jm.util.*;
> Note n = new Note(C4,QUARTER_NOTE);
Error: Undefined class 'C4'
> Note n = new Note(60,QUARTER_NOTE);
Error: Undefined class 'QUARTER_NOTE'
> Note n = new Note(60,101);
> Note n = new Note(60,0.5);
Error: Redefinition of 'n'
> n=new Note(60,0.5);
> Phrase phr = new Phrase();
> phr.addNote(n);
> View.notate(phr);
Error: Undefined class 'phrase'
> View.notate(phr);
```

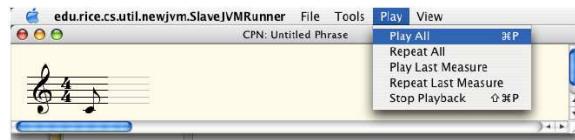


Figure 6.5: Playing all the notes in a score

6.3 Making a Simple Song Object

*Program
Example #28*

Example Java Code: *Amazing Grace as a Song Object*

```
1 import jm.music.data.*;
2 import jm.JMC;
3 import jm.util.*;
4 import jm.music.tools.*;
5
6 public class AmazingGraceSong {
```

```

8   private Score myScore = new Score("Amazing Grace");
9
10  public void fillMeUp(){
11      myScore.setTimeSignature(3,4);
12
13      double[] phrase1data =
14          {JMC.G4, JMC.QN,
15          JMC.C5, JMC.HN, JMC.E5,JMC.EN, JMC.C5,JMC.EN,
16          JMC.E5,JMC.HN,JMC.D5,JMC.QN,
17          JMC.C5,JMC.HN,JMC.A4,JMC.QN,
18          JMC.G4,JMC.HN,JMC.G4,JMC.EN,JMC.A4,JMC.EN,
19          JMC.C5,JMC.HN,JMC.E5,JMC.EN,JMC.C5,JMC.EN,
20          JMC.E5,JMC.HN,JMC.D5,JMC.EN,JMC.E5,JMC.EN,
21          JMC.G5,JMC.DHN};
22      double[] phrase2data =
23          {JMC.G5,JMC.HN,JMC.E5,JMC.EN,JMC.G5,JMC.EN,
24          JMC.G5,JMC.HN,JMC.E5,JMC.EN,JMC.C5,JMC.EN,
25          JMC.E5,JMC.HN,JMC.D5,JMC.QN,
26          JMC.C5,JMC.HN,JMC.A4,JMC.QN,
27          JMC.G4,JMC.HN,JMC.G4,JMC.EN,JMC.A4,JMC.EN,
28          JMC.C5,JMC.HN,JMC.E5,JMC.EN,JMC.C5,JMC.EN,
29          JMC.E5,JMC.HN,JMC.D5,JMC.QN,
30          JMC.C5,JMC.DHN
31      };
32      Phrase myPhrase = new Phrase();
33      myPhrase.addNoteList(phrase1data);
34      myPhrase.addNoteList(phrase2data);
35      //Mod. repeat(aPhrase, repeats);
36      // create a new part and add the phrase to it
37      Part aPart = new Part("Parts",
38                      JMC.FLUTE, 1);
39      aPart.addPhrase(myPhrase);
40      // add the part to the score
41      myScore.addPart(aPart);
42
43  };
44
45  public void showMe(){
46      View.notate(myScore);
47  };
48}

```

How it works:

- We start with the **import** statements needed to use JMusic.
- We're declaring a new **class** whose name is AmazingGraceSong. It's

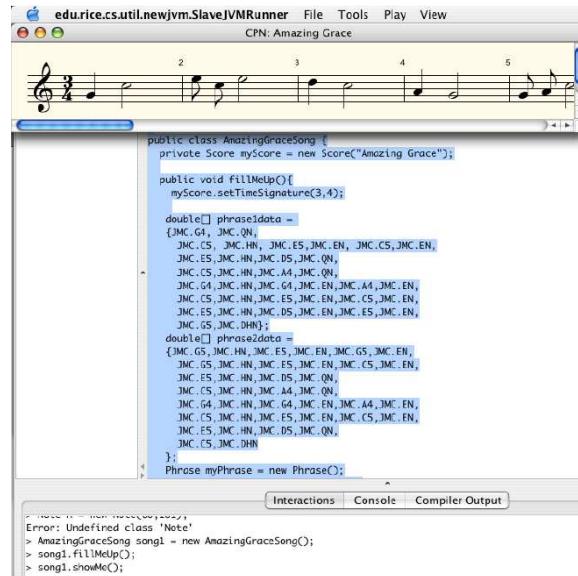


Figure 6.6: Trying the Amazing Grace song object

public meaning that anyone can access it.

- There is a variable named **myScore** which is of type class **Score**. This means that the score **myScore** is duplicated in each instance of the class **AmazingGraceSong**. It's **private** because we don't actually want users of **AmazingGraceSong** messing with the score.
- There are two methods, **fillMeUp** and **showMe**. The first method fills the song with the right notes and durations (see the phrase data arrays in **fillMeUp**) with a flute playing the song. The second one opens it up for notation and playing.

The phrase data arrays are named constants from the **JMC** class. They're in the order of note, duration, note, duration, and so on. The names actually all correspond to numbers, **doubles**.

Using the program (Figure 6.6):

```

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

```

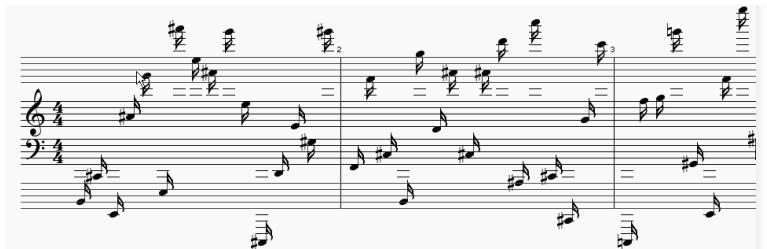


Figure 6.7: A hundred random notes

6.4 Simple structuring of notes with an array

Let's start out grouping notes into arrays. We'll use `Math.random()` to generate random numbers between 0.0 and 1.0. We'll generate 100 random notes (Figure 6.7).

```
> import jm.util.*;
> import jm.music.data.*;
> Note [] somenotes = new Note[100];
> for (int i = 0; i<100; i++)
{ somenotes[i]=new Note((int)
(128*Math.random()),0.25); }
> Phrase phr=new Phrase();
> for (int i= 0; i<100; i++)
{ phr.addNote(somenotes[i]); }
> View.notate(phr);
```

6.5 Making the Song Something to Explore

In a lot of ways `AmazingGraceSong` is a really lousy example—and not simply because it's a weak version of the tune. We can't really explore much with this version. What does it mean to have something that we can explore with?

How might one want to explore a song like this? We can come up with several ways, without even thinking much about it.

- How about changing the order of the pieces, or duplicating them?
Maybe use a *Call and response* structure?
- How about using different instruments?

We did learn in an earlier chapter how to create songs with multiple parts. We can easily do multiple voice and multiple part *Amazing Grace*. Check out the below.

* * *

Example Java Code: **Amazing Grace with Multiple Voices**

*Program
Example #29*

```

1 import jm.music.data.*;
2 import jm.JMC;
3 import jm.util.*;
4 import jm.music.tools.*;

6 public class MVAmazingGraceSong {
7     private Score myScore = new Score("Amazing Grace");
8
9     public Score getScore() {
10         return myScore;
11     }
12
13     public void fillMeUp(){
14         myScore.setTimeSignature(3,4);
15
16         double[] phrase1data =
17             {JMC.G4, JMC.QN,
18              JMC.C5, JMC.HN, JMC.E5,JMC.EN, JMC.C5,JMC.EN,
19              JMC.E5,JMC.HN,JMC.D5,JMC.QN,
20              JMC.C5,JMC.HN,JMC.A4,JMC.QN,
21              JMC.G4,JMC.HN,JMC.G4,JMC.EN,JMC.A4,JMC.EN,
22              JMC.C5,JMC.HN,JMC.E5,JMC.EN,JMC.C5,JMC.EN,
23              JMC.E5,JMC.HN,JMC.D5,JMC.EN,JMC.E5,JMC.EN,
24              JMC.G5,JMC.DHN};
25         double[] phrase2data =
26             {JMC.G5,JMC.HN,JMC.E5,JMC.EN,JMC.G5,JMC.EN,
27              JMC.G5,JMC.HN,JMC.E5,JMC.EN,JMC.C5,JMC.EN,
28              JMC.E5,JMC.HN,JMC.D5,JMC.QN,
29              JMC.C5,JMC.HN,JMC.A4,JMC.QN,
30              JMC.G4,JMC.HN,JMC.G4,JMC.EN,JMC.A4,JMC.EN,
31              JMC.C5,JMC.HN,JMC.E5,JMC.EN,JMC.C5,JMC.EN,
32              JMC.E5,JMC.HN,JMC.D5,JMC.QN,
33              JMC.C5,JMC.DHN
34      };
35
36      //
37      Phrase trumpetPhrase = new Phrase();
38      trumpetPhrase.addNoteList(phrase1data); // 22.0 beats long
39      double endphrase1 = trumpetPhrase.getEndTime();
40      System.out.println("End of phrase1:"+endphrase1);
41      trumpetPhrase.addNoteList(phrase2data);
42      // create a new part and add the phrase to it
43      Part part1 = new Part("TRUMPET PART",
44          JMC.TRUMPET, 1);
45      part1.addPhrase(trumpetPhrase);
46      // add the part to the score

```

```

48     myScore.addPart(part1);
49     //
50     Phrase flutePhrase = new Phrase(endphrase1);
51     flutePhrase.addNoteList(phrase1data); // 22.0 beats long
52     flutePhrase.addNoteList(phrase2data); // optionally, remove this
53     // create a new part and add the phrase to it
54     Part part2 = new Part("FLUTE PART",
55                           JMC.FLUTE, 2);
56     part2.addPhrase(flutePhrase);
57     // add the part to the score
58     myScore.addPart(part2);

59 }

60 }

61 public void showMe(){

62     View.notate(myScore);
63 }

64 }

65 }

66 }

```

We can use this program like this (Figure 6.8):

```

> MVAmazingGraceSong mysong = new MVAmazingGraceSong();
> song1.fillMeUp()
End of phrase1:22.0
> mysong.showMe();

```



Figure 6.8: Multi-voice *Amazing Grace* notation

How it works: The main idea that makes this program work is that we create two phrases, one of which starts when first phrase (which is 22 beats long) ends. You'll note the use of `System.out.println()` which is a method that takes a string as input and prints it to the console. Parsing that method is probably a little challenging. There is a big object that has a lot of important objects as part of it called `System`. It includes a connection to

the Interactions Pane called out. That connection (called a *stream*) knows how to print strings through the `println` (print line) method. The string concatenation operator, `+`, knows how to convert numbers into strings automatically.

But that's not a very satisfying example. Look at the `fillMeUp` method—that's pretty confusing stuff! What we do in the Interactions Pane doesn't give us much room to play around. The current structure doesn't lend itself to exploration.

How can we structure our program so that it's *easy* to explore, to try different things? How about if we start by thinking about how *expert musicians* think about music. They typically don't think about a piece of music as a single thing. Rather, they think about it in terms of a whole (a Score), parts (Part), and phrases (Phrase). They do think about these things in terms of a *sequence*—one part follows another. Each part will typically have its own notes (its own Phrase) and a starting time (sometimes parts start together, to get simultaneity, but at other times, will play after one another). Very importantly, there is an *ordering* to these parts. We can *model* that ordering by having each part know which other part comes next.

Let's try that in this next program.

Program
Example #30

Example Java Code: **Amazing Grace as Song Elements**

```

import jm.music.data.*;
2 import jm.JMC;
import jm.util.*;
4 import jm.music.tools.*;

6 public class AmazingGraceSongElement {
    // Every element knows its next element and its part (of the score)
8     private AmazingGraceSongElement next;
     private Part myPart;

10    // When we make a new element, the next part is empty, and ours is a blank new part
12    public AmazingGraceSongElement(){
        this.next = null;
        this.myPart = new Part();
    }

16    // addPhrase1 puts the first part of AmazingGrace into our part of the song
18    // at the desired start time with the given instrument
20    public void addPhrase1(double startTime, int instrument){
22        double[] phrase1data =
            {JMC.G4, JMC.QN,
             JMC.C5, JMC.HN, JMC.E5,JMC.EN, JMC.C5,JMC.EN,

```

```

24     JMC.E5,JMC.HN,JMC.D5,JMC.QN,
25     JMC.C5,JMC.HN,JMC.A4,JMC.QN,
26     JMC.G4,JMC.HN,JMC.G4,JMC.EN,JMC.A4,JMC.EN,
27     JMC.C5,JMC.HN,JMC.E5,JMC.EN,JMC.C5,JMC.EN,
28     JMC.E5,JMC.HN,JMC.D5,JMC.EN,JMC.E5,JMC.EN,
29     JMC.G5,JMC.DHN} ;
30
31     Phrase myPhrase = new Phrase(startTime);
32     myPhrase.addNoteList(phrase1data);
33     this.myPart.addPhrase(myPhrase);
34     // In MVAmaizingGraceSong, we did this when we initialized
35     // the part. But we CAN do it later
36     this.myPart.setInstrument(instrument);
37 }
38
39 public void addPhrase2(double startTime, int instrument) {
40     double[] phrase2data =
41     {JMC.G5,JMC.HN,JMC.E5,JMC.EN,JMC.G5,JMC.EN,
42      JMC.G5,JMC.HN,JMC.E5,JMC.EN,JMC.C5,JMC.EN,
43      JMC.E5,JMC.HN,JMC.D5,JMC.QN,
44      JMC.C5,JMC.HN,JMC.A4,JMC.QN,
45      JMC.G4,JMC.HN,JMC.G4,JMC.EN,JMC.A4,JMC.EN,
46      JMC.C5,JMC.HN,JMC.E5,JMC.EN,JMC.C5,JMC.EN,
47      JMC.E5,JMC.HN,JMC.D5,JMC.QN,
48      JMC.C5,JMC.DHN
49 };
50
51     Phrase myPhrase = new Phrase(startTime);
52     myPhrase.addNoteList(phrase2data);
53     this.myPart.addPhrase(myPhrase);
54     this.myPart.setInstrument(instrument);
55 }
56
57 // Here are the two methods needed to make a linked list of elements
58 public void setNext(AmazingGraceSongElement nextOne){
59     this.next = nextOne;
60 }
61
62 public AmazingGraceSongElement next(){
63     return this.next;
64 }
65
66 // We could just access myPart directly
67 // but we can CONTROL access by using a method
68 // (called an accessor)
69 // We'll use it in showFromMeOn
70 // (So maybe it doesn't need to be Public?)
71 public Part part(){
72     return this.myPart;
73 }
```

```

74      // Why do we need this?
76      // If we want one piece to start after another, we need
77      // to know when the last one ends.
78      // Notice: It's the phrase that knows the end time.
79      // We have to ask the part for its phrase (assuming only one)
80      // to get the end time.
81      public double getEndTime(){
82          return this.myPart.getPhrase(0).getEndTime();
83      }
84
85      // We need setChannel because each part has to be in its
86      // own channel if it has different start times.
87      // So, we'll set the channel when we assemble the score.
88      // (But if we only need it for showFromMeOn, we could
89      // make it PRIVATE...)
90      public void setChannel(int channel){
91          myPart.setChannel(channel);
92      }
93
94      public void showFromMeOn(){
95          // Make the score that we'll assemble the elements into
96          Score myScore = new Score("Amazing Grace");
97          myScore.setTimeSignature(3,4);
98
99          // Each element will be in its own channel
100         int channelCount = 1;
101
102         // Start from this element (this)
103         AmazingGraceSongElement current = this;
104         // While we're not through...
105         while (current != null)
106         {
107             // Set the channel, increment the channel, then add it in.
108             current.setChannel(channelCount);
109             channelCount = channelCount + 1;
110             myScore.addPart(current.part());
111
112             // Now, move on to the next element
113             // which we already know isn't null
114             current = current.next();
115         };
116
117         // At the end, let's see it!
118         View.notate(myScore);
119
120     }
121
122 }
```

* * *

So, imagine that we want to play the first part as a flute, and the second part as a piano. Here's how we do it.

Welcome to DrJava.

```
> import jm.JMC;
> AmazingGraceSongElement part1 = new AmazingGraceSongElement();
> part1.addPhrase1(0.0,JMC.FLUTE);
> AmazingGraceSongElement part2 = new AmazingGraceSongElement();
> part2.addPhrase2(part1.getEndTime(),JMC.PIANO);
> part1.setNext(part2);
> part1.showFromMeOn()
```

That's an awful lot of extra effort just to do this, but here's the cool part. Let's do several other variations on *Amazing Grace* without writing any more programs. Say that you have a fondness for banjo, fiddle, and pipes for *Amazing Grace* (Figure 6.9).

```
> AmazingGraceSongElement banjo1 = new AmazingGraceSongElement();
> banjo1.addPhrase1(0.0,JMC.BANJO);
> AmazingGraceSongElement fiddle1=new AmazingGraceSongElement();
> fiddle1.addPhrase1(0.0,JMC.FIDDLE);
> banjo1.setNext(fiddle1);
> banjo1.getEndTime()
22.0
> AmazingGraceSongElement pipes2=new AmazingGraceSongElement();
> pipes2.addPhrase2(22.0,JMC.PIPES);
> fiddle1.setNext(pipes2);
> banjo1.showFromMeOn();
```

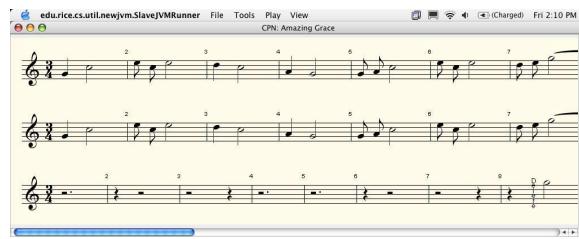


Figure 6.9: AmazingGraceSongElements with 3 pieces

But now you're feeling that you want more of an orchestra feel. How about if we throw all of this together? That's easy. AmazingGraceSongElement part1 is already linked to part2. AmazingGraceSongElement pipes1 isn't linked to anything. We'll just link part1 onto the end—very easy, to do a new experiment.

```
> pipes2.setNext(part1);
> banjo1.showFromMeOn();
```

Now we have a song with five pieces (Figure 6.10). “But wait,” you might be thinking. “The ordering is all wrong!” Fortunately, the score figures it out for us. The starting times are all that’s needed. The notion of a *next* element is just for our sake, to structure which pieces we want where.



Figure 6.10: AmazingGraceSongElements with 3 pieces

At this point, you should be able to see how to play with lots of different pieces. What if you have a flute echo the pipes, just one beat behind? What if you want to have several difference instruments playing the same thing, but one measure (three beats) behind the previous? Try them out!

Computer Science Idea: Layering software makes it easier to change
 Notice that Phrase and Part has disappeared here. All that we’re manipulating are song elements. A good layer allows you to ignore the layers below.

6.6 Making Any Song Something to Explore

What makes AmazingGraceSongElement something specific to the song *AmazingGrace*? It’s really just those two addPhrase methods. Let’s think about how we might generalize (abstract) these to make them usable to explore any song.

First, let’s create a second version (cunningly called AmazingGraceSongElement2) where there is only one addPhrase method, but you decide which phrase you

want as an input. We'll also clean up some of our protections here, while we're revising.

Example Java Code: Amazing Grace as Song Elements, Take 2

*Program
Example #31*

```

1  import jm.music.data.*;
2  import jm.JMC;
3  import jm.util.*;
4  import jm.music.tools.*;

6  public class AmazingGraceSongElement2 {
7      // Every element knows its next element and its part (of the score)
8      private AmazingGraceSongElement2 next;
9      private Part myPart;

10     // When we make a new element, the next part is empty, and ours is a blank new part
11     public AmazingGraceSongElement2(){
12         this.next = null;
13         this.myPart = new Part();
14     }
15
16     // setPhrase takes a phrase and makes it the one for this element
17     // at the desired start time with the given instrument
18     public void setPhrase(Phrase myPhrase, double startTime, int instrument){
19
20         //Phrases get returned from phrase1() and phrase2() with default (0.0) starTime
21         // We can set it here with whatever setPhrase gets as input
22         myPhrase.setStartTime(startTime);
23         this.myPart.addPhrase(myPhrase);
24         // In MVAmazingGraceSong, we did this when we initialized
25         // the part. But we CAN do it later
26         this.myPart.setInstrument(instrument);
27     }
28
29
30     // First phrase of Amazing Grace
31     public Phrase phrase1() {
32         double[] phrase1data =
33             {JMC.G4, JMC.QN,
34              JMC.C5, JMC.HN, JMC.E5,JMC.EN, JMC.C5,JMC.EN,
35              JMC.E5,JMC.HN,JMC.D5,JMC.QN,
36              JMC.C5,JMC.HN,JMC.A4,JMC.QN,
37              JMC.G4,JMC.HN,JMC.G4,JMC.EN,JMC.A4,JMC.EN,
38              JMC.C5,JMC.HN,JMC.E5,JMC.EN,JMC.C5,JMC.EN,
39              JMC.E5,JMC.HN,JMC.D5,JMC.EN,JMC.E5,JMC.EN,
40              JMC.G5,JMC.DHN};

42     Phrase myPhrase = new Phrase();

```

```

        myPhrase.addNoteList(phrase1data);
44    }
45
46    public Phrase phrase2() {
47        double[] phrase2data =
48            {JMC.G5,JMC.HN,JMC.E5,JMC.EN,JMC.G5,JMC.EN,
49             JMC.G5,JMC.HN,JMC.E5,JMC.EN,JMC.C5,JMC.EN,
50             JMC.E5,JMC.HN,JMC.D5,JMC.QN,
51             JMC.C5,JMC.HN,JMC.A4,JMC.QN,
52             JMC.G4,JMC.HN,JMC.G4,JMC.EN,JMC.A4,JMC.EN,
53             JMC.C5,JMC.HN,JMC.E5,JMC.EN,JMC.C5,JMC.EN,
54             JMC.E5,JMC.HN,JMC.D5,JMC.QN,
55             JMC.C5,JMC.DHN
56         };
57
58        Phrase myPhrase = new Phrase();
59        myPhrase.addNoteList(phrase2data);
60        return myPhrase;
61    }
62
63
64    // Here are the two methods needed to make a linked list of elements
65    public void setNext(AmazingGraceSongElement2 nextOne){
66        this.next = nextOne;
67    }
68
69    public AmazingGraceSongElement2 next(){
70        return this.next;
71    }
72
73    // We could just access myPart directly
74    // but we can CONTROL access by using a method
75    // (called an accessor)
76    private Part part(){
77        return this.myPart;
78    }
79
80    // Why do we need this?
81    // If we want one piece to start after another, we need
82    // to know when the last one ends.
83    // Notice: It's the phrase that knows the end time.
84    // We have to ask the part for its phrase (assuming only one)
85    // to get the end time.
86    public double getEndTime(){
87        return this.myPart.getPhrase(0).getEndTime();
88    }
89
90    // We need setChannel because each part has to be in its
91    // own channel if it has different start times.
92    // So, we'll set the channel when we assemble the score.

```

```

94     private void setChannel(int channel){
95         myPart.setChannel(channel);
96     }
97
98     public void showFromMeOn(){
99         // Make the score that we'll assemble the elements into
100        // We'll set it up with the time signature and tempo we like
101        Score myScore = new Score("Amazing Grace");
102        myScore.setTimeSignature(3,4);
103        myScore.setTempo(120.0);
104
105        // Each element will be in its own channel
106        int channelCount = 1;
107
108        // Start from this element (this)
109        AmazingGraceSongElement2 current = this;
110        // While we're not through...
111        while (current != null)
112        {
113            // Set the channel, increment the channel, then add it in.
114            current.setChannel(channelCount);
115            channelCount = channelCount + 1;
116            myScore.addPart(current.part());
117
118            // Now, move on to the next element
119            // which we already know isn't null
120            current = current.next();
121        };
122
123        // At the end, let's see it!
124        View.notate(myScore);
125    }
126}

```

We can use this to do the flute for the first part and a piano for the second in much the same way as we did last time.

```

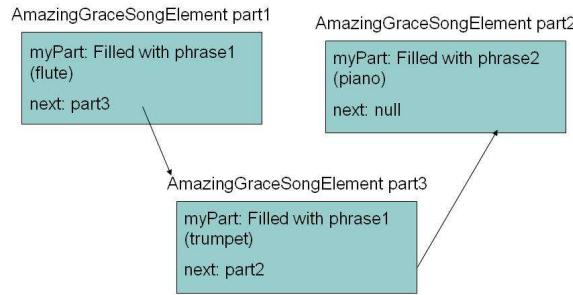
> import jm.JMC;
> AmazingGraceSongElement2 part1 = new AmazingGraceSongElement2();
> part1.setPhrase(part1.phrase1(), 0.0, JMC.FLUTE);
> AmazingGraceSongElement2 part2 = new AmazingGraceSongElement2();
> part2.setPhrase(part2.phrase2(), 22.0, JMC.PIANO);
> part1.setNext(part2);
> part1.showFromMeOn();

```

Let's go one step further, then make sure we understand what `showFromMeOn` is doing.

```
> AmazingGraceSongElement2 part3 = new AmazingGraceSongElement2();
> part3.setPhrase(part3.phrase1(), 0.0, JMC.TRUMPET);
> part1.setNext(part3);
> part3.setNext(part2);
> part1.showFromMeOn();
```

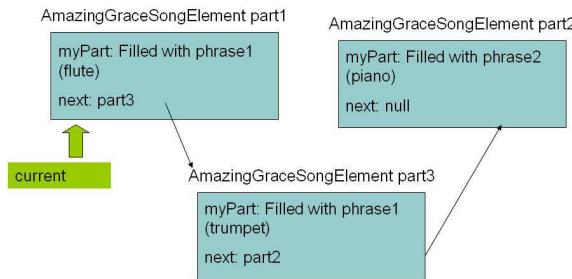
How it works: What we are doing here is to create a new part, part3, and to insert it *between* part1 and part2. We might think of the result as looking like this.



What is happening when we execute part1.showFromMeOn()? Let's trace it slowly.

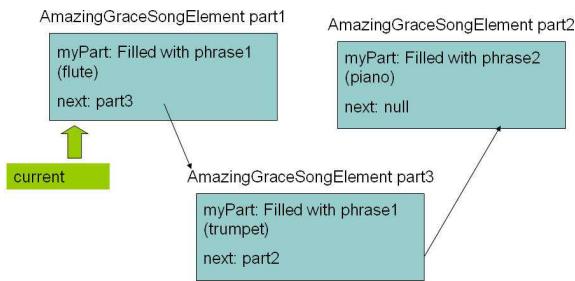
- First, we start out with current pointing at **this**, which is our part1. I like to think about traversing a linked list as being like pulling myself hand-over-hand on a ladder. Your right hand is current, and it's now holding on to the node1 rung of the ladder.

Step 1:
// Start from this element (this)
AmazingGraceSongElement2 current = this;



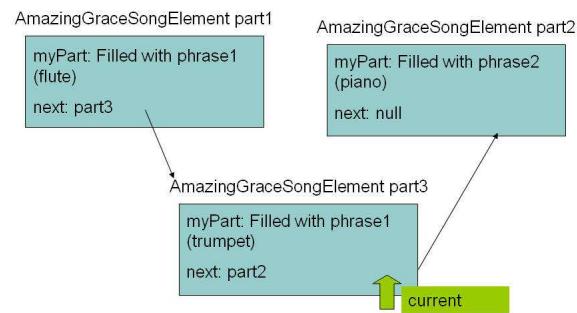
- Since current is not null (our right hand is holding something), we go ahead and process it.

Step 2:
 // While we're not through...
 while (current != null)
 { //BLAH BLAH BLAH – PROCESS THIS PART



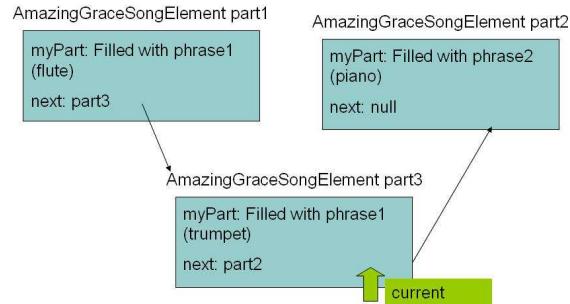
- We now feel out with our left hand for the rung connected to our current rung—that is a way of thinking about what `current.next()` is doing. Once we find the next rung with our left hand, we grab it with our right hand. We now have a new current node to process.

Step 3:
 // Now, move on to the next element
`current = current.next();`
 };



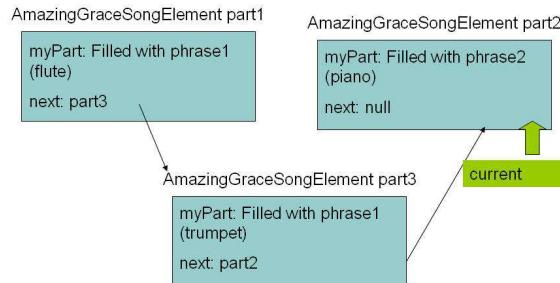
- This one isn't null either, so we process it.

```
Step 4:
// While we're not through...
while (current != null)
{ //BLAH BLAH BLAH - PROCESS THIS PART
```



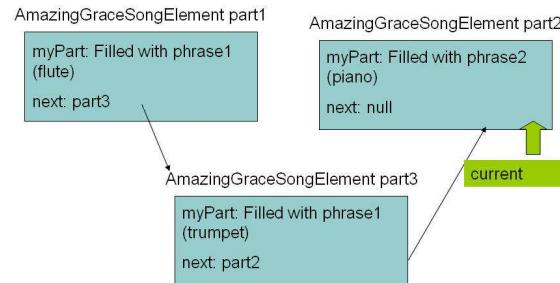
- Moving our left hand out to the next rung, then grabbing that new rung with our right, we have a new current node to process.

```
Step 5:
// Now, move on to the next element
current = current.next();
};
```



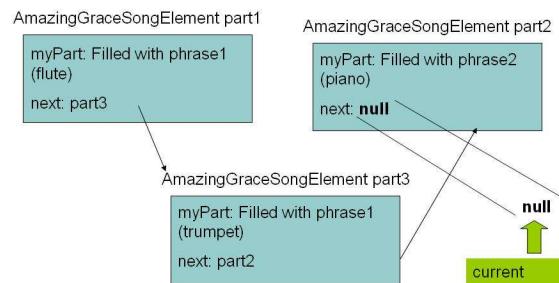
- Still not null, so let's process it.

```
Step 6:
// While we're not through...
while (current != null)
{ //BLAH BLAH BLAH - PROCESS THIS PART
```



- Now we reach out again with our left hand and find...nothing! We grab that nothing (`null`) with our right hand now, too. We're past the end of the ladder, er, linked list!

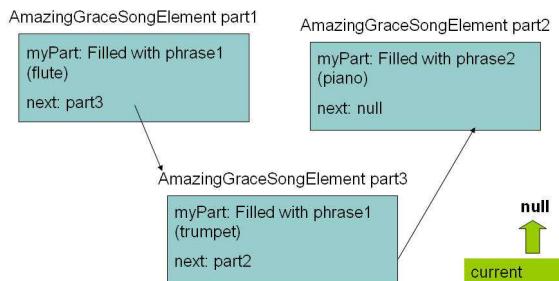
```
Step 7:
// Now, move on to the next element
current = current.next();
};
```



- We check at the top of the loop—yup, we're done.

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

STOP THE LOOP!



Now let's make a few observations about this code. Notice the `part2.phrase2()` expression. What would have happened if we did `part1.phrase2()` there instead? Would it have worked? (Go ahead, try it. We'll wait.) It would because both objects know the same `phrase1()` and `phrase2()` methods.

That doesn't really make a lot of sense, does it, in terms of what each object should know? Does every song element object need to know how to make every other song elements' phrase? We can get around this by creating a `static` method. Static methods are known to the `class`, not to the individual objects (`instances`). We'd write it something like this:

```
// First phrase of Amazing Grace
static public Phrase phrase1() {
```

```

double [ ] phrase1data =
{JMC.G4, JMC.QN,
 JMC.C5, JMC.HN, JMC.E5,JMC.EN, JMC.C5,JMC.EN,
 JMC.E5,JMC.HN,JMC.D5,JMC.QN,
 JMC.C5,JMC.HN,JMC.A4,JMC.QN,
 JMC.G4,JMC.HN,JMC.G4,JMC.EN,JMC.A4,JMC.EN,
 JMC.C5,JMC.HN,JMC.E5,JMC.EN,JMC.C5,JMC.EN,
 JMC.E5,JMC.HN,JMC.D5,JMC.EN,JMC.E5,JMC.EN,
 JMC.G5,JMC.DHN};

Phrase myPhrase = new Phrase();
myPhrase.addNoteList(phrase1data );
return myPhrase;
}

```

We'd actually use this method like this:

```

> import jm.JMC;
> AmazingGraceSongElement2 part1 = new AmazingGraceSongElement2();
> part1.setPhrase(AmazingGraceSongElement2.phrase1(), 0.0, JMC.FLUTE);

```

Now, that makes sense in an object-oriented kind of way: it's the *class* *AmazingGraceSongElement2* that knows about the phrases in the song *Amazing Grace*, not the instances of the class—not the different elements. But it's not really obvious that it's important for this to be about *Amazing Grace* at all! Wouldn't *any* song elements have basically this structure? Couldn't these phrases (now that they're in static methods) go in *any* class?

Generalizing SongElement and SongPhrase

Let's make a *generic* *SongElement* class, and a new class *SongPhrase* that we could stuff lots of phrases in.

Program
Example #32

Example Java Code: General Song Elements and Song Phrases

```

import jm.music.data.*;
2 import jm.JMC;
import jm.util.*;
4 import jm.music.tools.*;

6 public class SongElement {
    // Every element knows its next element and its part (of the score)
8     private SongElement next;
     private Part myPart;
10
    // When we make a new element, the next part is empty, and ours is a blank new part
12     public SongElement(){
        this.next = null;
    }
}

```

```

14     this.myPart = new Part();
15 }
16
17 // setPhrase takes a phrase and makes it the one for this element
18 // at the desired start time with the given instrument
19 public void setPhrase(Phrase myPhrase, double startTime, int instrument){
20     myPhrase.setStartTime(startTime);
21     this.myPart.addPhrase(myPhrase);
22     this.myPart.setInstrument(instrument);
23 }
24
25
26 // Here are the two methods needed to make a linked list of elements
27 public void setNext(SongElement nextOne){
28     this.next = nextOne;
29 }
30
31 public SongElement next(){
32     return this.next;
33 }
34
35 // We could just access myPart directly
36 // but we can CONTROL access by using a method
37 // (called an accessor)
38 private Part part(){
39     return this.myPart;
40 }
41
42 // Why do we need this?
43 // If we want one piece to start after another, we need
44 // to know when the last one ends.
45 // Notice: It's the phrase that knows the end time.
46 // We have to ask the part for its phrase (assuming only one)
47 // to get the end time.
48 public double getEndTime(){
49     return this.myPart.getPhrase(0).getEndTime();
50 }
51
52 // We need setChannel because each part has to be in its
53 // own channel if it has different start times.
54 // So, we'll set the channel when we assemble the score.
55 private void setChannel(int channel){
56     myPart.setChannel(channel);
57 }
58
59 public void showFromMeOn(){
60     // Make the score that we'll assemble the elements into
61     // We'll set it up with a default time signature and tempo we like
62     // (Should probably make it possible to change these — maybe with inputs?)
63     Score myScore = new Score("My Song");

```

```

64     myScore.setTimeSignature(3,4);
65     myScore.setTempo(120.0);
66
67     // Each element will be in its own channel
68     int channelCount = 1;
69
70     // Start from this element (this)
71     SongElement current = this;
72     // While we're not through...
73     while (current != null)
74     {
75         // Set the channel, increment the channel, then add it in.
76         current.setChannel(channelCount);
77         channelCount = channelCount + 1;
78         myScore.addPart(current.part());
79
80         // Now, move on to the next element
81         // which we already know isn't null
82         current = current.next();
83     };
84
85     // At the end, let's see it!
86     View.notate(myScore);
87
88 }
89
90 }

import jm.music.data.*;
2 import jm.JMC;
3 import jm.util.*;
4 import jm.music.tools.*;

6 public class SongPhrase {

8     // First phrase of Amazing Grace
9     static public Phrase AG1() {
10         double[] phrase1data =
11             {JMC.G4, JMC.QN,
12              JMC.C5, JMC.HN, JMC.E5, JMC.EN, JMC.C5, JMC.EN,
13               JMC.E5, JMC.HN, JMC.D5, JMC.QN,
14               JMC.C5, JMC.HN, JMC.A4, JMC.QN,
15               JMC.G4, JMC.HN, JMC.G4, JMC.EN, JMC.A4, JMC.EN,
16               JMC.C5, JMC.HN, JMC.E5, JMC.EN, JMC.C5, JMC.EN,
17               JMC.E5, JMC.HN, JMC.D5, JMC.EN, JMC.E5, JMC.EN,
18               JMC.G5, JMC.DHN};

20     Phrase myPhrase = new Phrase();
21     myPhrase.addNoteList(phrase1data);
22     return myPhrase;

```

```

24   }
25   // Second phrase of Amazing Grace
26   static public Phrase AG2() {
27     double[] phrase2data =
28       {JMC.G5,JMC.HN,JMC.E5,JMC.EN,JMC.G5,JMC.EN,
29        JMC.G5,JMC.HN,JMC.E5,JMC.EN,JMC.C5,JMC.EN,
30        JMC.E5,JMC.HN,JMC.D5,JMC.QN,
31        JMC.C5,JMC.HN,JMC.A4,JMC.QN,
32        JMC.G4,JMC.HN,JMC.G4,JMC.EN,JMC.A4,JMC.EN,
33        JMC.C5,JMC.HN,JMC.E5,JMC.EN,JMC.C5,JMC.EN,
34        JMC.E5,JMC.HN,JMC.D5,JMC.QN,
35        JMC.C5,JMC.DHN
36     };
37
38     Phrase myPhrase = new Phrase();
39     myPhrase.addNoteList(phrase2data);
40     return myPhrase;
41   }
42 }
```

We can use this like this:

```

> import jm.JMC;
> SongElement part1 = new SongElement();
> part1.setPhrase(SongPhrase.AG1(), 0.0, JMC.FLUTE);
> SongElement part2 = new SongElement();
> part2.setPhrase(SongPhrase.AG2(), 22.0, JMC.PIANO);
> part1.setNext(part2);
> part1.showFromMeOn();
```

We now have a structure to do more songs and more general explorations.

Adding More Phrases

Example Java Code: **More phrases to play with**

*Program
Example #33*

```

import jm.music.data.*;
2 import jm.JMC;
3 import jm.util.*;
4 import jm.music.tools.*;

6 public class SongPhrase {
```



```
58     JMC.A3,JMC.DHN} ;  
59  
60     Phrase myPhrase = new Phrase();  
61     myPhrase.addNoteList(phrasedata);  
62     return myPhrase;  
63 }  
64  
65 //Little Riff1  
66 static public Phrase riff1() {  
67     double[] phrasedata =  
68     {JMC.G3,JMC.EN,JMC.B3,JMC.EN,JMC.C4,JMC.EN,JMC.D4,JMC.EN};  
69  
70     Phrase myPhrase = new Phrase();  
71     myPhrase.addNoteList(phrasedata);  
72     return myPhrase;  
73 }  
74  
75 //Little Riff2  
76 static public Phrase riff2() {  
77     double[] phrasedata =  
78     {JMC.D4,JMC.EN,JMC.C4,JMC.EN,JMC.E4,JMC.EN,JMC.G4,JMC.EN};  
79  
80     Phrase myPhrase = new Phrase();  
81     myPhrase.addNoteList(phrasedata);  
82     return myPhrase;  
83 }  
84  
85 //Little Riff3  
86 static public Phrase riff3() {  
87     double[] phrasedata =  
88     {JMC.C4,JMC.QN,JMC.E4,JMC.EN,JMC.G4,JMC.EN,JMC.E4,JMC.SN,  
89     JMC.G4,JMC.SN,JMC.E4,JMC.SN,JMC.G4,JMC.SN,JMC.C4,JMC.QN};  
90  
91     Phrase myPhrase = new Phrase();  
92     myPhrase.addNoteList(phrasedata);  
93     return myPhrase;  
94 }  
95  
96 //Little Riff4  
97 static public Phrase riff4() {  
98     double[] phrasedata =  
99     {JMC.C4,JMC.QN,JMC.E4,JMC.QN,JMC.G4,JMC.QN,JMC.C4,JMC.QN};  
100  
101     Phrase myPhrase = new Phrase();  
102     myPhrase.addNoteList(phrasedata);  
103     return myPhrase;  
104 }  
105  
106 }
```

108 }

```
> SongElement house = new SongElement();
> house.setPhrase(SongPhrase.house(), 0.0, JMC.HARMONICA);
> house.showFromMeOn();

> SongElement riff1 = new SongElement();
> riff1.setPhrase(SongPhrase.riff1(), 0.0, JMC.HARMONICA);
> riff1.showFromMeOn();
> SongElement riff2 = new SongElement();
> riff2.setPhrase(SongPhrase.riff2(), 0.0, JMC.TENOR_SAX);
> riff2.showFromMeOn();
```

But music is really about repetition and playing off pieces and variations. Try something like this (Figure 6.11).

```
> SongElement riff1 = new SongElement();
> riff1.setPhrase(SongPhrase.riff1(), 0.0, JMC.HARMONICA);
> riff1.showFromMeOn();
-- Constructing MIDI file from 'My Song' ... Playing with JavaSound ... Completed MIDI
> SongElement riff2 = new SongElement();
> riff2.setPhrase(SongPhrase.riff2(), 0.0, JMC.TENOR_SAX);
> riff2.showFromMeOn();
-- Constructing MIDI file from 'My Song' ... Playing with JavaSound ... Completed MIDI
> riff2.getEndTime()
2.0
> SongElement riff4 = new SongElement();
> riff4.setPhrase(SongPhrase.riff1(), 2.0, JMC.TENOR_SAX);
> SongElement riff5 = new SongElement();
> riff5.setPhrase(SongPhrase.riff1(), 4.0, JMC.TENOR_SAX);
> SongElement riff6 = new SongElement();
> riff6.setPhrase(SongPhrase.riff2(), 4.0, JMC.HARMONICA);
> SongElement riff7 = new SongElement();
> riff7.setPhrase(SongPhrase.riff1(), 6.0, JMC.JAZZ_GUITAR);
> riff1.setNext(riff2);
> riff2.setNext(riff4);
> riff4.setNext(riff5);
> riff5.setNext(riff6);
> riff6.setNext(riff7);
> riff1.showFromMeOn();
```

Computing phrases

If we need some repetition, we don't have to type things over and over again—we can ask the computer to do it for us! Our phrases in **class** SongPhrase don't have to come from constants. It's okay if they are computed phrases.

We can use steel drums (or something else, if we want) to create rhythm.

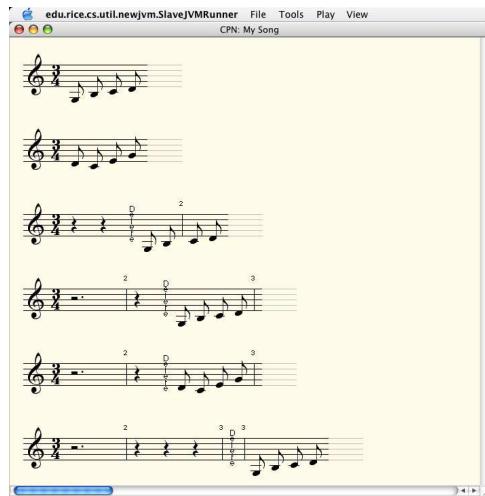


Figure 6.11: Playing some different riffs in patterns

```
> SongElement steel = new SongElement();
> steel.setPhrase(SongPhrase.riff1(), 0.0, JMC.STEEL_DRUM);
> steel.showFromMeOn();
```

Example Java Code: **Computed Phrases**

*Program
Example #34*

```
//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};

    int counter1;
    int counter2;

    Phrase myPhrase = new Phrase();
    // 3 of riff1, 1 of riff2, and repeat all of it 3 times
    for (counter1 = 1; counter1 <= 3; counter1++)
    {for (counter2 = 1; counter2 <= 3; counter2++)
        myPhrase.addNoteList(riff1data);
    myPhrase.addNoteList(riff2data);
    };
    return myPhrase;
}
```

```

//Larger Riff2
static public Phrase pattern2() {
    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};

    int counter1;
    int counter2;

    Phrase myPhrase = new Phrase();
    // 2 of riff1, 2 of riff2 , and repeat all of it 3 times
    for (counter1 = 1; counter1 <= 3; counter1++)
    {for (counter2 = 1; counter2 <= 2; counter2++)
        myPhrase.addNoteList(riff1data);
    for (counter2 = 1; counter2 <= 2; counter2++)
        myPhrase.addNoteList(riff2data);
    };
    return myPhrase;
}

//Rhythm Riff
static public Phrase rhythm1() {
    double[] riff1data =
{JMC.G3,JMC.EN,JMC.REST,JMC.HN,JMC.D4,JMC.EN};
    double[] riff2data =
{JMC.C3,JMC.QN,JMC.REST,JMC.QN};

    int counter1;
    int counter2;

    Phrase myPhrase = new Phrase();
    // 2 of rhythm riff1 , 2 of rhythm riff2 , and repeat all of it 3 times
    for (counter1 = 1; counter1 <= 3; counter1++)
    {for (counter2 = 1; counter2 <= 2; counter2++)
        myPhrase.addNoteList(riff1data);
    for (counter2 = 1; counter2 <= 2; counter2++)
        myPhrase.addNoteList(riff2data);
    };
    return myPhrase;
}

> import jm.JMC;
> SongElement sax1 = new SongElement();
> sax1.setPhrase(SongPhrase.pattern1(), 0.0, JMC.TENOR_SAX);
> sax1.showFromMeOn();
-- Constructing MIDI file from'My Song'... Playing with JavaSound ... Completed MIDI

```

```

> SongElement sax2 = new SongElement();
> sax2.setPhrase(SongPhrase.pattern2(), 0.0, JMC.TENOR_SAX);
> sax2.showFromMeOn()
-- Constructing MIDI file from 'My Song'... Playing with JavaSound ... Completed MIDI playback
> sax1.setNext(sax2);
> sax1.showFromMeOn();
-- Constructing MIDI file from 'My Song'... Playing with JavaSound ... Completed MIDI playback
> sax1.setNext(null); // I decided I didn't like it.
> SongElement rhythm1=new SongElement();
> rhythm1.setPhrase(SongPhrase.rhythm1(), 0.0, JMC.STEEL_DRUM);
> sax1.setNext(rhythm1); // I put something else with the sax
> sax1.showFromMeOn();
-- Constructing MIDI file from 'My Song'... Playing with JavaSound ... Completed MIDI playback

```

Here's what the sax plus rhythm looked like (Figure 6.12).



Figure 6.12: Sax line in the top part, rhythm in the bottom

Computer Science Idea: Layering software makes it easier to change, Part 2

Notice that all our Editor Pane interactions now are with SongPhrase. We don't have to change SongElements anymore—they work, so now we can ignore them. We're not dealing with Phrases and Parts anymore, either. As we develop layers, if we do it right, we only have to deal with one layer at a time (Figure 6.13).

If we're computing phrases, how about if we compute from random notes on up?

Example Java Code: **10 random notes SongPhrase**

*Program
Example #35*

```

/*
 * 10 random notes
 */
static public Phrase random() {

```

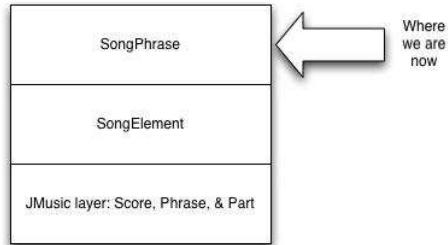


Figure 6.13: We now have layers of software, where we deal with only one at a time

```

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;
}

```

How it works: Math.random() returns a number between 0.0 and 1.0. There are 128 possible notes in MIDI. Multiplying 128*Math.random() gives us a note between 0 and 127. These are 10 completely random notes.

Complete randomness isn't the most pleasant thing to listen to. We can control the randomness a bit, mathematically.

*Program
Example #36*

Example Java Code: **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;
}

```

How it works: Here, we generate a random number between 0 and 4 by multiplying `Math.random()` by 5. Recall that note 60 is middle C. If we add our random number to 60, we play one of the five notes just above middle C.

We obviously can keep going from there. Perhaps we might generate a random number, then use it to choose (as if flipping a coin) between some different phrases to combine. Or perhaps we use arithmetic to only choose among certain notes (not necessarily in a range—perhaps selected from an array) that we want in our composition. It is really possible to have a computer “compose” music driven by random numbers.

6.7 Exploring Music

What we’ve built for music exploration is *okay*, but not great. What’s wrong with it?

- It’s hard to use. We have to specify each phrase’s start time and instrument. That’s a lot of specification, and it doesn’t correspond to how musicians tend to think about music structure. More typically, musicians see a single music *part* as having a single instrument and start time (much as the structure of the class `Part` in the underlying JMusic classes).
- While we have a linked list for connecting the elements of our songs, we don’t *use* the linked list for anything. Because each element has its own start time, there is no particular value to having an element before or after any other song element.

The way we’re going to address these problems is by a *refactoring*. We are going to *move* a particular aspect of our design to another place in our design. Currently, every instance of `SongElement` has its own `Part` instance—that’s why we specify the instrument and start time when we create the `SongElement`. What if we move the creation of the part until we collect all the `SongElement` phrases? Then we don’t have to specify the instrument and start time until later. What’s more, the ordering of the linked list will define the ordering of the note phrases.

Computer Science Idea: Refactoring refines a design.

We refactor designs in order to improve them. Our early decisions about where to what aspect of a piece of software might prove to be inflexible or downright *wrong* (in the sense of not describing what we want to de-

scribe) as we continue to work. Refactoring is a process of simplifying and improving a design.

There is a cost to this design. There will be only *one* instrument and start time associated with a list of song elements. We'll correct that problem in the next section.

We're going to rewrite our SongElement class for this new design, and we're going to give it a fairly geeky, abstract name—in order to make a point. We're going to name our class **SongNode** to highlight that each element in the song is now a *node* in a *list* of song elements. Computer scientists typically use the term *node* to describe pieces in a list or *tree*.

*Program
Example #37*

Example Java Code: **SongNode class**

```

import jm.music.data.*;
2 import jm.JMC;
import jm.util.*;
4 import jm.music.tools.*;

6 public class SongNode {
    /**
     * the next SongNode in the list
     */
10 private SongNode next;
    /**
     * the Phrase containing the notes and durations associated with this node
     */
14 private Phrase myPhrase;

16 /**
     * When we make a new element, the next part is empty, and ours is a blank new par
18 */
20 public SongNode(){
    this.next = null;
    this.myPhrase = new Phrase();
22 }

24 /**
     * setPhrase takes a Phrase and makes it the one for this node
26 * @param thisPhrase the phrase for this node
     */
28 public void setPhrase(Phrase thisPhrase){
    this.myPhrase = thisPhrase;
30 }

```

```

32  /*
33   * Creates a link between the current node and the input node
34   * @param nextOne the node to link to
35   */
36  public void setNext(SongNode nextOne){
37      this.next = nextOne;
38  }
39
40  /*
41   * Provides public access to the next node.
42   * @return a SongNode instance (or null)
43   */
44  public SongNode next(){
45      return this.next;
46  }
47
48  /*
49   * Accessor for the node's Phrase
50   * @return internal phrase
51   */
52  private Phrase getPhrase(){
53      return this.myPhrase;
54  }
55
56  /*
57   * Accessor for the notes inside the node's phrase
58   * @return array of notes and durations inside the phrase
59   */
60  private Note [] getNotes(){
61      return this.myPhrase.getNoteArray();
62  }
63
64  /*
65   * Collect all the notes from this node on
66   * in an part (then a score) and open it up for viewing.
67   * @param instrument MIDI instrument (program) to be used in playing this list
68   */
69  public void showFromMeOn(int instrument){
70      // Make the Score that we'll assemble the elements into
71      // We'll set it up with a default time signature and tempo we like
72      // (Should probably make it possible to change these — maybe with inputs?)
73      Score myScore = new Score("My Song");
74      myScore.setTimeSignature(3,4);
75      myScore.setTempo(120.0);
76
77      // Make the Part that we'll assemble things into
78      Part myPart = new Part(instrument);
79
80      // Make a new Phrase that will contain the notes from all the phrases

```

```

82     Phrase collector = new Phrase();
84
85     // Start from this element (this)
86     SongNode current = this;
87     // While we're not through...
88     while (current != null)
89     {
90         collector.addNoteList(current.getNotes());
91
92         // Now, move on to the next element
93         current = current.next();
94     };
95
96     // Now, construct the part and the score.
97     myPart.addPhrase(collector);
98     myScore.addPart(myPart);
99
100    // At the end, let's see it!
101    View.notate(myScore);
102
103}
104

```

We can use this new class to do some of the things that we did before (Figure 6.14).

```

> SongNode first = new SongNode();
> first.setPhrase(SongPhrase.riff1());
> import jm.JMC; // We'll need this!
> first.showFromMeOn(JMC.FLUTE); // We can play with just one node
-- Constructing MIDI file from 'My Song'... Playing with JavaSound
... Completed MIDI playback -----
> SongNode second = new SongNode();
> second.setPhrase(SongPhrase.riff2());
> first.next(second); // OOPS!
Error: No 'next' method in 'SongNode' with arguments: (SongNode)
> first.setNext(second);
> first.showFromMeOn(JMC.PIANO);

```

Remember the documentation for the JMusic classes that we saw earlier in the book? That documentation can actually be automatically generated from the comments that we provide. *Javadoc* is the name for the specialized commenting structure and the tool that generates HTML documentation from that structure. The commenting structure is: (XXX-TO-DO See DrJava docs for now.) (Figure 6.15)

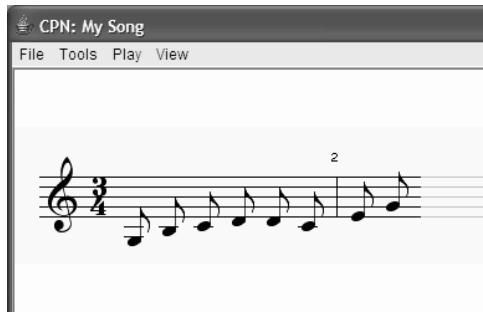


Figure 6.14: First score generated from ordered linked list



Figure 6.15: Javadoc for the class SongNode

Now Let's Play!

Now we can really play with repetition and weaving in at regular intervals—stuff of real music! Let's create two new methods: One that repeats an input phrase several times, and one that weaves in a phrase every n nodes.

Example Java Code: Repeating and weaving methods

*Program
Example #38*

```

1  /*
2   * copyNode returns a copy of this node
3   * @return another song node with the same notes
4   */

```

```

public SongNode copyNode(){
6   SongNode returnMe = new SongNode();
7   returnMe.setPhrase(this.getPhrase());
8   return returnMe;
9 }
10
11 /**
12  * Repeat the input phrase for the number of times specified.
13  * It always appends to the current node, NOT insert.
14  * @param nextOne node to be copied in to list
15  * @param count number of times to copy it in.
16  */
17 public void repeatNext(SongNode nextOne,int count) {
18   SongNode current = this; // Start from here
19   SongNode copy; // Where we keep the current copy
20
21   for (int i=1; i <= count; i++)
22   {
23     copy = nextOne.copyNode(); // Make a copy
24     current.setNext(copy); // Set as next
25     current = copy; // Now append to copy
26   }
27 }
28
29 /**
30  * Insert the input SongNode AFTER this node,
31  * and make whatever node comes NEXT become the next of the input node.
32  * @param nextOne SongNode to insert after this one
33  */
34 public void insertAfter(SongNode nextOne)
35 {
36   SongNode oldNext = this.next(); // Save its next
37   this.setNext(nextOne); // Insert the copy
38   nextOne.setNext(oldNext); // Make the copy point on to the rest
39 }
40
41 /**
42  * Weave the input phrase count times every skipAmount nodes
43  * @param nextOne node to be copied into the list
44  * @param count how many times to copy
45  * @param skipAmount how many nodes to skip per weave
46  */
47 public void weave(SongNode nextOne, int count, int skipAmount)
48 {
49   SongNode current = this; // Start from here
50   SongNode copy; // Where we keep the one to be weaved in
51   SongNode oldNext; // Need this to insert properly
52   int skipped; // Number skipped currently
53 }
54

```

```

56   for (int i=1; i <= count; i++)
57   {
58     copy = nextOne.copyNode(); // Make a copy
59
59   //Skip skipAmount nodes
60   skipped = 1;
61   while ((current.next() != null) && (skipped < skipAmount))
62   {
63     current = current.next();
64     skipped++;
65   }
66
67   if (current.next() == null) // Did we actually get to the end early?
68   break; // Leave the loop
69
70   oldNext = current.next(); // Save its next
71   current.insertAfter(copy); // Insert the copy after this one
72   current = oldNext; // Continue on with the rest
73 }
74 }
```

First, let's make 15 copies of one pattern (Figure 6.16).

```

> import jm.JMC;
> SongNode first = new SongNode();
> SongNode riff1 = new SongNode();
> riff1.setPhrase(SongPhrase.riff1());
> first.repeatNext(riff1,15);
> first.showFromMeOn(JMC.FLUTE);
```

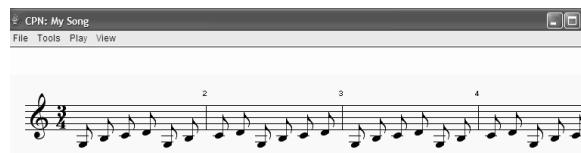


Figure 6.16: Repeating a node several times

Now, let's weave in a second pattern every-other (off by 1) node, for seven times (Figure 6.17).

```

> SongNode riff2 = new SongNode();
> riff2.setPhrase(SongPhrase.riff2());
> first.weave(riff2,7,1);
> first.showFromMeOn(JMC.PIANO);
```

And we can keep weaving in more.

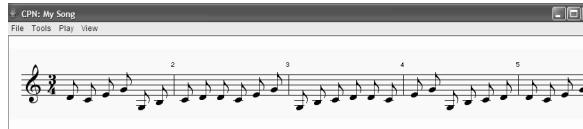


Figure 6.17: Weaving a new node among the old

```
> SongNode another = new SongNode();
> another.setPhrase(SongPhrase.rhythml());
> first.weave(another,10,2);
> first.showFromMeOn(JMC.STEEL_DRUMS);
```

Now, repeatNext is not the most polite method in the world. Consider what happens if we call it on node1 and node1 *already has a next!*. The rest of the list simply gets blown away! But now that we have insertAfter, we can produce a more friendly and polite version, repeatNextInserting, which preserves the rest of the list.

*Program
Example #39*

Example Java Code: **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
    }
}
```

Linked Lists versus Arrays

What are the advantages of using linked lists here, rather than arrays? They are *not* all in the favor of linked lists!

How complicated is it to traverse a linked list (visit all the elements) versus an array? Here's a linked list traversal:

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

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

Basically, we're walking hand-over-hand across all the nodes in the list. Think of `current` as your right hand.

- We put our right hand (`current`) on the node at **this**.
- Is our right hand empty? **while** (`current= null`)!?
- Process our right hand.
- Then with your left hand feel down the next link to the next node—that's what `current.next()` is doing.
- Now, grab with your right hand whatever your left hand was holding—`current = current.next()`.
- Back to the top of the loop to see if our right hand is empty.
- When we reach the end of the list, our right hand is holding nothing.

Traversing an array is much easier: it's just a **for** loop.

```
> // Now, traverse the array and gather them up.
> Phrase myphrase = new Phrase()
> for (int i=0; i<100; i++)
    {myphrase.addNote( someNotes[i]); }
```

But what if we want to change something in the middle? That's where linked lists shine. Here's inserting something into the middle of a linked list:

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

You know that those `setNext` calls are just a single line of code.

How about inserting into the middle of an array? We saw that in the last chapter and below. This code is not only long and complicated, but it

is also slow. Insertion into a linked list is $O(1)$ —it always takes the same amount of time, no matter how big the things are being inserted. Insertion into the middle of an array (presuming that you move things over to make room, like the insertion in the linked list does) is $O(n)$. That will always be slower.

```
public void insertAfter(Sound inSound, int start){

    SoundSample current=null;
    // Find how long insound is
    int amtToCopy = inSound.getLength();
    int endOfThis = this.getLength()-1;
    // If too long, copy only as much as will fit
    if (start + amtToCopy > endOfThis)
        {amtToCopy = endOfThis-start -1};

    // ** First, clear out room.
    // Copy from endOfThis-amtToCopy up to endOfThis
    for (int i=endOfThis-amtToCopy; i > start ; i--)
    {
        current = this.getSample(i);
        current.setValue(this.getSampleValueAt(i+amtToCopy));
    }

    //** Second, copy in inSound up to amtToCopy
    for (int target=start,source=0;
          source < amtToCopy;
          target++, source++) {
        current = this.getSample(target);
        current.setValue(inSound.getSampleValueAt(source));
    }
}
```

Which one is more memory efficient, that is, stores the same information in less memory? Arrays are more efficient, certainly. For every element in the linked list, there is additional memory needed to keep track of “And here’s the next one.” It is really quite clear which note follows which other note in an array.

On the other hand, if you do not know the size of the thing that you want before you get started—if maybe you will have dozens of notes one time, and hundreds the next—then linked lists have a distinct advantage. Arrays cannot grow, nor shrink. They simply are the size that they are. Typically, then, you make your arrays larger than you think that you will need—which is a memory inefficiency of its own. Linked lists can grow or shrink as needed.

Creating a Music Tree

Now, let's get back to the problem of having multiple parts, something we lost when we went to the ordered linked list implementation. We'll create a SongPart class that will store the instrument and the start of a SongPhrase list. Then we'll create a Song class that will store multiple parts—two parts, each a list of nodes. This structure is a start toward a *tree* structure.

Example Java Code: **SongPart class**

*Program
Example #40*

```

1  import jm.music.data.*; import jm.JMC; import jm.util.*; import
2  jm.music.tools.*;

4  public class SongPart {

6      /*
7       * SongPart has a Part
8       */
9      public Part myPart;
10     /*
11      * SongPart has a SongNode that is the beginng of its
12      */
13     public SongNode myList;
14

15    /**
16     * Construct a SongPart
17     * @param instrument MIDI instrument (program)
18     * @param startNode where the song list starts from
19     */
20    public SongPart(int instrument, SongNode startNode)
21    {
22        myPart = new Part(instrument);
23        myList = startNode;
24    }

25    /**
26     * Collect parts of this SongPart
27     */
28    public Phrase collect(){
29        return this.myList.collect(); // delegate to SongNode's collect
30    }

31    /**
32     * Collect all notes in this SongPart and open it up for viewing.
33     */
34    public void show(){
35        // Make the Score that we'll assemble the part into
36    }
}

```

```

38   // We'll set it up with a default time signature and tempo we like
39   // (Should probably make it possible to change these — maybe with inputs?)
40   Score myScore = new Score("My Song");
41   myScore.setTimeSignature(3,4);
42   myScore.setTempo(120.0);

44   // Now, construct the part and the score.
45   this.myPart.addPhrase(this.collect());
46   myScore.addPart(this.myPart);

48   // At the end, let's see it!
49   View.notate(myScore);
50 }

52 }

```

*Program
Example #41*

Example Java Code: **Song class—root of a tree-like music structure**

```

1 import jm.music.data.*; import jm.JMC; import jm.util.*; import
2 jm.music.tools.*;

4 public class Song {
5   /**
6    * first Channel
7    */
8   public SongPart first;

10  /**
11   * second Channel
12   */
13  public SongPart second;

14  /**
15   * Take in a SongPart to make the first channel in the song
16   */
17  public void setFirst(SongPart channel1){
18    first = channel1;
19    first.myPart.setChannel(1);
20  }

22  /**
23   * Take in a SongPart to make the second channel in the song
24   */
25  public void setSecond(SongPart channel2){
26

```

```

28     second = channel2;
29     first.myPart.setChannel(2);
30   }
31
32   public void show(){
33     // Make the Score that we'll assemble the parts into
34     // We'll set it up with a default time signature and tempo we like
35     // (Should probably make it possible to change these — maybe with inputs?)
36     Score myScore = new Score("My Song");
37     myScore.setTimeSignature(3,4);
38     myScore.setTempo(120.0);
39
40     // Now, construct the part and the score.
41     first.myPart.addPhrase(first.collect());
42     second.myPart.addPhrase(second.collect());
43     myScore.addPart(first.myPart);
44     myScore.addPart(second.myPart);
45
46     // At the end, let's see it!
47     View.notate(myScore);
48   }
49
50 }
```

While our new structure is very flexible, it's not the easiest thing to use. We don't want to have to type everything into the Interactions Pane every time. So, we'll create a class that has its main method that will run on its own. You can execute it using RUN DOCUMENT'S MAIN METHOD (F2) in the TOOLS menu. Using MySong, we can get back to having multi-part music in a single score (Figure 6.18).

Example Java Code: **MySong** class with a main method

*Program
Example #42*

```

1  import jm.music.data.*;
2  import jm.JMC;
3  import jm.util.*;
4  import jm.JMC;
5
6  public class MyFirstSong {
7    public static void main(String [] args) {
8      Song songroot = new Song();
9
10     SongNode node1 = new SongNode();
```

```

12     SongNode riff3 = new SongNode();
13     riff3.setPhrase(SongPhrase.riff3());
14     node1.repeatNext(riff3, 16);
15
16     SongNode riff1 = new SongNode();
17     riff1.setPhrase(SongPhrase.riff1());
18     node1.weave(riff1, 7, 1);
19     SongPart part1 = new SongPart(JMC.PIANO, node1);
20
21     songroot.setFirst(part1);
22
23     SongNode node2 = new SongNode();
24     SongNode riff4 = new SongNode();
25     riff4.setPhrase(SongPhrase.riff4());
26     node2.repeatNext(riff4, 20);
27     node2.weave(riff1, 4, 5);
28     SongPart part2 = new SongPart(JMC.STEEL_DRUMS, node2);
29
30     songroot.setSecond(part2);
31     songroot.show();
32 }
```



Figure 6.18: Multi-part song using our classes

The point of all of this is to create a *structure* which enables us easily to explore music compositions, in the ways that we will most probably want to explore. We imagine that most music composition exploration will consist of defining new phrases of notes, then combining them in interesting ways: defining which come after which, repeating them, and weaving them in with the rest. At a later point, we can play with which instruments we want to use to play our parts.

Exercises

1. The Song structure that we've developed on *top* of JMusic is actually pretty similar to the actual implementation of the classes Score, Part, and Phrase *within* the JMusic system. Take one of the music exam-

ples that we've built with our own linked list, and re-implement it using only the JMusic classes.

2. Add into Song the ability to record different starting times for the composite SongParts. It's the internal Phrase that remembers the start time, so you'll have to pass it down the structure.
3. The current implementation of repeatAfter in SongNode append's the input node, as opposed to inserting it. If you could insert it, then you could repeat a bunch of a given phrase *between* two other nodes. Create a repeatedInsert method that does an insertion rather than an append.
4. The current implementation of Song implements *two* channels. Channel *nine* is the *MIDI Drum Kit* where the notes are different percussion instruments (Figure 6.2). Modify the Song class take a third channel, which gets assigned to MIDI channel 9 and plays a percussion SongPart.

35	Acoustic Bass Drum	51	Ride Cymbal 1
36	Bass Drum 1	52	Chinese Cymbal
37	Side Stick	53	Ride Bell
38	Acoustic Snare	54	Tambourine
39	Hand Clap	55	Splash Cymbal
40	Electric Snare	56	Cowbell
41	Lo Floor Tom	57	Crash Cymbal 2
42	Closed Hi Hat	58	Vibraslap
43	Hi Floor Tom	59	Ride Cymbal 2
44	Pedal Hi Hat	60	Hi Bongo
45	Lo Tom Tom	61	Low Bongo
46	Open Hi Hat	62	Mute Hi Conga
47	Low -Mid Tom Tom	63	Open Hi Conga
48	Hi Mid Tom Tom	64	Low Conga
49	Crash Cymbal 1	65	Hi Timbale
50	Hi Tom Tom	66	Lo Timbale

Table 6.2: MIDI Drum Kit Notes

5. Using the methods developed in class to play with linked list of music, create a song.
 - You must use weave and repeatNext (or repeatNextInserting) to create patterns in your music. You probably want to use the SongNode and SongPhrase classes.
 - You must use weave and either of the repeats *at least five* in your piece. In other words, repeat a set of nodes, then repeat

another set, then repeat another set. Then weave in nodes with one pattern, then weave in nodes with another pattern. That would be five. Or do one repeat to create a basic tempo, then four weaves to bring in other motifs.

- You must use at least four unique riffs from SongPhrase. (It's okay for you to make all four of them yourself.)
- You must also create your own riffs in SongPhrase. You can simply type in music, or you can compute your riff any way you want to do it is fine. Just create at least one phrase (of more than a couple notes) that is interesting and unique.
- All told, you must have at least 10 nodes in your final song.

You will create a class (with some cunning name like MyWovenSong) with a main that will assemble your song, then open it with showFromMeOn (or showwhatever you need to do open up the notation View on your masterpiece).

Here's the critical part: Draw a picture of your resultant list structure. Show us where all the nodes are in your final composition. You can do this by drawing with a tool like Paint or Visio or even PowerPoint, or you can draw it on paper then scan it in. You can turn in JPEG, TIFF, or PPT files.

7 Structuring Images using Linked Lists

Chapter Learning Objectives

The villagers and wildebeests scenes are not single images. They are collections of many images—not just of the villagers and wildebeests themselves, but of the elements of the scene, too. How do we structure these images? We could use arrays, but that doesn't give us enough flexibility to insert new pictures, make things disappear (delete them from the scene), and move elements around. To do that, we will need *linked lists* of images.

The computer science goals for this chapter are:

- To use and manipulate linked lists (of images).
- To use an *abstract superclass* to create a single linked list of multiple kinds of objects.

The media learning goals for this chapter are:

- To use different interpretations of the linearity of linked lists: to represent left-to-right ordering, or to represent front-to-back ordering (or *layering*).

We know a lot about manipulating individual images. We know how to manipulate the pixels of an image to create various effects. We've encapsulated a bunch of these in methods to make them pretty easy to use. The question is how to build up these images into composite images. How do we create *scenes* made up of lots of images?

When computer graphics and animation professionals construct complicated scenes such as in *Toy Story* and *Monsters, Inc.*, they go beyond thinking about individual images. Certainly, at some point, they care about how Woody and Nemo are created, how they look, and how they get inserted into the frame—but all as part of how the *scene* is constructed.

How do we describe the structure of a scene? How do we structure our objects in order to describe scenes that we want to describe, but what's more, how do we describe them in such a way that we can change the scene (e.g., in order to define an animation) in the ways that we'll want to later? Those are the questions of this chapter.

7.1 Simple arrays of pictures

The simplest thing to do is to simply list all the pictures we want in array. We then compose them each into a background (Figure 7.1).

```
> 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("flower1.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();
```



Figure 7.1: Array of pictures composed into a background

7.2 Listing the Pictures, Left-to-Right

We met a *linked list* in the last chapter. We can use the same concept for images.

Let's start out by thinking about a scene as a collection of pictures that lay next to one another. Each element of the scene is a picture and knows the next element in the sequence. The elements form a *list* that is *linked* together—that's a *linked list*.

We'll use three little images drawn on a blue background, to make them easier to chromakey into the image (Figure 7.2).

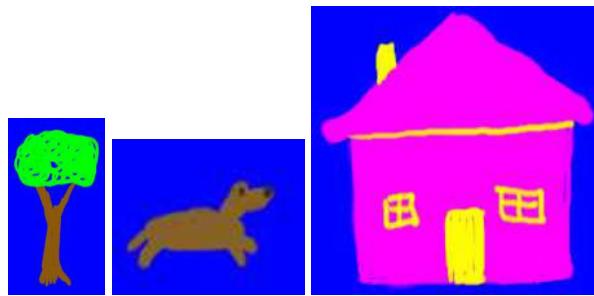


Figure 7.2: Elements to be used in our scenes

Example Java Code: **Elements of a scene in position order**

*Program
Example #43*

```

public class PositionedSceneElement {
    private Picture myPic;
    private PositionedSceneElement next;
    public PositionedSceneElement(Picture heldPic){
        myPic = heldPic;
        next = null;
    }
    public void setNext(PositionedSceneElement nextOne){

```

```

28     this.next = nextOne;
29 }
30
31     public PositionedSceneElement getNext(){
32         return this.next;
33     }
34
35     /**
36      * Returns the picture in the node.
37      * @return the picture in the node
38      */
39     public Picture getPicture(){
40         return this.myPic;
41     }
42
43     /**
44      * Method to draw from this node on in the list , using bluescreen.
45      * Each new element has it's lower-left corner at the lower-right
46      * of the previous node. Starts drawing from left-bottom
47      * @param bg Picture to draw drawing on
48      */
49     public void drawFromMeOn(Picture bg) {
50         PositionedSceneElement current;
51         int currentX=0, currentY = bg.getHeight()-1;
52
53         current = this;
54         while (current != null)
55         {
56             current.drawMeOn(bg, currentX, currentY);
57             currentX = currentX + current.getPicture().getWidth();
58             current = current.getNext();
59         }
60     }
61
62     /**
63      * Method to draw from this picture , using bluescreen .
64      * @param bg Picture to draw drawing on
65      * @param left x position to draw from
66      * @param bottom y position to draw from
67      */
68
69     private void drawMeOn(Picture bg, int left, int bottom) {
70         // Bluescreen takes an upper left corner
71         this.getPicture().bluescreen(bg, left,
72                                     bottom-this.getPicture().getHeight());
73     }
74 }
```

* * *

To construct a scene, we create our PositionedSceneElement objects from the original three pictures. We connect the elements in order, then draw them all onto a background (Figure 7.3).

```
> 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();
> bg.write("D:/cs1316/first-house-scene.jpg");
```



Figure 7.3: Our first scene

This successfully draws a scene, but is it easy to recompose into new scenes? Let's say that we decide that we actually want the dog between trees two and three, instead of tree three and the house. To change the list, we need tree2 to point at the doggy element, doggy to point at tree3, and tree3 to point at the house (what the doggy used to point at). Then redraw the scene on a new background (Figure 7.4).

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

```
> bg.show();
> bg.write("D:/cs1316/second-house-scene.jpg");
```



Figure 7.4: Our second scene

Generalizing moving the element

Let's consider what happened in this line:

```
> tree3.setNext(house); tree2.setNext(doggy); doggy.setNext(tree3);
```

The first statement, `tree3.setNext(house);`, gets the doggy out of the list. `tree3` used to point to (`setNext`) `doggy`. The next two statements put the doggy after `tree2`. The second statement, `tree2.setNext(doggy);`, puts the doggy after `tree2`. The last statement, `doggy.setNext(tree3);`, makes the doggy point at what `tree2 used to point at`. All together, the three statements in that line:

- Remove the item `doggy` from the list.
- Insert the item `doggy` after `tree2`.

We can write methods to allow us to do this removing and insertion more generally.

*Program
Example #44*

Example Java Code: **Methods to remove and insert elements in a list**

```
2   /**
 * Method to remove node from list , fixing links appropriately.
 * @param node element to remove from list.
 */
```

```

4  public void remove(PositionedSceneElement node){
5      if (node==this)
6      {
7          System.out.println("I can't remove the first node from the list.");
8          return;
9      };
10
11     PositionedSceneElement current = this;
12     // While there are more nodes to consider
13     while (current.getNext() != null)
14     {
15         if (current.getNext() == node){
16             // Simply make node's next be this next
17             current.setNext(node.getNext());
18             // Make this node point to nothing
19             node.setNext(null);
20             return;
21         }
22         current = current.getNext();
23     }
24 }

26 /**
27 * Insert the input node after this node.
28 * @param node element to insert after this.
29 */
30 public void insertAfter(PositionedSceneElement node){
31     // Save what "this" currently points at
32     PositionedSceneElement oldNext = this.getNext();
33     this.setNext(node);
34     node.setNext(oldNext);
35 }
```

The first method allows us to remove an element from a list, like this:

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

The result is that doggy is removed entirely (Figure 7.5).

Now we can re-insert the doggy wherever we want, say, after tree1 (Figure 7.6):

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



Figure 7.5: Removing the doggy from the scene



Figure 7.6: Inserting the doggy into the scene

7.3 Listing the Pictures, Layering

In the example from last section, we used the *order* of the elements in the linked list to determine *position*. We can decide what our representations encode. Let's say that we didn't want to just have our elements be in a linear sequence—we wanted them to each know their positions anywhere on the screen. What, then, would order in the linked list encode? As we'll see, it will encode *layering*.

* * *

am
ple #45

Example Java Code: LayeredSceneElements

```
1  public class LayeredSceneElement {  
2      /**  
3       * the picture that this element holds  
4       */  
5      private Picture myPic;  
6  
7      /**  
8       * the next element in the list  
9       */  
10     private LayeredSceneElement next;  
11  
12     /**  
13      * The coordinates for this element  
14      */  
15      private int x, y;  
16  
17      /**  
18       * Make a new element with a picture as input, and  
19       * next as null, to be drawn at given x,y  
20       * @param heldPic Picture for element to hold  
21       * @param xpos x position desired for element  
22       * @param ypos y position desired for element  
23       */  
24      public LayeredSceneElement(Picture heldPic, int xpos, int ypos){  
25          myPic = heldPic;  
26          next = null;  
27          x = xpos;  
28          y = ypos;  
29      }  
30  
31      /**  
32       * Methods to set and get next elements  
33       * @param nextOne next element in list  
34       */  
35      public void setNext(LayeredSceneElement nextOne){  
36          this.next = nextOne;  
37      }  
38  
39      public LayeredSceneElement getNext(){  
40          return this.next;  
41      }  
42  
43      /**  
44       * Returns the picture in the node.  
45       * @return the picture in the node
```

```

        */
48  public Picture getPicture(){
    return this.myPic;
}
50 }

52 /**
 * Method to draw from this node on in the list , using bluescreen.
54 * Each new element has it's lower-left corner at the lower-right
55 * of the previous node. Starts drawing from left-bottom
56 * @param bg Picture to draw drawing on
57 */
58 public void drawFromMeOn(Picture bg) {
    LayeredSceneElement current;
59
60     current = this;
61     while (current != null)
62     {
63         current.drawMeOn(bg);
64         current = current.getNext();
65     }
66 }
67

68 /**
70 * Method to draw from this picture , using bluescreen .
71 * @param bg Picture to draw drawing on
72 */
73

74 private void drawMeOn(Picture bg) {
    this.getPicture().bluescreen(bg,x,y);
}
75

76 /**
77 * Method to remove node from list , fixing links appropriately.
78 * @param node element to remove from list .
79 */
80 public void remove(LayeredSceneElement node){
81     if (node==this)
82     {
83         System.out.println("I can't remove the first node from the list.");
84         return;
85     };
86

87     LayeredSceneElement current = this;
88     // While there are more nodes to consider
89     while (current.getNext() != null)
90     {
91         if (current.getNext() == node){
92             // Simply make node's next be this next
93             current.setNext(node.getNext());
94             // Make this node point to nothing
95             node.setNext(null);
96

```

```

98         return;
99     }
100    current = current.getNext();
101 }
102
103 /**
104 * Insert the input node after this node.
105 * @param node element to insert after this.
106 */
107 public void insertAfter(LayeredSceneElement node){
108     // Save what "this" currently points at
109     LayeredSceneElement oldNext = this.getNext();
110     this.setNext(node);
111     node.setNext(oldNext);
112 }
```

Our use of LayeredSceneElement is much the same as the PositionedSceneElement, except that when we create a new element, we also specify its position on the screen.

```

> 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();
> bg.write("D:/cs1316/first-layered-scene.jpg");
```

The result (Figure 7.7) shows the house in front of a tree and the dog. In the upper left, we can see one tree overlapping the other.

How it works: Let's talk about how one piece of this class works, the removal of a node.

```

/** Method to remove node from list , fixing links appropriately.
 * @param node element to remove from list .
 */
public void remove(LayeredSceneElement node){
    if (node==this)
```



Figure 7.7: First rendering of the layered scene

```

{
    System.out.println("I can't remove the first node from the list.");
    return;
};

LayeredSceneElement current = this;
// While there are more nodes to consider
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();
}
}
  
```

You might be wondering what that @param is about in the comments. This is a note to JavaDoc that the method remove takes a parameter. What comes after @param is the parameter name, and then a comment explaining the parameter. This comment will appear in the JavaDoc to explain what the method does and how to use it.



Figure 7.8: A doubly-linked list

The first thing we do in `remove` is to check to see if the node to be removed is the same node that we asked to do the remove, `this`. Typically, you call `remove` on the first node in the list. Currently, our implementation of lists doesn't allow us to remove the first node in the list. If we execute `hisOnethisOne.remove()`, how could we change `thisOne` to point at anything else? We'll see how to fix that later.

We then traverse the list, checking for `current.getNext() != null!`. That's a little unusual—typically, we're checking for `current == null!` when we traverse a list. Why the difference? Because we want to find the node *before* the one we're removing. We need to connect the nodes *around* the node that that's already there. Once we find the node to remove, we want to make the one *before* it point to whatever the removed node *currently* points to. That essentially routes the linked list around the node to be removed—and poof! it's gone. If we stopped when we found the node we were looking for, as opposed to `current.getNext()` being the node we are looking for, we will have gone too far. This is a key idea in linked lists: *there is no link from a node back to the node that is pointing to it*. If you have a pointer to a node, you don't know who points to that node.

That does not have to be true. One can have linked lists where a node points both to its next *and* to its previous (Figure 7.8). We call those doubly-linked lists. They are powerful for finding and replacing nodes, since you can traverse them forwards and backwards. They are more complicated, though—inserting and deleting involves patching up both next links and previous links. They also waste more space than a *singly-linked list*. With a singly-linked list, each piece of data in a node also has a next link associated with it. Compared to an array, a singly-linked list requires an extra reference for every data element. A doubly-linked list gives up *two* references for each data element.

Reordering elements in a list

Now, let's reorder the elements in the list, without changing the elements—not even their locations. We'll reverse the list so that we start with the house, not the first tree. (Notice that we set the `tree1` element to point to `null`—if we didn't do that, we'd get an infinite loop with `tree1` pointing to itself.)

The resultant figure (Figure 7.9) has completely different layering. The

trees in the upper left have swapped, and the tree and dog are now in front of the house.

```
> house.setNext(doggy); doggy.setNext(tree3); tree3.setNext(tree2); tree2.setNext(tr);
> tree1.setNext(null);
> bg = new Picture(400,400);
> house.drawFromMeOn(bg);
> bg.show();
> bg.write("D:/cs1316/second-layered-scene.jpg");
```



Figure 7.9: Second rendering of the layered scene

Have you ever used a drawing program like *Visio* or even *PowerPoint* where you brought an object forward, or sent it to back? What you were doing is, quite literally, exactly what we’re doing when we’re changing the order of elements in the list of `PositionedSceneElements`. In tools such as Visio or PowerPoint, each drawn object is an element in a list. To draw the screen, the program literally walks the list (*traverses* the list) and draws each object. We call the re-creation of the scene through traversing a data structure a *rendering* of the scene. If the list gets reordered (with bringing an object forward or sending it to the back), then the layering changes. “Bringing an object forward” is about moving an element one position further *back* in the list—the things at the end get drawn *last* and thus are on *top*.

One other observation: Did you notice how similar both of these elements implementations are?

7.4 Reversing a List

In the last example, we reversed the list “by hand” in a sense. We took each and every node and reset what it pointed to. What if we had a *lot* of elements, though? What if our scene had dozens of elements in it? Reversing the list would take a lot of commands. Could we write down the *process* of reversing the list, so that we can encode it?

There are actually several different ways of reversing a list. Let’s do it in two different ways here. The first way we’ll do it is by repeatedly getting the last element of the original list, removing it from the list, then adding it to the new reversed list. That will work, but slowly. To find the last element of the list means traversing the whole list. To add an element to the end of the list means walking to the end of the new list and setting the last element there to the new element.

Here’s a method that implements that process.

Example Java Code: **Reverse a list**

*Program
Example #46*

```

1  /**
2   * Reverse the list starting at this,
3   * and return the last element of the list.
4   * The last element becomes the FIRST element
5   * of the list, and THIS points to null.
6   */
7  public LayeredSceneElement reverse() {
8      LayeredSceneElement reversed, temp;
9
10     // Handle the first node outside the loop
11     reversed = this.last();
12     this.remove(reversed);
13
14     while (this.getNext() != null)
15     {
16         temp = this.last();
17         this.remove(temp);
18         reversed.add(temp);
19     }
20
21     // Now put the head of the old list on the end of
22     // the reversed list.
23     reversed.add(this);
24
25     // At this point, reversed
26     // is the head of the list
27     return reversed;
28 }
```

* * *

The core of this program is:

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

So how expensive is this loop?

- We go through this loop once for each element in the list.
- For each element in the list, we find the `last()` (which requires another complete traversal)
- And when we `add()`, which we know requires another `last()` which is another traversal.

The bottom line is, that for each node in the list, we touch every other node. We call that an $O(n^2)$ algorithm—as the data grows larger (n), the number of steps to execute (the running time) increases as a square of the data size (n^2). For a huge list of lots images (maybe wildebeests running down a ridge?), that's very expensive.

How would you do it in real life? Imagine that you have a bunch of cards laid out in a row, and you need to reverse them. How would you do it? One way to do it is to *pile* them up, and then set them back out. A pile (called a *stack* in computer science) has an interesting *property* in that the last thing placed on the pile is the first one to remove from the pile—that's called LIFO, *Last-In-First-Out*. We can use that property to reverse the list. We can define a Stack class to represent the abstract notion of a *pile*, then use it to reverse the list. We will see that in the next chapter.

7.5 Animation

Any movie (including animated scenes) consist of a series of images (typically called *frames*) played in a sequence fast enough to trick our eyes into seeing a continuous image. How would you create an animation using data structures? We can come up with a different definition than “a sequence of frames” now. We can think of an animation as “Modify the structure describing your scene, then *render* it (turn it into an image), and repeat!”

We can do that even with our simple linked lists. We will create an animation of a doggy at the beginning of a grove of trees, then running between the trees. At the end, he turns around and runs back.

* * *

am
ple #47

Example Java Code: Create a simple animation of a dog running

```
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;  
  
    // Set up the whole animation  
    public void setUp(){  
        frames = new FrameSequence("C:/Temp/");  
  
        //FileChooser.setMediaPath("C:/cs1316/mEDIAsources/");  
        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 whole animation  
    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);

48      // Dog moving right
50      bg = new Picture(FileChooser.getMediaPath("jungle.jpg"));
51      tree1.remove(doggy);
52      tree2.insertAfter(doggy);
53      tree1.drawFromMeOn(bg);
54      frames.addFrame(bg);

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

62      bg = new Picture(FileChooser.getMediaPath("jungle.jpg"));
63      tree1.remove(doggy);
64      house.insertAfter(doggy);
65      tree1.drawFromMeOn(bg);
66      frames.addFrame(bg);

68      //Dog moving left
69      bg = new Picture(FileChooser.getMediaPath("jungle.jpg"));
70      tree1.remove(doggy);
71      house.insertAfter(doggyflip);
72      tree1.drawFromMeOn(bg);
73      frames.addFrame(bg);

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

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

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

92  }

94  public void replay(){
95      frames.replay(500); //3 frames per second

```

```
98    }
```

This is definitely not a great animation (Figure 7.10). It looks more like the trees are hopping out of the way of the dog, like the houses hopping away from the Knight Bus in the Harry Potter novels. But this is our first example of how a real computer-generated animation works: By using a data structure, which gets changed and re-rendered in a loop.



Figure 7.10: A few frames from the AnimatedPositionedScene

7.6 Lists with Two Kinds of Elements

Why should we have to choose between having elements positioned for us, left-to-right, or layered for us, back-to-front? That doesn't seem like a reasonable limitation for an animation designer — you may want some images just lined up where you don't care so much about the positioning (e.g., trees in a forest, spectators in the audience) and other images you want to be in very specific places (e.g., your main characters). We can easily imagine having a list where we have both positioned and layered elements. Some nodes we want to position left-to-right, and other nodes we'd want to go at particular places, and nodes earlier in the list would be understood to be rendered behind other nodes.

That turns out to be a new thing for us to do from a Java perspective. Consider that PositionedSceneElement. Its next has the type PositionedSceneElement. How could the next element be anything *but* a PositionedSceneElement? If the next element could be something else, we couldn't use the same how could we draw some things at their (x, y) position and *others* left-to-right?

There's another reason to rewrite our existing scene element classes, besides wanting to intermix different kinds of elements. The current implementation has *so much* duplicated code. Check out last and add in

`PositionedSceneElement` and `LayeredSceneElement`. Basically the same thing—why do we have to duplicate this code? Why can't we tell the computer “Here's last—always use this definition for all linked lists, please!”?

The way to solve both of these problems (being able to intermix kinds of elements and rewriting to reduce redundancy of code) is with a *superclass*. We will create a superclass called `SceneElement`, then we will create a *subclass* for positioned and layered kinds of scene elements. We'll call these `SceneElementLayered` and `SceneElementPositioned`. This structure solves several of the above problems.

- A variable that is declared the type of a superclass can *also* hold objects that have the type of a subclass. So, a variable of type `SceneElement` can also reference a `SceneElementLayered` or `SceneElementPositioned`. That's how we'll define next so that each element can point to any other kind of `SceneElement`.
- Any methods in `SceneElement` are *inherited* by its subclasses. So if we define `last` and `add` in `SceneElement`, any instances of `SceneElementLayered` and `SceneElementPositioned` will automatically inherit those methods. Every instance of `SceneElementLayered` and `SceneElementPositioned` will know how to do `add` and `last` because the superclass `SceneElement` knows how to do it.
- Where the subclasses need to be different, they *can* be made different. So `SceneElementPositioned` will have an `x` and `y` for positioned, but `SceneElementLayered` will not.

If one of the classes will have an (x, y) and the other one will not, each of the subclasses will have to drawWith themselves differently. The `drawWith` method for `SceneElementPositioned` will have to figure out where each element goes left-to-right, and same method for `SceneElementLayered` will draw the element at its desired `x` and `y`.

Since `SceneElement` will not actually know how to draw itself, it is not really a useful object. It is a useful class, though. A class whose main purpose is to hold methods (behavior) and fields (data and structure) that subclasses will inherit is referred to as an *abstract class*. Java even has a keyword **abstract** that is used to identify abstract classes.

An abstract class can also have *abstract methods* (also identified with the **abstract** keyword). These are methods in an abstract class that do not actually have implementations in the abstract class—there is no body for the method. The point of defining an abstract method in an abstract class is to tell Java, “All my subclasses will define this method and provide a real body for this method. I'm just defining it here so that you know that you can expect it—it's okay for a variable declared to be *me* to call this method.”

* * *

am
ple #48

Example Java Code: **Abstract method drawWith in abstract class SceneElement**

```
/*
 * 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
```

Okay, now that we know how to declare the abstract class and the abstract method drawWith, we can consider how to implement the drawWith method for each of the subclasses of SceneElement, SceneElementPositioned and SceneElementLayered. The challenge of implementing drawWith is to keep track of the next *positioned* element when you have to go draw a *layered* element at some particular (x, y) . There are lots of ways of doing that, like holding some nextX and nextY that remembers the next position even when drawing a layered element. Here's the one that we will use here: We will use a turtle to keep track of *where* the element should be drawn. Specifically, each subclass will implement drawWith taking a turtle as input, and they will use that turtle as a *pen* to draw the picture in the node at the right place.

Let's define the SceneElement abstract class first. It knows its picture and knows how to do all the basic list manipulations (Figure 7.11—yet only knows “abstractly” how to draw. A key part of this class definition is that next and all the methods that return an object are declared as class SceneElement—which means that they can actually be any instance in the hierarchy.

<p><i>Abstract Class SceneElement</i></p> <p>It knows its Picture myPic and its next (SceneElement).</p> <p>It knows how to get/set next, to reverse() and insertAfter(), and to drawFromMeOn().</p> <p>It defines drawWith(turtle), but leaves it for its subclasses do complete.</p>

Figure 7.11: The abstract class SceneElement, in terms of what it knows and can do

Program
Example #49

Example Java Code: SceneElement

```

1  /**
2   * An element that knows how to draw itself in a scene with a turtle
3   */
4  public abstract class SceneElement{
5
6      /**
7       * the picture that this element holds
8       */
9      protected Picture myPic;
10
11     /**
12      * the next element in the list — any SceneElement
13      */
14     protected SceneElement next;
15
16     /**
17      * Methods to set and get next elements
18      * @param nextOne next element in list
19      */
20     public void setNext(SceneElement nextOne){
21         this.next = nextOne;
22     }
23
24     public SceneElement getNext(){
25         return this.next;
26     }
27
28     /**
29      * Returns the picture in the node.
30      * @return the picture in the node
31      */
32     public Picture getPicture(){
33         return this.myPic;
34     }
35
36     /**
37      * Method to draw from this node on in the list.
38      * For positioned elements, compute locations.
39      * Each new element has it's lower-left corner at the lower-right
40      * of the previous node. Starts drawing from left-bottom
41      * @param bg Picture to draw drawing on
42      */
43     public void drawFromMeOn(Picture bg) {
44

```

```

        SceneElement current;
46
47    // Start the X at the left
48    // Start the Y along the bottom
49    int currentX=0, currentY = bg.getHeight()-1;
50
51    Turtle pen = new Turtle(bg);
52    pen.setPenDown(true); // Pick the pen up
53
54    current = this;
55    while (current != null)
56    { // Position the turtle for the next positioned element
57        pen.moveTo(currentX, currentY-current.getPicture().getHeight());
58        pen.setHeading(0);
59
60        current.drawWith(pen);
61        currentX = currentX + current.getPicture().getWidth();
62
63        current = current.getNext();
64    }
65
66    /*
67     * Use the given turtle to draw oneself
68     * @param t the Turtle to draw with
69     */
70
71    public abstract void drawWith(Turtle t);
72    // No body in the superclass
73
74    /** Method to remove node from list , fixing links appropriately.
75     * @param node element to remove from list .
76     */
77    public void remove(SceneElement node){
78        if (node==this)
79        {
80            System.out.println("I can't remove the first node from the list.");
81            return;
82        }
83
84        SceneElement current = this;
85        // While there are more nodes to consider
86        while (current.getNext() != null)
87        {
88            if (current.getNext() == node){
89                // Simply make node's next be this next
90                current.setNext(node.getNext());
91                // Make this node point to nothing
92                node.setNext(null);
93                return;
94            }

```

```

        current = current.getNext();
96    }
}
98 /**
100 * Return the count of elements in this list
101 */
102 public int count() {
103     SceneElement current = this;
104     int count = 0;
105     while (current != null){
106         count = count + 1;
107         current = current.getNext();}
108     return count;
109 }
110 /**
111 * Insert the input node after this node.
112 * @param node element to insert after this.
113 */
114 public void insertAfter(SceneElement node){
115     // Save what "this" currently points at
116     SceneElement oldNext = this.getNext();
117     this.setNext(node);
118     node.setNext(oldNext);
119 }
120 /**
121 * Return the last element in the list
122 */
123 public SceneElement last() {
124     SceneElement current;
125
126     current = this;
127     while (current.getNext() != null)
128     {
129         current = current.getNext();
130     };
131     return current;
132 }
133 /**
134 * Reverse the list starting at this,
135 * and return the last element of the list.
136 * The last element becomes the FIRST element
137 * of the list, and THIS goes to null.
138 */
139 public SceneElement reverse() {
140     SceneElement reversed, temp;
141
142     // Handle the first node outside the loop
143 }
```

```

146     reversed = this.last();
147     this.remove(reversed);

148     while (this.getNext() != null)
149     {
150         temp = this.last();
151         this.remove(temp);
152         reversed.add(temp);
153     };
154     // At this point, reversed
155     // is the head of the list
156     return reversed;
157 }

158 /**
159 * Add the input node after the last node in this list.
160 * @param node element to insert after this.
161 */
162 public void add(SceneElement node){
163     this.last().insertAfter(node);
164 }
165
166 }
```

The two subclasses, `SceneElementPositioned` and `SceneElementLayered`, are really quite short. They only specify what's *different* from the superclass. The relationship between a superclass and a subclass is often called, by object-oriented designers, a *generalization-specialization relationship* (sometimes shortened to *gen-spec*). The `SceneElement` is the general form of a scene element, that describes how scene elements *generally* work. The two subclasses just specify how they are *special*, different from the general case (Figure 7.12).

Example Java Code: **SceneElementPositioned**

*Program
Example #50*

```

public class SceneElementPositioned extends SceneElement {
2
    /**
4     * Make a new element with a picture as input, and
5     * next as null.
6     * @param heldPic Picture for element to hold
7     */
8     public SceneElementPositioned(Picture heldPic){
9         myPic = heldPic;
10        next = null;
```

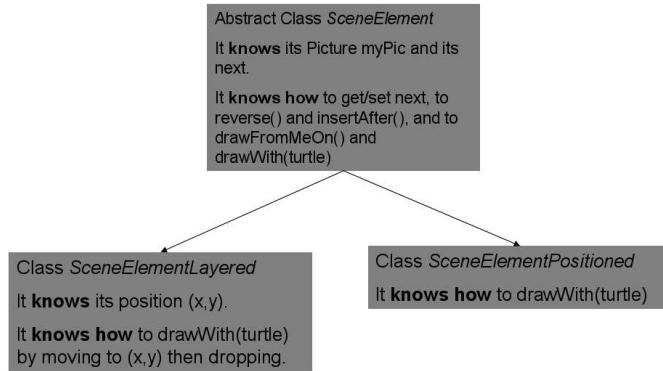


Figure 7.12: The abstract class SceneElement and its two subclasses

```

12 }
13 /**
14 * Method to draw from this picture.
15 * @param pen Turtle to use for drawing
16 */
17 public void drawWith(Turtle pen) {
18     pen.drop(this.getPicture());
19 }
20 }
```

*Program
Example #51*

Example Java Code: **SceneElementLayered**

```

1  public class SceneElementLayered extends SceneElement {
2
3      /**
4      * The coordinates for this element
5      */
6      private int x, y;
7
8      /**
9      * Make a new element with a picture as input, and
10     * next as null, to be drawn at given x,y
11     * @param heldPic Picture for element to hold
12     * @param xpos x position desired for element
13     * @param ypos y position desired for element
14     */
15 }
```

```

15  public SceneElementLayered(Picture heldPic, int xpos, int ypos){
16      myPic = heldPic;
17      next = null;
18      x = xpos;
19      y = ypos;
20  }
21
22  /**
23   * Method to draw from this picture.
24   * @param pen Turtle to draw with
25   */
26  public void drawWith(Turtle pen) {
27      // We just ignore the pen's position
28      pen.moveTo(x,y);
29      pen.drop(this.getPicture());
30  }
31 }
```

This structure will only make sense when we try it out. Here's a simple example of a picture drawn with both kinds of scene elements in a single linked list (Figure 7.13). Notice that the dog is *under* the flower, both of which are out of the linear sequence across the bottom.

Example Java Code: **MultiElementScene**

*Program
Example #52*

```

1  public class MultiElementScene {
2
3      public static void main(String[] args){
4          // Be sure to set the media path first.
5          //FileChooser.setMediaPath("D:/cs1316/mEDIAsources /");
6
7          // We'll use this for filling the nodes
8          Picture p = null;
9
10         p = new Picture(FileChooser.getMediaPath("swan.jpg"));
11         SceneElement node1 = new SceneElementPositioned(p.scale(0.25));
12
13         p = new Picture(FileChooser.getMediaPath("horse.jpg"));
14         SceneElement node2 = new SceneElementPositioned(p.scale(0.25));
15
16         p = new Picture(FileChooser.getMediaPath("dog.jpg"));
17         SceneElement node3 = new SceneElementLayered(p.scale(0.25),50,50);
18
19         p = new Picture(FileChooser.getMediaPath("flower1.jpg"));
20         SceneElement node4 = new SceneElementLayered(p.scale(0.5),10,30);
```



Figure 7.13: A scene rendered from a linked list with different kinds of scene elements

```

22     p = new Picture(FileChooser.getMediaPath("graves.jpg"));
23     SceneElement node5 = new SceneElementPositioned(p.scale(0.25));
24
25     node1.setNext(node2); node2.setNext(node3);
26     node3.setNext(node4); node4.setNext(node5);
27
28     // Now, let's see it!
29     Picture bg = new Picture(600,600);
30     node1.drawFromMeOn(bg);
31     bg.show();
32   }
33 }
```

It is easier to see how a single turtle is being used to draw all of these elements if we change how we traverse the list with `drawFromMeOn` in the `SceneElement` class. We will simply traverse the list with the turtle's pen *down*.

* * *

Example Java Code: Modified drawFromMeOn in SceneElement

*Program
Example #53*

```
/*
 * Method to draw from this node on in the list.
 * For positioned elements, compute locations.
 * 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) {
    SceneElement current;

    // Start the X at the left
    // Start the Y along the bottom
    int currentX=0, currentY = bg.getHeight()-1;

    Turtle pen = new Turtle(bg);
    pen.setPenDown(true); // NOW, LEAVE THE PEN DOWN

    current = this;
    while (current != null)
    { // Position the turtle for the next positioned element
        pen.moveTo(currentX, currentY-current.getPicture().getHeight());
        pen.setHeading(0);

        current.drawWith(pen);
        currentX = currentX + current.getPicture().getWidth();

        current = current.getNext();
    }
}
```

The result is a picture showing the trace of where the turtle moved throughout the traversal (Figure 7.14). The pen gets created in the center, then travels down to the left corner to draw the swan, then moves right (the width of the swan) to draw the horse. The turtle pen moves into position for drawing the flower, but instead moves to the dog's saved position and draws there. The turtle moves back into place to draw the next positioned scene element, but again moves up to draw the flower (on top of a corner of the dog picture). Finally, it moves to the next position and draws

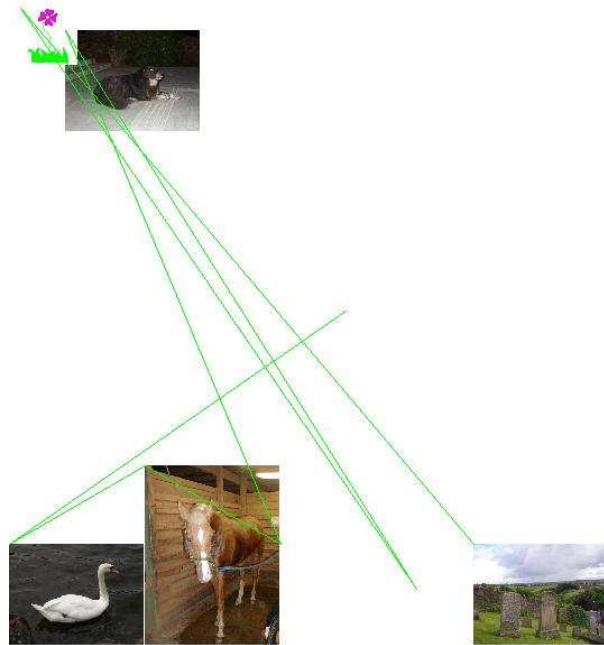


Figure 7.14: Same multi-element scene with pen traced

the graves there.

Structuring Our Multi-Element Lists

One can imagine using our new kind of multi-element lists to create complicated scenes.

- Imagine creating large sequences of objects next to one another, like trees in a forest, or orcs about to enter into battle in *The Lord of the Rings*. We could create these by inserting as many nodes as we need with `SceneElementPositioned`. We wouldn't want to position each of these separately—we simply want to add them all into the list and have them drawn at the right place.
- Now, you might want to position some specific characters at specific positions in the scene, such as an elf and a dwarf to battle the orcs, or a plastic spaceman getting ready to enter the scene.

A linear list is not the best way to present such a scene. You would want to keep track of this band of orcs here, and that band of orcs there, with a forest behind, and a few brave heroes getting ready to keep the horde at bay. A linear list doesn't give you an easy way of keeping track of the various pieces. A linear list is just a long list of nodes.

It would be easier to structure and manipulate the scene if one part of the linked list represented the Fifth Army of Orcs, and another part represented the forest, and the heroes were elsewhere—and if all those parts were clearly labeled and manipulable. To keep track of just that kind of structure, we are going to introduce a *scene graph*, a kind of *tree*. A tree structures a linked list into a hierarchy or clusters. A scene graph is an important data structure in defining scenes in modern computer animations. That's where we're heading next.

Ex. 2 — Create a method `copyList` that copies the list at `this` and returns a new list with the same elements in it.

Ex. 3 — Create a method `copyList(int n,m)` that starts at `this` as the 0th element in the list, goes to the *n*th element, and returns a new list from the *n*th to the *m*th element.

Ex. 4 — Using any of the linked lists of pictures that we created (where ordering represented linearity, or layering, or using turtles to walk the list), implement three additional methods (where `firstpicturnodeinlist` is a node, not actually a picture):

- `firstnodeinlist .findAndReplaceRepeat(oldelement,newelement,n):` Find `oldelement` (it's a node, not a picture, so you can look for an exact match), remove `oldelement` from the list, and then insert at `oldelement`'s place *n* copies of the node `newelement`. Imagine this as implementing the special effect where the witch disappears and gets replaced with three smaller copies of the witch at the same place.
- `firstnodeinlist .replaceWithModification(findelement,int type):` Find the node `findelement` (a node, not a picture, so you can do an exact match), then replace it with a node containing a modified picture of the picture in `findelement`. The type indicates the kind of modification. If 1, negate the picture. If 2, mirror horizontal. If 3, mirror vertical. If 4, sunset. If anything else, insert grayscale. This is about changing the image in some predefined way for any node in the list.
- `firstnodeinlist .replaceWithModifications(findelement,double[] types):` Find the node `findelement` (a node, not a picture, so you can do an exact match), then replace it with nodes containing a modified picture of the picture in `findelement`. Insert as many nodes as there are entries in the `types` array, e.g., if there are 3 values in the array, insert 3 copies. The value in the `types` array indicates the kind of image modification for that element. If 1, negate the picture. If 2, mirror horizontal. If 3, mirror vertical. If 4, sunset. If anything else, insert grayscale. So, if the array was {1,2,5}, you would insert three copies of the picture in `findelement` the first negated, the second mirrored horizontal, and the third grayscaled. This is an example of taking a base wildebeest and replacing it with several copies having different colors.

You can be sure that `oldelement` or `findelement` will never be equal to the `firstnodeinlist`.

You must also provide a class named PictureTest that has a main() method which utilizes all three of your new methods. When the grader executes the main(), it should (a) show a background with three or more pictures in it, (b) then show a new picture after using findAndReplaceRepeat, and (c) a third picture after using replaceWithModification, then (d) a final fourth picture after using replaceWithModifications.

Ex. 5 — Create a class with a main method that sets up a scene with LayeredSceneElement, then change the layering of just a *single* element using remove and insertAfter—much as we did to make the doggy run in our first animation using PositionedSceneElement.

Ex. 6 — Take any linked list implementation that we have created thus far, and recreate it as a doubly-linked list. Make sure that insertAfter and remove work, and implement insertBefore as well.

8 Abstract Data Types: Separating the Meaning from the Implementation

Chapter Learning Objectives

One of the most powerful ideas in computer science is that the *definition* of a data structure can be entirely separated from the *implementation* of the data structure. This idea is powerful because it allows us to design and implement *other* code that uses the given data structure (1) without knowing the implementation of the data structure and (2) even if the implementation of the data structure changes.

The computer science goals for this chapter are:

- To explain the separation of definition and implementation.
- To explain the definition of the queue and offer two implementations of those.
- To explain the definition of the stack and offer two implementations of those.

The media learning goals for this chapter are:

- To have a faster way of reversing the elements in a list by using a stack.

8.1 Introducing the Stack

A *stack* is a data structure that corresponds to how a stack (literally!) of plates work. Imagine a stack of fine plates (Figure 8.1). You only put new plates on a stack from the top. You *can* insert plates in the middle, but not easily, and you risk scraping the plates. You only remove plates from a stack from the top. Removing from the middle or (worse yet) the bottom is dangerous and risks damaging the whole stack.

Those basic notions of how a physical stack work correspond to how the stack data structure works. Think of a stack as a list of elements—just a sequence of items. (Are the items in a linked list? An array? We



Figure 8.1: A pile of plates—only put on the top, never remove from the bottom

are *explicitly* not saying at this point.) The first item pushed on the stack stays at the bottom of the stack. Later items are put at the top of the stack (Figure 8.2).

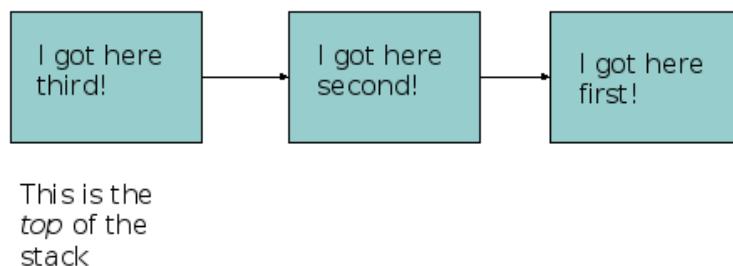


Figure 8.2: Later items are at the head (top) of stack

New items are added at the head of the stack (Figure 8.3). When a new item is removed from the stack, it is always the last, newest item in the list which is removed first (Figure 8.4). For that reason, a stack is sometimes also called a *LIFO list*—Last In, First Out. The last item inserted into the stack is the first one back out.

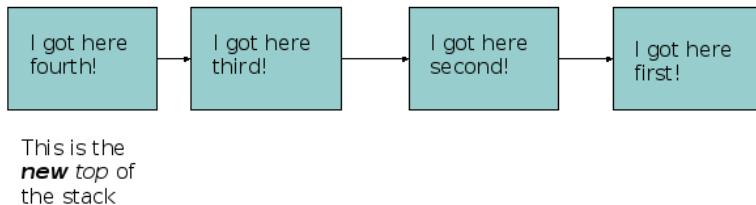


Figure 8.3: New items are inserted at the top (head) of the stack

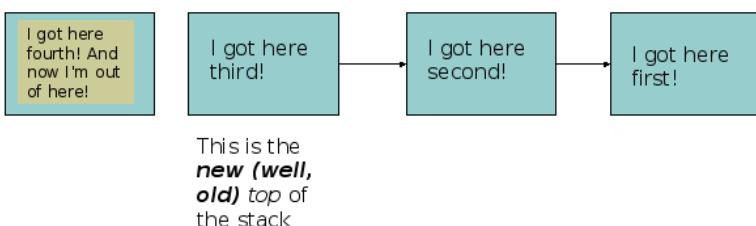


Figure 8.4: Items are removed from the top (head) of a stack

Defining an Abstract Data Type

A stack is an example of an *Abstract Data Type*. An abstract data type defines the operations, structure, and behavior of a data structure *without* specifying how those operations, structure, and behavior are actually implemented. By separating the concerns of *what* is supposed to happen from *how* it is supposed to happen, the task of programming with that data type is made easier.

There are formal ways of defining an abstract data type (ADT). These formal methods use symbols and logical equations to define behavior and structure without getting into the implementation. Those aren't our focus here. Instead, we will focus on a general notion of how the ADT works.

What should someone be able to do with a stack data structure? Here is a list of basic operations for a stack with a brief description of what each does. If we got these right, these should mesh with the intuitive understanding of stacks described previously.

- `push(anObject)`: Tack a new object onto the top of the stack
- `pop()`: Pull the top (head) object off the stack.
- `peek()`: Get the top of the stack, but don't remove it from the stack.
- `size()`: Return the size of the stack
- `empty()`: Returns true if the size of the stack is zero—nothing is left in the stack.

Computer Science Idea: Separation of Concerns

By separating the definition of the data structure from the implementation, we can use the new data structure in programs without regard for implementation. An important implication is that the implementation can be changed (improved, upgraded, whatever) and the programs that use the data structure need not change *at all!*

For example, here is a program that uses a stack, PictureStack, that contains SceneElement nodes.

Program
Example #54

Example Java Code: Testing program for a PictureStack

```
public class PictureStackTest {  
    public static void main(String [] args){  
        PictureStack stack = new PictureStack();  
        SceneElementPositioned element = null;  
  
        // Push 3 pictures on the stack: Swan, temple, horse  
        element=new SceneElementPositioned(  
            new Picture(FileChooser.getMediaPath("swan.jpg")));  
        stack.push(element);  
        element=new SceneElementPositioned(  
            new Picture(FileChooser.getMediaPath("temple.jpg")));  
        stack.push(element);  
        element=new SceneElementPositioned(  
            new Picture(FileChooser.getMediaPath("horse.jpg")));  
        stack.push(element);  
  
        //Pop one — should be the horse  
        stack.pop().getPicture().show()  
  
        //Push another — the turtle  
        element=new SceneElementPositioned(  
            new Picture(FileChooser.getMediaPath("turtle.jpg")));  
        stack.push(element);  
  
        //Pop another — should be the turtle  
        stack.pop().getPicture().show();  
  
        //Pop another — should be the temple  
        stack.pop().getPicture().show();  
    }  
}
```

```
}
```

Given that we're pushing and popping instances of SceneElementPositioned, you have probably guessed that the internal implementation is a linked list. It need not be. The PictureStack could be implemented as an array of SceneElement. The important point is this: *It doesn't matter!* No matter how PictureStack is implemented, the above example *should just work*, as long as it meets the definition of a stack.

The idea of separation of concerns is actually built into Java as a construct. In Java, one can define an Interface. An **interface** is akin to an **abstract class** in that it defines a set of methods and no one can instantiate an **interface**. An **interface** is just the definition of the operations of an abstract data type, like those we just defined for a stack. A particular class can declare that it **implements** a given **interface**. That implementation is then a promise, a contract with programs that use the particular class—that the methods in the **interface** are implemented and implemented properly so that the operation of the data type works.

Implementation of a Stack

At this point, our discussion of abstract data types is rather abstract—it's all definition and no implementation. Let's implement PictureStack to see how it might work. First, let's define the basic structure of the class.

Example Java Code: **PictureStack**

*Program
Example #55*

```
public class PictureStack {  
  
    /** Where we store the elements */  
    private SceneElement elements=null;  
  
    /// Constructor ////  
    public PictureStack() {  
        // If our SceneElement had an empty first  
        // node, we'd create it here.  
    }  
  
    // Methods go here  
}
```

We can already see that we are going to have some kind of linked list of SceneElement instances. Let's define those basic methods.

* * *

Program
Example #56

Example Java Code: **PictureStack—push, peek, and pop**

```
//// Methods ///
public void push(SceneElement element){
    if (elements == null)
        {// First new element on the stack
           // becomes the one that we point at
           elements = element;}
    else
        { // New elements go at the front
           //The rest get added to the front
           //by making the new one point to the
           //rest of the list, then change
           //the pointer to the new front.
        element.setNext(elements);
        elements = element;
    }
}

//Top element is the first one
public SceneElement peek(){
    return elements;
}

// Get the first one to return,
// then point to the second one
public SceneElement pop(){
    SceneElement toReturn = this.peek();
    elements=elements.getNext();
    return toReturn;
}
```

We are going to use the variable elements to point to the head of the list. To push a new element on the list is to make the new item point to the rest of the list, and to make elements point to the new element. To peek is simply to return whatever elements points at. Pop is a matter of getting the peek, making elements point to the next element (elements.getNext()), and returning what was previously peeked.

The other two methods needed to match the definition of a stack follow.

* * *

am
ple #57

Example Java Code: PictureStack—size and empty

```
public int size(){return elements.count();}

/** Empty? */
public boolean empty() {return this.size() == 0;}
```

Given this definition, our PictureStackTest class should work correctly now. However, there were more ways to implement the stack. It's worthwhile understanding more than one way to define the stack.

Multiple Implementations of a Stack

Just to make it simpler for us to trace, let's consider the implementation of a class Stack that stores simple strings. If we implement Stack correctly, the below should work:

```
Welcome to DrJava.
> Stack stack = new Stack()
> stack.push("This")
> stack.push("is")
> stack.push("a")
> stack.push("test")
> stack.size()
4
> stack.peek()
"test"
> stack.pop()
"test"
> stack.pop()
"a"
> stack.pop()
"is"
> stack.pop()
"This"
> stack.pop()
java.util.NoSuchElementException:
```

Should that last pop, when there was nothing left in the stack, have generated that error? If not, what error should it have generated? Java does give us the opportunity to define our own exceptions and **throw** that exception (e.g., cause it to stop execution and appear in the Interactions Pane or console). We will explore exceptions in a later chapter.

The two most common implementations of stacks use linked lists and arrays. Let's look at each of those. First, we will use a stack implementation with a linked list, where we will use Java's provided implementation

of a linked list. Yes, Java has a linked list! How is it implemented? Is it a singly or doubly linked list? We don't know and we don't care—that's separation of concerns. A bigger question is why we haven't been using Java's linked list all along. We'll explain that when we get to simulations.

Program
Example #58

Example Java Code: Stack implementation with a linked list

```

import java.util.LinkedList; // Need for LinkedList

public class Stack {

    /** Where we store the elements */
    private LinkedList elements;

    /// Constructor ///
    public Stack() {
        elements = new LinkedList();
    }

    /// Methods ///
    public void push(Object element){
        // New elements go at the front
        elements.addFirst(element);
    }

    public Object peek(){
        return elements.getFirst();
    }

    public Object pop(){
        Object toReturn = this.peek();
        elements.removeFirst();
        return toReturn;
    }

    public int size(){return elements.size();}

    /** Empty? */
    public boolean empty() {return this.size() == 0;}
}

```

How it works: In this implementation, we have decided that the head of the stack is the first item in the linked list. Java's linked list implementation provides a `getFirst` and a `addFirst` method—the former is the heart of `peek` and the latter is how we push. Since we have `removeFirst` too, that's

how we pop. It is interesting that the definition of empty is identical in both implementations we have seen of stacks.

Java's linked list implementation can actually store any object at all. Its implementation works around the class Object. Every class (even if it declares no superclass) is a subclass of Object. Thus, an Object variable can hold anything at all.

If we use this implementation of Stack in the example earlier, it will work as expected. Here's a different implementation, using an array. We will use an array of Object instances, so that we can store anything we want in our stack. In this example, the class Stack2 (use it just like Stack) has an index in the array for a top.

Example Java Code: Stack implementation with an array

*Program
Example #59*

```
/*
 * Implementation of a stack as an array
 */
public class Stack2 {

    private static int ARRSIZE = 20;

    /** Where we'll store our elements */
    private Object[] elements;

    /** Index where the top of the stack is */
    private int top;

    /// Constructor /////
    public Stack2() {
        elements = new Object[ARRSIZE];
        top = 0;
    }
    //METHODS GO HERE
}
```

Here, the variable elements is an array of objects. We have a **static** variable that declares that our array will have at most 20 elements in it. To create a Stack2, we create an array of Objects, and set the top at element index zero.

Example Java Code: Stack implementation with an array—methods

*Program
Example #60*

```
//// Methods ///
public void push(Object element){
    // New elements go at the top
    elements[top]=element;
    // then add to the top
    top++;
    if (top==ARRAYSIZE){
        System.out.println("Stack overflow!");
    }
}

public Object peek(){
    if (top==0){
        System.out.println("Stack empty!");
        return null;
    } else{
        return elements[top-1];
    }
}

public Object pop(){
    Object toReturn = this.peek();
    top--;
    return toReturn;
}

/** Size is simply the top index */
public int size(){return top;}

/** Empty? */
public boolean empty() {return this.size() == 0;}
```

How it works: In this implementation, the variable `top` is an index to the next *empty* element in the array. The size is then the same as the index. To insert a new element, we set the value at `elements[top]` and increment `top`. The top value is at `elements[top-1]`. To remove an element, we decrement `top`.

We can now test `Stack2` and find that it works remarkably like our `Stack`. When we first create an instance of `Stack2`, it looks like Figure 8.5. After we push `'Matt'` on the stack, it looks like Figure 8.6—the variable `top` always points at the *next* element on the stack. We then push `'Katie'` and `'Jenny'`, then pop `'Jenny'`, resulting in Figure 8.7—`Jenny` is still in the array, but since `top` points at that element, `Jenny` will be over-written with the next `push()`.

```
Welcome to DrJava.
> Stack2 stack = new Stack2();
> stack.push("Matt")
> stack.push("Katie")
```

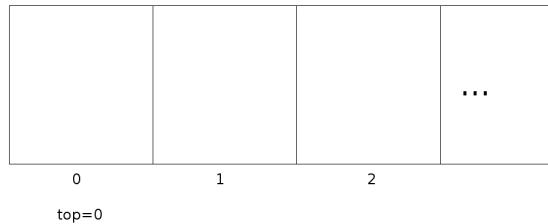


Figure 8.5: An empty stack as an array

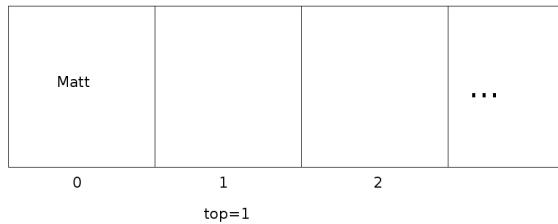


Figure 8.6: After pushing Matt onto the stack-as-an-array

```
> stack.push("Jenny")
> stack.size()
3
> stack.peek()
"Jenny"
> stack.pop()
"Jenny"
> stack.pop()
"Katie"
> stack.pop()
"Matt"
> stack.pop()
Stack empty!
null
```

There are problems with Stack2 as an implementation of a stack. For example, what happens if you need more than 20 elements in your stack? And actually, it's 19 elements—`top` always points at an unused cell. However, within the limitations of Stack2 (e.g., as long as you have fewer than 20 elements in the stack), it is a perfectly valid implementation of a stack. Both Stack and Stack2 are implementations of the same stack ADT.

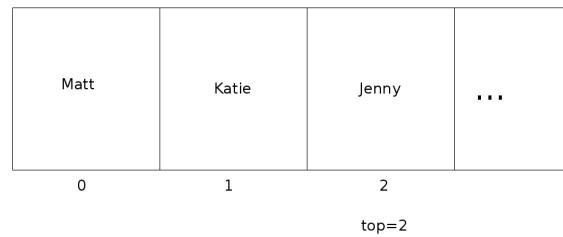


Figure 8.7: After pushing Katie and Jenny, then popping Jenny

Uses of a Stack

We haven't seen enough algorithms yet to be able to describe many uses of stacks, though they are quite useful. For example, the process of converting an equation like $4 * \sin(x/2)$ into operations that a computer can execute uses a stack. There is one use of a stack that we have already seen a need for—reversing a list.

Below is the method that we wrote previously for reversing a list.

Program
Example #61

Example Java Code: Reverse a list-repeated

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

It's awfully inefficient. To get each item in the list, it goes to the last() (which requires touching every element in the list), removes it (which involves touching every node in order to find the one before the one we want to remove), and then adds it to the end of the reversed list (by walking all the elements of the reversed list to get the last). Overall, touching each node of n nodes requires touching every other node at least once, meaning that it's an $O(n^2)$ algorithm.

We can do this much faster with a stack. We simply walk the list pushing everything onto the stack. We pop them off again, and they assemble in reversed order.

Example Java Code: Reverse with a stack

Program
Example #62

```
/**
 * Reverse2: Push all the elements on
 * the stack, then pop all the elements
 * off the stack.
 */
public LayeredSceneElement reverse2() {
    LayeredSceneElement reversed, current, popped;
    Stack stack = new Stack();

    // Push all the elements on the list
    current=this;
    while (current != null)
    {
        stack.push(current);
        current = current.getNext();
    }

    // Make the last element (current top of stack) into new first
    reversed = (LayeredSceneElement) stack.pop();

    // Now, pop them all onto the list
    current = reversed;
    while (stack.size()>0) {
        popped=(LayeredSceneElement) stack.pop();
        current.insertAfter(popped);
        current = popped;
    }
}
```

```
    return reversed;  
}
```

You will notice that we have to cast the popped elements. Our Stack returns instances of Object. To get them to be able to access methods like getNext() and insertAfter(), we have to tell Java that these are actually our linked list nodes. So we cast, like popped=(LayeredSceneElement) stack.pop();.

There is a big difference in time of execution here. Each node is touched exactly twice—once going on the stack, and once coming off. There is no last nor remove here to cause us extra traversals of the list. This is an $O(n)$ algorithm—much faster than the last one.

Does that time really matter? If you really needed to reverse a linked list, does $O(n)$ or $O(n^2)$ matter? Couldn't you just recreate the linked list, even, rather than reverse it? There are situations where the size of the list does matter.

Consider a scene from Pixar's *The Incredibles*, where the monorail enters the bad guy's lair through a waterfall that parts (incredible!) to allow the monorail entry. Each droplet in that scene was modeled as an object that was nearly transparent—it showed up only 1 millionth of a bit of color. Thus, everywhere that looks white (which is nearly everywhere in a waterfall scene) is actually over one million objects stacked up at that spot on the screen.

Now, imagine that those droplets are stored in a linked list, and the director says "Ooh! Oooh! Now, let's show the waterfall from the *inside* as the monorail passes through!" At that point, you have to reverse the linked list. You could recreate the list—but that is a big list to recreate. If your algorithm is $O(n)$, that's a million steps per screen element. That's a lot of processing, but not insurmountable. If your algorithm is $O(n^2)$, that's a million-squared operations per screen element—one trillion operations per pixel. Computers are fast. Computers are not infinitely fast.

8.2 Introduction to Queues

A *queue* is another useful abstract data type. A queue models what the British call "a queue," and what we in the United States call "a line." A queue is a *FIFO list*, *First In First Out list*. The first one in the line is the first one served. When someone new comes into the line, the bouncer at the front of the line yells out, "Get in the back of the line!" New elements enter at the end, not at the front, never in the middle (that's called "cutting in the line").

Unlike a stack, which just has a *head* or *top*, a queue has both a *head* and a *tail* (Figure 8.8). Elements are popped from the *head* (Figure 8.9). When a new element comes in, it gets pushed onto the queue at the *tail* (Figure 8.10).

The basic operations of a queue are pretty similar to those of a stack.

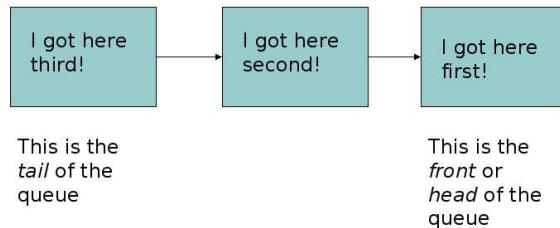


Figure 8.8: A basic queue

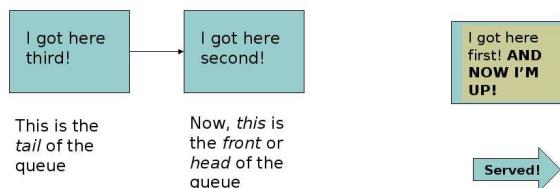


Figure 8.9: Elements are removed from the top or head of the queue

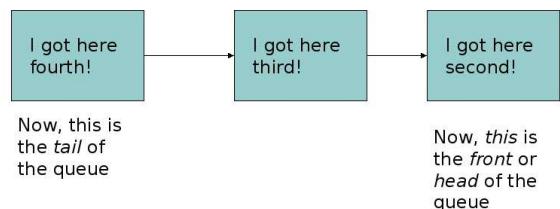


Figure 8.10: New elements are pushed onto the tail of the queue

- `push(anObject)`: Tack a new object onto the tail of the queue
- `pop()`: Pull the top (head) object off the queue.
- `peek()`: Get the head of the queue, but dont remove it from the queue.
- `size()`: Return the size of the queue
- `empty()`: Return true or false, if the size of the queue is zero.

We should be able to use a queue like this:

```
> Queue line = new Queue();
> line.push("Fred");
> line.push("Mary");
> line.push("Jose");
> line.size()
```

```
3
> line.peek()
"Fred"
> line.pop()
"Fred"
> line.peek()
"Mary"
> line.pop()
"Mary"
> line.peek()
"Jose"
> line.pop()
"Jose"
> line.pop()
java.util.NoSuchElementException:
```

Implementing a Queue

We can implement a queue using either a linked list, or an array—just like a stack. A linked list implementation of a queue is again made very easy by Java's `LinkedList` implementation.

*Program
Example #63*

Example Java Code: Queue implemented as a linked list

```
import java.util.*; // LinkedList representation

/**
 * Implements a simple queue
 */
public class Queue {
    /** Where we'll store our elements */
    public LinkedList elements;

    /** Constructor
    public Queue(){
        elements = new LinkedList();
    }

    /** Methods

    /** Push an object onto the Queue */
    public void push(Object element){
        elements.addFirst(element);
    }

    /** Peek at, but don't remove, top of queue */
    public Object peek(){
```

```

return elements.getLast();}

/** Pop an object from the Queue */
public Object pop(){
    Object toReturn = this.peek();
    elements.removeLast();
    return toReturn;
}

/** Return the size of a queue */
public int size() { return elements.size();}

/** Empty? */
public boolean empty() {return this.size() == 0;}
}

```

How it works: In this implementation of a queue, the elements are stored in an instance of Java's LinkedList. The front of the linked list is the tail. The last of the linked list is the head. (That may seem backwards, but any arrangement of head/tail to front/back is arbitrary. That one might find an implementation confusing is yet another good reason for separating the concerns.) To pop the stack then requires removing the last (removeLast), and to push is adding to the first (addFirst).

The critical issue in implementing a queue is that, from the perspective of the use of the queue, the implementation doesn't matter. As long as those same basic operations are available, the implementation of the queue can be swapped out, improved, made faster (or slower, for that matter)—and it just doesn't matter to the program using the queue.

An obvious alternative to the linked list implementation seen above is an implementation based on an array. Implementing a queue using an array to store the elements is much the same as implementing a stack with an array except that, as before, we need both a head and a tail.

Example Java Code: Queue implemented as an array

*Program
Example #64*

```

/*
 * Implements a simple queue
 */
public class Queue2 {

    private static int ARRSIZE = 20;

    /** Where we'll store our elements */
    private Object[] elements;
}

```

```

/* The indices of the head and tail */
private int head;
private int tail;

/// Constructor
/* Both the head and the tail point at the same empty cell */
public Queue2(){
    elements = new Object[ARRAYSIZE];
    head = 0;
    tail = 0;
}

/* Push an object onto the Queue */
public void push(Object element){
    if ((tail + 1) >= ARRAYSIZE) {
        System.out.println("Queue underlying implementation failed");
    }
    else {
        // Store at the tail,
        // then increment to a new open position
        elements[tail] = element;
        tail++;
    }
}

/* Peek at, but don't remove, top of queue */
public Object peek(){
    return elements[head];
}

/* Pop an object from the Queue */
public Object pop(){
    Object toReturn = this.peek();
    if (((head + 1) >= ARRAYSIZE) ||
        (head > tail)) {
        System.out.println("Queue underlying implementation failed.");
        return toReturn;
    }
    else {
        // Increment the head forward, too.
        head++;
        return toReturn;
    }
}

/* Return the size of a queue */
public int size() { return tail - head; }

/* Empty? */
public boolean empty() { return this.size() == 0; }
}

```

* * *

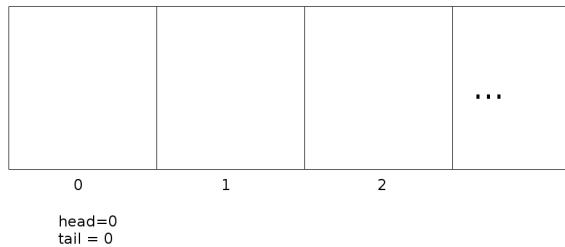


Figure 8.11: When the queue-as-array starts out, head and tail are both zero

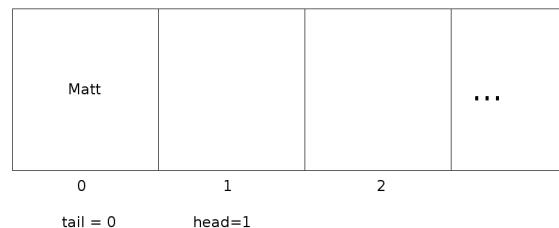


Figure 8.12: Pushing Matt onto the queue moves up the head to the next empty cell

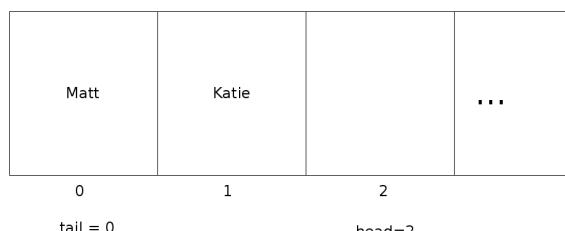


Figure 8.13: Pushing Katie on moves the head further right

How it works: Each of the head and tail variables holds an index number on the underlying array. The size of the queue is the difference between the *head* and the *tail*. When the queue is first created, both the *head* and the *tail* are zero, the first cell in the array (Figure 8.11). When we push something on, the *tail* increments to the next empty cell in the array (which has been implemented as the *head* variable in this implementation, Figure 8.12 and Figure 8.13). Popping from the *head* increments it (again, variable *tail* in this implementation, Figure 8.14).

The problems of implementing queues as arrays are even greater than the ones with implementing stacks as arrays, because now we have both a

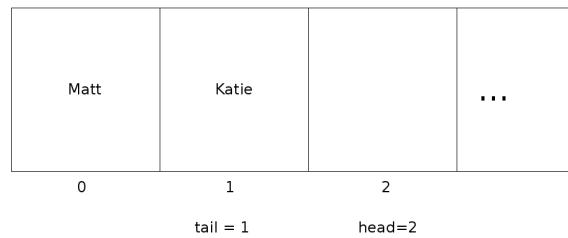


Figure 8.14: Popping Matt moves the tail up to Katie

head and a tail to keep track of. Notice that the head and tail keep moving along the array, so the implementation can't even store 20 elements at once. It can only store 20 elements *ever*—once we get past 20, both head and tail are at the end of the array. There are implementations of queues in arrays that are smarter than this one. For example, there's no reason why we can't wrap around the queue—once we increment past the end, just start re-using cells at the beginning of the array. It's a little trickier to make sure that the head and tail don't overlap one another. Then, the 20 element limitation is just the number of items at once, which isn't unreasonable.

We haven't seen uses of the queue yet. Queues will become very important when we create discrete event simulations. When several agents want the same resource (e.g., when customers line up at the cashier station, when moviegoers line up at the ticket office), they form a line. We will need queues for modeling those lines.

Ex. 7 — Change the implementation of the queue as an array so that the head and tail variables match the head and tail concepts.

Ex. 8 — Change the implementation of the queue as a linked list so that the head is at the front and the tail is at the end.

Ex. 9 — Reimplement the queue in an array with wraparound—when the head or tail get to the end of the array, they wrap around to the front.

Part III

Trees: Hierarchical Structures for Media

9 Trees of Images

Chapter Learning Objectives

Having worked with linked lists of images, we are in a position to think about non-linear lists of images. By moving to trees, we can represent structure and hierarchy.

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.

9.1 Representing scenes with trees

A list can only really represent a single dimension—either a linear placement on the screen, or a linear layering front-to-back. A full scene has multiple dimensions. More importantly, a real scene has an organization to it. There's the village over there, and the forest over here, and the squadron of Orcs emerging from the scary cave mouth to the north. Real scenes *cluster*—there is structure and organization to them.

Where we left our efforts to create a dynamic data structure for representing scenes, we had implemented a special kind of tree where some nodes layered themselves at a particular (x, y) , and other nodes just laid themselves out left-to-right. One problem with this structure is that it's linear—none of the structure or organization we would want to model is available in a linear linked list. A second problem is an issue of *responsibility*. Should a picture know where it is going to be drawn? Shouldn't it just be drawn wherever it's told to be drawn—whether that's in a long linear list like long lines of Orcs in *The Lord of the Rings* movies¹, or whether it's particularly positioned like a specific house, tree, or Hobbit.

¹Do you think someone positioned each of those thousands of Orcs by hand? Or something auto-positioned them?

* * *

Computer Science Idea: Distribution of Responsibility

A key idea in object-oriented programming is distributing responsibility throughout the collection of objects in a model. Each object should only know what it needs to know do its job, and its methods should just be sufficient to implement that job.

We can represent an entire scene with a tree. Computer scientists call the tree that is rendered to generate an entire scene a *scene graph*. Scene graphs are a common data structure representing three-dimensional scenes. An interesting aspect of scene graphs is that they typically embed operations within the branches of the tree which effect rendering of their children. Figure 9.1 is a simple scene graph² with branches that *rotate* in three dimensions their children elements. Figure 9.2 is a more sophisticated scene graph³ based on the *Java 3-D* libraries—you’ll see that it also represents more sophisticated aspects of rendering the scene, like where the lighting is and implementing *translations* that move the drawing of the children to a particular place on the screen.

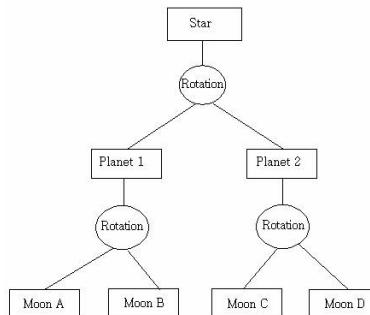


Figure 9.1: A simple scene graph

We are going to use scene graphs to represent a simple two-dimensional movie. A scene graph is a kind of tree. We are going to use our tree to cluster and organize our scene, and we will use operations in the branches of the tree to help define how the scene should be rendered.

²From <http://www.gamedev.net/reference/articles/article2028.asp>

³Used with permission from http://ocw.mit.edu/OcwWeb/Civil-and-Environmental-Engineering/1-124JFall2000/LectureNotes/detail/java_3d_lecture.htm

9.2. OUR FIRST SCENE GRAPH: ATTACK OF THE KILLER WOLVES

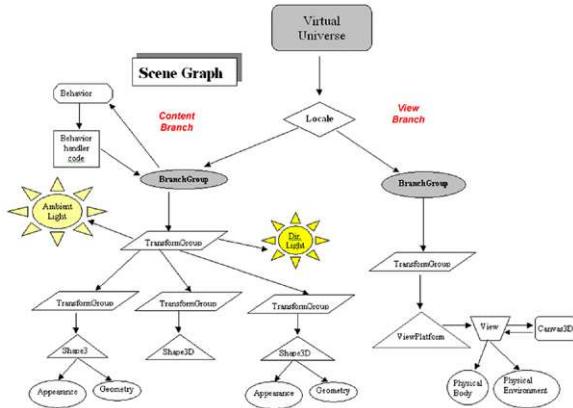


Figure 9.2: A more sophisticated scene graph based on Java 3-D

9.2 Our First Scene Graph: Attack of the Killer Wolves

Here is the story that we will be rendering with our first scene graph. The peaceful village in the forest rests, unsuspecting that a pack of three blood-thirsty doggies, er, fierce wolves are sneaking up on it (Figure 9.3). Closer they come, until the hero bursts onto the scene (Figure 9.4)! The fear-stricken wolves scamper away (Figure 9.5).

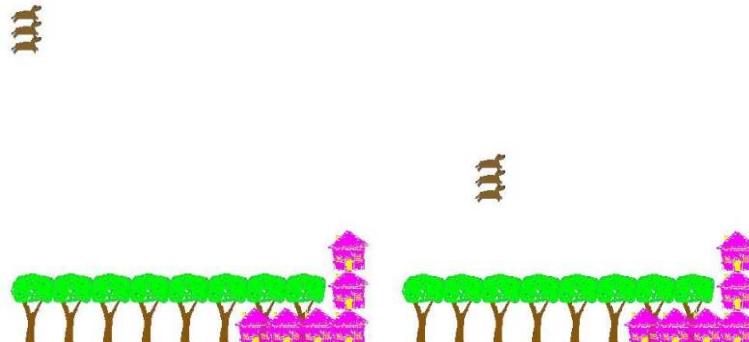


Figure 9.3: The nasty wolves sneak up on the unsuspecting village in the forest

This scene is described as a scene graph (Figure 9.6). There is a root object, of class Branch. Each of the individual pictures are instances of BlueScreenNode which are kinds of PictNode (Picture Nodes) that use chromakey (“blue screen”) for rendering themselves onto the screen. The po-

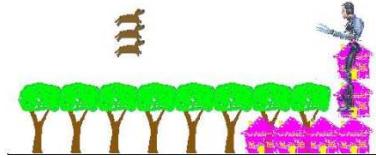


Figure 9.4: Then, our hero appears!

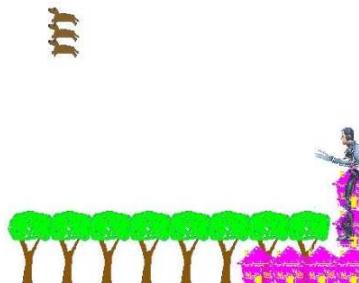


Figure 9.5: And the nasty wolfies scamper away

sition of nodes is done with `MoveBranch` branches. Some of the objects are laid out vertically (like the wolfies) using a `VBranch` instance, and others horizontally (like the forest) using a `HBranch` instance.

This scene graph underlying this scene is quite clearly a tree (Figure 9.7). There is a *root* at the top of the tree that everything is connected to. Each node has at most one *parent node*. There are *branch nodes* that have *children nodes* connected to them. At the ends of each branch, there are *leaf nodes* that have no children. In our scene graph, the leaves are all nodes that draw something. The branches collect the children, and thus, structure the scene. In addition, the branches in the scene graph *do* something.

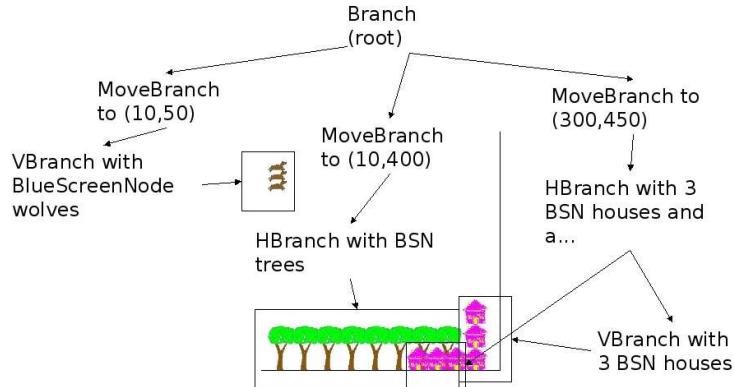


Figure 9.6: Mapping the elements of the scene onto the scene graph

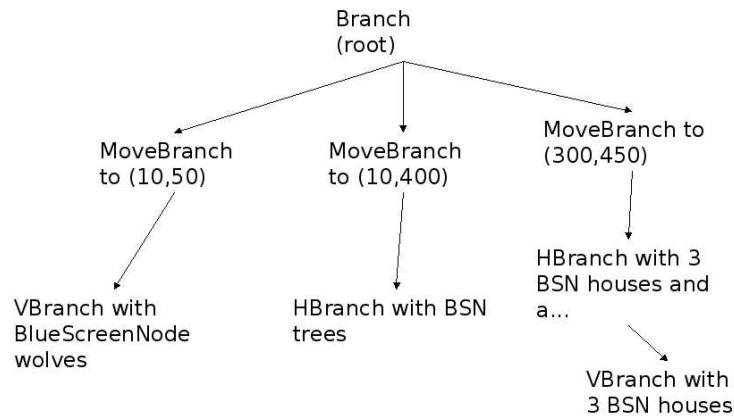


Figure 9.7: Stripping away the graphics—the scene graph is a tree

9.3 The Classes in the SceneGraph

The heart of the classes in this scene graph implementation is `DrawableNode`. All nodes and branches inherit from `DrawableNode` (Figure 9.8), an abstract superclass. The class `DrawableNode` defines how to be a node in a linked list—it defines `next`, and it defines how to add, `insertAfter`, and the other linked list operations. Most importantly, `DrawableNode` defines how to *draw* the node—albeit, abstractly.

`DrawableNode` defines how to `drawOn` a given background picture (Program Program

Example #66 (page 206)). It simply creates a turtle and draws the node with the turtle (`drawWith`). The method `drawWith` is abstract. The definition of how any particular node actually draws itself is left to the super-

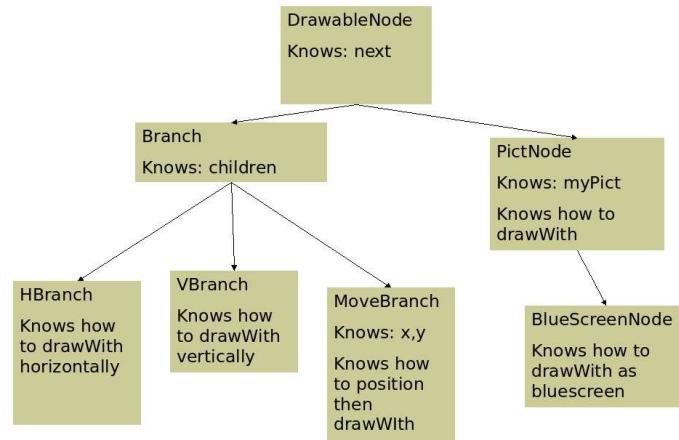


Figure 9.8: Hierarchy of classes used in our scene graph

classes.

Program
Example #65

Example Java Code: The drawing part of **DrawableNode**

```

/*
 * 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);
}
  

```

* * *

Common Bug: Class structure \neq object structure

It's easy to get confused between the tree of classes in Figure 9.8 and the tree of objects in Figure 9.7. The tree of classes describes which classes inherit from which other classes, where `DrawableNode` is the root of the tree, the superclass or parent class of all the other classes in hierarchy. *Objects* created from these classes do form a tree (that's why we made those classes), just not the same tree. The trees don't represent the same things. The tree in Figure 9.7 represents a scene—it doesn't say anything about inheritance. A key observation is that there are several instances of the same class in the scene graph Figure 9.7, though each class appears only once in the class hierarchy Figure 9.8.

`DrawableNode` has two child classes—`Branch` and `PictNode`. `Branch` and its subclasses know how to deal with children. A `Branch` knows how to tell all of its children to draw, and then ask its siblings to draw. `PictNode` and its subclass, `BlueScreenNode` know how to deal with drawing pictures, the leaves of the trees. A `PictNode` holds a picture, and draws its picture by simply asking the turtle pen to drop the picture on the background. A `BlueScreenNode` has a picture, via inheritance from `PictNode`, but uses chromakey to put the picture onto the background.

Subclasses of `Branch` are about positioning and ordering the children of the branch.

- A `MoveBranch` positions the pen to a particular place before asking the children to draw. The `MoveBranch` knows x and y variables for where it should start drawing. An instance of `MoveBranch` is useful for positioning part of the scene.
- The branches `VBranch` and `HBranch` do automatic positioning of the children. A `VBranch` lays out its children vertically, and a `HBranch` lays out its children horizontally. Both add a `gap` variable for how much space to skip between children.

Thus, a `MoveBranch` with a `PictNode` or `BlueScreenNode` essentially does what a `SceneElementLayered` instance does—draw a particular picture at a particular place on the frame. First elements drawn appear below later elements in the tree. A `VBranch` or `HBranch` instance with picture nodes lets you do what a `SceneElementPositioned` instance does—automatic positioning of a picture. Our scene graph implementation, then, has all the capabilities of our earlier linked list, with the addition of structuring.

9.4 Building a scene graph

Let's walk through the WolfAttackMovie class, to show an example of using these classes to construct a scene graph and then create a movie that is so bad that not even the "Rotten Tomatoes" website would review it.

Program
Example #66

Example Java Code: Start of WolfAttackMovie class

```
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, wolfrretreat, hero;
```

It makes sense that an instance of WolfAttackMovie will need to know the root of its scene graph, sceneRoot, and that it would be an instance of Branch. It also makes sense that a WolfAttackMovie instance will need to know a FrameSequence instance for storing and playing back the frames of the memory. What probably does not make sense is why there are three MoveBranch instances defined as instance variables for WolfAttackMovie.

It's not good object-oriented programming practice—rather, we placed the variables there to deal with scope between methods. Within WolfAttackMovie, we have a method setUp which creates the scene graph, and a later method renderAnimation which creates all the frames. To change the position of some elements in the scene, we will change the (x, y) position of the corresponding MoveBranch instances. How do we find the right MoveBranch instances to, for example, move the wolves closer to the village. Option one is to access the right branch through the root: searching the children and nexts to find it. Option two is to store the MoveBranch reference in an instance variable that is then within scope of *both* setUp and renderAnimation, so that we can create it in setUp and change it in renderAnimation. We're using option two here. It's not a great idea to use instance variables to

simply solve scoping problems. The advantage of option two is that it is simple.

The `setUp` method is large in `WolfAttackMovie`. It involves creating all the branches that we will need in the tree, which is a larger task than simply creating the first scene.

Example Java Code: **Start of `setUp` method in `WolfAttackMovie`**

*Program
Example #67*

```
/*
 * 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"));

    //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 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);
```

How it works: The method `setUp` creates the pictures that we will need in the movie, then defines the forest branch of the tree (Figure 9.9). The code above creates the `MoveBranch` instance stored in the variable `forest`. It creates an `HBranch` named `trees` that will hold all the trees. Each of the `BlueScreenNode` instances for each tree is created and named `treenode`. As the trees are created, they are added as children of the `trees` branch. Finally, the `trees` branch is added as a child of the `forest`.

Next, we create the wolves. There's a critical difference between creating the `MoveBranch` for the `wolfentry` versus the `MoveBranch` for the `forest`.

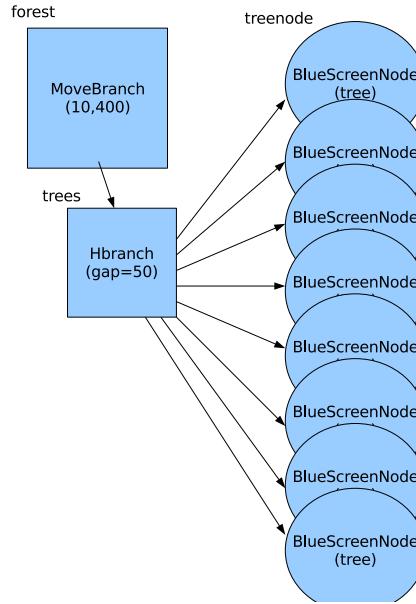


Figure 9.9: The forest branch created in `setUp`—arrows point to children

* * *

Common Bug: Declaring it hides the instance variable

By declaring the variable `forest` as a `MoveBranch`, we make `forest` a variable local to the method `setUp`. That means that the variable `forest` will not exist in `renderAnimation`. The variable `wolfentry` is not declared, so the reference in `setUp` is actually referencing the instance variable `wolfentry`. If we *had* declared `wolfentry` as a `MoveBranch`, the `setUp` would work still. However, `renderAnimation` would be referencing the `wolfentry` defined as an instance variable, which has no variable, and won't actually do anything. The wolves would never move. Declaring the variable in `setUp` makes it impossible to reach the instance variable with the same name.

The `wolfentry` branch is constructed much like the `forest` branch. There is a `MoveBranch` instance for positioning, then a `VBranch` for storing the wolves, then three `BlueScreenNode` instances for storing the actual wolves.

*Program
Example #68*

Example Java Code: **Rest of `setUp` for `WolfAttackMovie`**

```

// 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);

// 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);

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

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

```

How it works: The rest of setUp works similarly. We create a branch for wolfretreat, which is amazingly similar to wolfentry except that the wolves are flipped—they face away from the village. The village is a bit more complicated. There is an HBranch that contains three BlueScreenNode houses and a VBranch (with three more houses) as children. When the village is rendered, it will draw one house, then another, then the third, and then, at the spot where a fourth house would go, there is a vertical line of three more houses going straight up.

At the very end of setUp, we add into the sceneRoot the three branches that appear in the first scene: the forest, the village, and the wolfentry. Where is the wolfretreat and the hero branches? They are not in the initial

scene. They appear later, when we render the whole thing.

Program
Example #69

Example Java Code: **Rendering just the first scene in WolfAttackMovie**

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

When developing a movie, it's important to see if the basic `setUp` worked. When we were creating this first movie, we would often create a `WolfAttackMovie` instance, set it up, then `renderScene` to see how the first scene looked. We would then adjust positions of the forest and the village, and repeat until we were happy with the basic set up.

Now that we have our basic scene graph created, we can render our movie.

Program
Example #70

Example Java Code: **renderAnimation in WolfAttackMovie**

```
/**  
 * Render the whole animation  
 **/  
public void renderAnimation() {  
    frames = new FrameSequence("C:/Temp/");  
    frames.show();  
    Picture bg;  
  
    // First, the nasty wolfies 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);  
    }
```

```

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

// 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);

// 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);
}
}

```

How it works: Rendering a frame from the scene graph is basically these three lines:

```

// Render the frame
bg = new Picture(500,500);
sceneRoot.drawOn(bg);
frames.addFrame(bg);

```

The rest of renderAnimation is changing the scene graph around this rendering.

- During the first 25 frames, the evil wolves are sneaking up on the poor, unsuspecting village. Between each frame, the wolves are moved by changing the position of the wolfentry branch, with a horizontal (*x*) velocity of 5 pixels per frame and a vertical velocity of 10 pixels per frame.

```
wolfentry.moveTo(wolfentry.getXPos()+5, wolfentry.getYPos()+10);
```

- And then, the hero arrives! We insert the hero branch into the code and render.
- Now, we remove the wolfentry branch and insert the wolfretreat branch. We carefully make sure that the wolves are retreating from where they last stopped entering, with the line:

```
// Make sure that they retreat from the same place that they were at
wolfretreat.moveTo(wolfentry.getXPos(), wolfentry.getYPos());
```

- The wolves then scamper away, at a faster rate than when they entered.

We run the movie like this:

```
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();
```

Common Bug: When you run out of memory, within DrJava

Rendering large movies can easily eat up all available memory. You can make DrJava give your Interactions Pane more memory to execute, which would allow you to run large movie code from within DrJava. You can insert the options `-Xms128m -Xmx512m` in the PREFERENCES (under the EDIT menu) to take effect for *all* uses of the Interactions Pane (Figure 9.10).

Common Bug: Increasing memory at the command line

If you are not running DrJava, or you are so tight on memory that you don't even want DrJava in memory, you still have options if you are running out of memory. You can run your movies from the command line and specify the amount of memory you need.

First, you have to be able to run Java from the command line, which means being able to use the `java` and `javac` (*Java compiler*) commands. We have found that some Java installations do not have these set up right. Make sure that your Java JDK bin directory is in your class PATH. For Windows, you use the SYSTEM control panel to change environment variables. To use `javac` to compile your programs, you may need to change the CLASSPATH in System environment variables to point to your JAVA-SOURCE directory. If you have `javac` set up right, you should be able to

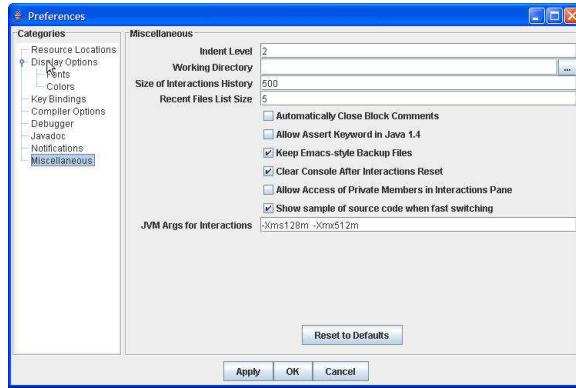


Figure 9.10: Reserving more memory for the Interactions Pane in DrJava’s Preferences pane

compile your class files from outside of DrJava—at a command prompt, type `javac (YourMovieClassNameHere).java`.

To run it, type `java -Xms128m -Xmx512m (YourMovieClassNameHere) -Xms128m` says, “At the very least, give this program 128 megabytes to run.” `-Xmx512m` says, “And give it up to 512 megabytes.”

Now, when you run java like this, you’re actually executing the **public static void** main method. Here’s what we added to `WolfAttackMovie` to make it work from the command line:

```
public static void main(String[] args){
    WolfAttackMovie wam = new WolfAttackMovie();
    wam.setUp();
    wam.renderAnimation();
    wam.replay();
}
```

9.5 Implementing the Scene Graph

Recall lesson from the last chapter, that the way we *think* about a data structure does not have to match the way that it is implemented. While it is useful to think about branches having several children, we do not have to implement multiple references between (say) the forest branch and the tree nodes. Branches must have a way of adding and getting children—that’s part of the definition of a tree. How it’s implemented is up to us.

Our scene graphs are actually implemented as *lists of lists*. All siblings are connected through next links. Each branch is connected to its *first* child, and only the first child, through its children link. The class `Branch` defines `children`, so all branches have the `children` link by inheri-

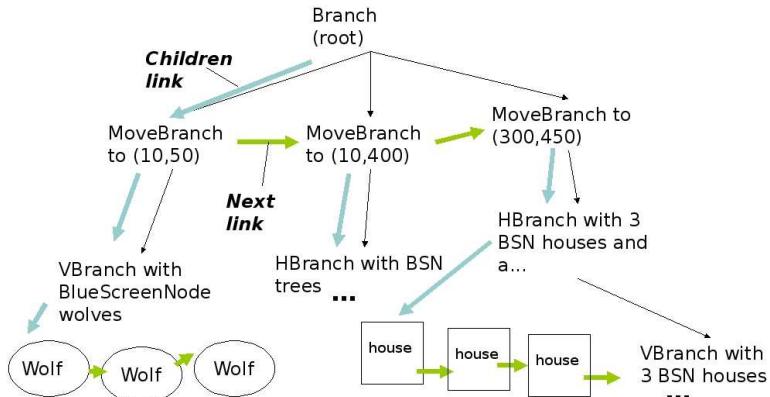


Figure 9.11: The implementation of the scene graph overlaid on the tree abstraction

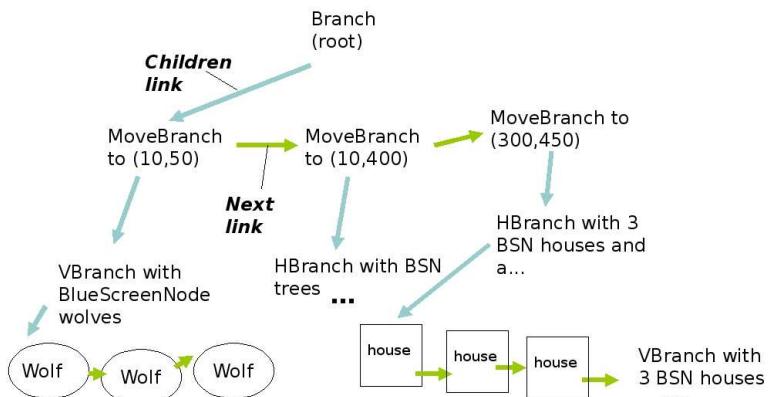


Figure 9.12: The actual implementation of the scene graph

tance. Figure 9.11 shows the abstraction of the scene graph tree (as seen in Figure 9.7), overlaid with the actual implementation.

We can think of our notion of parent nodes with children nodes as an abstraction. It is a useful way of thinking about trees. There are many ways of implementing that abstraction, though. Figure 9.12 shows the actual implementation of our scene graph.

The advantage of a list-of-lists implementation is that we are introducing no new data structure implementation ideas. It's all linked lists. There is one *special* kind of link, the children, both other than that, it's all next links and the same methods we've used all along.

* * *

Common Bug: Can't remove the first child

There is a bug in our implementation of linked lists up to this point in the book, and that same bug extends to our implementation of children in branches. We cannot remove the first item in a list. We will fix this bug in a few chapters, but for now, it dictates the order in which we add children to a branch. Notice that we add the forest as a child of the sceneRoot first. In general, forests don't move (Macbeth excepted) and don't disappear (rain forests excepted). Therefore, we won't have to remove the forest branch to the root, so it's safe to add first. If we would have added wolfentry as the first child of the sceneRoot, we would never have been able to remove it.

**Implementing the abstract superclass for the scene graph:
DrawableNode**

The heart of our scene graph (literally, the base of the class hierarchy for these classes) is DrawableNode. It's actually not complicated at all. It's mostly the same as SceneElement or any of the other linked lists we've seen so-far.

Example Java Code: **DrawableNode**

*Program
Example #71*

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

    /**
     * Constructor for DrawableNode just sets
     * next to null
     */
    public DrawableNode(){
        next = null;
    }

    /**
     * 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;
}

</**
* 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);
}

/** Method to remove node from list , fixing links appropriately.
* @param node element to remove from list.
**/
public void remove(DrawableNode node){
    if (node==this)
    {
        System.out.println("I can't remove the first node from the list.");
        return;
    };

    DrawableNode current = this;
    // While there are more nodes to consider
    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();
    }
}

/**
* Insert the input node after this node.

```

```

* @param node element to insert after this.
*/
public void insertAfter(DrawableNode node){
    // Save what "this" currently points at
    DrawableNode oldNext = this.getNext();
    this.setNext(node);
    node.setNext(oldNext);
}

public DrawableNode last() {
    DrawableNode current;

    current = this;
    while (current.getNext() != null)
    {
        current = current.getNext();
    };
    return current;
}

public void add(DrawableNode node){
    this.last().insertAfter(node);
}
}

```

How it works: Class DrawableNode has two main functions.

- The first is to be an abstract superclass for defining linked lists. For that reason, it has a next variable, operations for getting and setting next, and list operations like add and remove.
- The second is to be “drawable.” Every DrawableNode subclass instance must know how to drawOn and drawWith.

Implementing the leaf nodes: PictNode and BlueScreenNode

Leaf nodes in the scene graphs are the pictures themselves. In WolfAttackMovie, these are the trees, houses, wolves, and hero. The class PictNode simply

drops pictures—we don't actually use it in WolfAttackMovie. It is the superclass for BlueScreenNode, the leaf node class that we use in the example movie.

Program
Example #72

Example Java Code: **PictNode**

```

/*
 * 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;

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

    /**
     * Method to return a string with information
     * about this node
     */
    public String toString()
    {
        return "PictNode (with picture: "+myPict+" and next: "+next;
    }

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

```

How it works: Class PictNode is really quite simple. It has an instance variable, myPict that represents the picture to be shown for the node. The



Figure 9.13: How we actually got some of the bluescreen pictures in this book, such as our hero in WolfAttackMovie

heart of the class is the `drawWith` method which only asks the input turtle to drop the picture.

The class `BlueScreenNode` inherits from `PictNode` and is only slightly more complicated. Instead of dropping the picture, it uses chromakey to remove the blue background for the image. (Figure 9.13 shows how that blue background got there in the first place.)

Example Java Code: **BlueScreenNode**

*Program
Example #73*

```
/*
 * 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
    }

    /**
     * Method to return a string with information
     * about this node
     */
    public String toString()
```

```

{
    return "BlueScreenNode (with picture: "+myPict+" and next: "+next;
}

/*
 * 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());
}
}

```

How it works: The method `drawWith` in `BlueScreenNode` is a little tricky. It asks the turtle that comes in as input “What’s your picture?”, e.g., `Picture bg = pen.getPicture();`. It then calls the `Picture` method `bluescreen` to draw the picture, on the background from the turtle, at the *turtle’s x, y*, i.e., `myPict.bluescreen(bg,pen.getXPos(),pen.getYPos());`.

An interesting side comment to make here is that both of these classes know how to `toString`, that is, return a string representation of themselves. Each simply returns the printable representation of its string and its next. For a brand new node, the next is simply `null`.

```

> BlueScreenNode bsn = new BlueScreenNode(
new Picture(FileChooser.pickAFile()))
> bsn
BlueScreenNode (with picture: Picture, filename /home/guzdial/cs1316/MediaSources/St
height 333 width 199 and next: null

```

Implementing the branches: `Branch`, `MoveBranch`, `VBranch`, and `HBranch`

The branch classes (the class `Branch` and its subclasses) have children links. Because they inherit from `DrawableNode`, they also have `next` links. Thus, branch instances link to both children and siblings—that’s what makes them into branches of trees.

Notice that the type of the children link is `DrawableNode`. That means that *any* other kind of node can be a child of any branch—either leaf nodes or other other branches. Thus, a tree can be of any *depth* (the maximum number of nodes visited to get from the root to a leaf node) or complexity.

*Program
Example #74*

Example Java Code: `Branch`

```
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;
}

/**
 * Method to return a string with information
 * about this branch
 */
public String toString()
{
    String childString = "No children", nextString="No next";
    if (children != null)
        {childString = children.toString();}
    if (next != null)
        {nextString = next.toString();}

    return "Branch (with child: "+childString+" and next: "+
        nextString+)";
}

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

/*
 * 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);
    current = current.getNext();
}

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

How it works: The most important responsibility of branches is to manage their children. Instances of Branch (and its subclasses) know their children reference. A branch adds a child (addChild) by first seeing if children is null—if it is, then the new addition is what children is set to; if it is not, then we add the new addition to the children list.

Branches also know how to `drawWith`. The definition of `drawWith` says:

- First, ask all my children to draw.

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

- Then, tell my next that it can draw.

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

There is a similar algorithm going on in the definition of `toString`, to convert a branch to a printable representation. The algorithm is to collect all the string representations of the children, then of the next, and then return all those pieces.

This means that we can actually watch the construction of a branch, by printing it at each step. Below we see a BlueScreenNode, Branch, and PictNode created and connected. The BlueScreenNode shows up first, and when we add the PictNode, it appears as the next of the BlueScreenNode.

```
> BlueScreenNode bsn = new BlueScreenNode(new Picture(FileChooser.pickAFile()))
> bsn
```

```

BlueScreenNode (with picture: Picture, filename /home/guzdial/cs1316/MediaSources/Student10.jpg
height 333 width 199 and next: null
> Branch b = new Branch()
> b.addChild(bsn)
> b
Branch (with child: BlueScreenNode (with picture:
Picture, filename /home/guzdial/cs1316/MediaSources/student1.jpg
height 369 width 213 and next: null and next: No next)
> PictNode pn = new PictNode(new Picture(FileChooser.pickAFile()))
> b.addChild(pn)
> b
Branch (with child: BlueScreenNode (with picture:
Picture, filename /home/guzdial/cs1316/MediaSources/student1.jpg
height 369 width 213 and next: PictNode (with picture:
Picture, filename /home/guzdial/cs1316/MediaSources/Student7.jpg
height 422 width 419 and next: null and next: No next))

```

The two branch subclasses that position their children automatically, HBranch and VBranch, only provide two methods: a constructor for storing the gap between children, and a new drawWith which moves the turtle appropriately between drawings of children.

Example Java Code: **HBranch**

*Program
Example #75*

```

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;
    }

    /**
     * Method to return a string with information
     * about this branch
     */
    public String toString()
    {
        return "Horizontal "+super.toString();
    }
}

```

```

/*
 * Ask all our children to draw,
 * then let next draw.
 * @param pen Turtle to draw with
 */
public void drawWith(Turtle pen){
    DrawableNode current = children;

    // 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)
    {current = this.getNext();
     current.drawWith(pen);}
}
}

```

*Program
Example #76*

Example Java Code: **VBranch**

```

public class VBranch extends Branch {

    /**
     * Vertical gap between children
     */
    int gap;

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

    /**
     * Method to return a string with information
     * about this branch
     */
}

```

```

public String toString()
{
    return "Vertical "+super.toString();
}

/*
 * Ask all our children to draw,
 * then let next draw.
 * @param pen Turtle to draw with
 */
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)
    {current = this.getNext();
     current.drawWith(pen);}
}
}

```

The class MoveBranch differs in that it keeps track of an x, y position

Example Java Code: **MoveBranch**

*Program
Example #77*

```

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;
    }
}

```

```

}

/*
 * 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;
}

/*
 * Method to return a string with information
 * about this branch
 */
public String toString()
{
    return "Move (" +x+ "," +y+) "+super.toString();
}

/*
 * 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
}
}


```

What would happen if we decided that the `moveTo` of the pen in `MoveBranch`'s `drawWith` should occur *after* processing the children and before processing the next? What would happen if all the branches processed the next before the children? These aren't unreasonable possibilities. They describe different kinds of *traversals* that one can make of the data structure.

The Line between Structure and Behavior

Here's a thought experiment: Where is the *program* in our scene graph?

- Is the program the class `DrawableNode` and its subclasses? That can't be it—by themselves, they don't do anything at all. The `DrawableNode` class hierarchy defines the *behavior*, but not how that behavior is structured and executed.
- Is the program in the class `WolfAttackMovie`? It is true that `WolfAttackMovie` assembles all the pieces from the `DrawableNode` classes in order to define the movie. The class `WolfAttackMovie` defines the *structure* of the movie from those classes.

- However, the actual scenes are not defined in WolfAttackMovie. Instead, the scenes emerge from executing that movie. The actual layout of the trees and village houses is not specified in WolfAttackMovie. Instead, it's determined by the branches in the tree when the tree is rendered. The actual data structure and its use blends structure and behavior.

9.6 Exercises

Ex. 10 — Implement one different traversal of the scene graph, as described at the end of the branch subsection above. Does it make a difference in how the scene looks?

Ex. 11 — Most of renderAnimation in WolfAttackMovie is about changing the location of MoveBranch instances. Create a new class RelativeMoveBranch which is created with a *starting location* (x, y) and a *horizontal velocity* and *vertical velocity*. Each time that an instance of a RelativeMoveBranch is told to update, it automatically updates its current location by its velocity. Rewrite WolfAttackMovie to use this new kind of branch.

Ex. 12 — Both PictNode and BlueScreenNode presume that a character is represented by a single image, just a single picture, e.g., if our wolves moved, changed position of their head and paws, or even appeared to walk. The animation would be improved considerably if the characters changed. Create a new class, CharacterNode that maintains a list of pictures (your choice if the implementation is an array, vector, or linked list of pictures) for a given character. Each time an instance of CharacterNode is asked to drawWith, the current character image is incremented (perhaps randomly among the list of pictures), so that the next rendering draws a different image for the character.

Ex. 13 — Create an animation of *at least* 20 frames with sounds associated with frames.

Here's how you need to do it:

- Create a tree of images that describe your scene. I'll refer to this as the scene structure.
- Create another list with the sounds to be played during the animation of this tree. (Sounds could be a rest!)
- Here's the key part! When you animate your scene (for at least 20 scenes!) and play your sounds, YOU CANNOT MAKE ANY NEW SOUNDS! You must play your sounds out of your list of sounds. In other words, the sounds must be pre-made and assembled in a structure before you start your animation.

You can use and modify any of the data structures that we've described in class. You can create new data structures if you'd like.

You will also need to create a class (call it AnimationRunner). We should be able to use it by creating an instance of the class
(`AnimationRunner ar = new AnimationRunner()`),
and then...

- The method `setUp()` sent to the instance (`ar.setUp()`) should set up the two data structures.
- The method `play()` sent to the instance should play the movie with sound.
- The method `replay()` should replay the `FrameSequence`, but won't do the sound, too.

There are *several* ways to handle the animation. Here's one:

- Render the first frame to a `FrameSequence` instance.
- For each frame that you want to generate (There must be at least 20 frames):
 - Make a change in the scene structure. Any change you want is acceptable.
 - Render the scene structure.
 - Play the sound in the frame list element. Use `blockingPlay()` instead of `play()` on your sound to make the processing wait (synchronize) until the sound is done before moving on. (Hint: If your sound is over 1/10 of a second long, you can't get 10 frames per second! These will be short sounds! Want something to play over two frames? Play the first half in one frame, and the second half in the next frame.) Note: You don't *have* to use `blockingPlay()`. You can use a mixture of `play()` and `blockingPlay()` to get better response while still maintaining synchrony.

10 Lists and Trees for Structuring Sounds

Chapter Learning Objectives

In the past chapters, we used linked lists and trees to structure images and music. In this chapter, we'll add a new medium to our repertoire for dynamic data structures, sampled sounds (e.g., .WAV files). But we'll use *recursion* for the traversals, which provides us a more compact way of describing our traversals.

The computer science goals for this chapter are:

- To iterate across linked lists and trees using recursion.
- To replace elements in a data structure, and to define what the “same” node means.
- To explore different kinds of traversals. We will take an operation embedded in a branch, and choose to apply it to the next branch or to the children branch.

The media learning goals for this chapter are:

- To use our new, dynamic ways of structuring media to describe sound patterns and music.

10.1 Composing with Sampled Sounds and Linked Lists: Recursive Traversals

We originally started down this path of linked lists and trees in order to create a way of composing music flexibly. Our original efforts was with MIDI. However, one might also want to compose sound using sampled or recorded sound. In this chapter, we use the same data structures that we have developed in the previous few chapters in the service of supporting flexible composition of sampled sounds in linked lists.

It's pretty straight forward for us to define a simple linked list structure now. The below definition of a linked list node, SoundElement, that has a Sound instance within it has little new for us.

Program
Example #78

* * *

Example Java Code: **SoundElement**

```


    /**
     * Sounds for a linked list
     */
    public class SoundElement {

        /**
         * The sound this element is associated with
         */
        Sound mySound;

        /**
         * The next element to process
         */
        public SoundElement next;

        /**
         * Constructor sets next to null
         * and references the input sound.
         */
        public SoundElement(Sound aSound){
            next = null;
            mySound = aSound;
        }

        /**
         * Return my sound
         */
        public Sound getSound(){return mySound;}

        /**
         * Play JUST me, blocked.
         */
        public void playSound(){mySound.blockingPlay();}
        public void blockingPlay(){mySound.blockingPlay();}

        /**
         * Provide a printable representation of me
         */
        public String toString(){
            return "SoundElement with sound: "+mySound+ " (and next: "+next+").";
        }

        /**
         * Methods to set and get next elements
        

```

```

* @param nextOne next element in list
*/
public void setNext(SoundElement nextOne){
    this.next = nextOne;
}

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

public void playFromMeOn(){
    this.collect().play();
}

public Sound collect(){
    if (this.getNext() == null)
    {return mySound;}
    else
    {return mySound.append(this.getNext().collect());}
}

public void remove(SoundElement node){
    if (node==this)
    {
        System.out.println("I can't remove the first node from the list.");
        return;
    };

    SoundElement current = this;
    // While there are more nodes to consider
    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();
    }
}

/**
 * Insert the input node after this node.
 * @param node element to insert after this.
 */
public void insertAfter(SoundElement node){
    // Save what "this" currently points at
    SoundElement oldNext = this.getNext();
    this.setNext(node);
    node.setNext(oldNext);
}

/**
 * Return the last element in the list
 */
public SoundElement last() {
    SoundElement current;

    current = this;
    while (current.getNext() != null)
    {
        current = current.getNext();
    };
    return current;
}

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

```

How it works: There are two interesting elements in this linked list implementation. Both of them involve the use of *recursion* for traversing a data structure.

Look at how we define `toString` for `SoundElement`.

```
return "SoundElement with sound: "+mySound+" (and next: "+next+" ).";
```

That's really straightforward, isn't it? Display "SoundElement with...", then the printable representation of the sound, and then the printable representation of the next. When the next is just `null`, that makes perfect

sense how it would work.

```
> Sound s = new Sound("D:/cs1316/mediasources/shh-a-h.wav");
> Sound t = new Sound("D:/cs1316/mediasources/croak-h.wav");
> Sound u = new Sound("D:/cs1316/mediasources/clap-q.wav");
> SoundElement e1 = new SoundElement(s);
> SoundElement e2 = new SoundElement(t);
> SoundElement e3 = new SoundElement(u);
> e1.playFromMeOn();
> e1
SoundElement with sound: Sound file:
D:/cs1316/mediasources/shh-a-h.wav
number of samples: 11004 (and next: null).
```

That makes sense. The interesting thing is that this works, still, when we have additional SoundElement instances in it.

```
> e1.setNext(e2)
> e1
SoundElement with sound:
Sound file: D:/cs1316/mediasources/shh-a-h.wav
number of samples: 11004 (and next:
SoundElement with sound: Sound file:
D:/cs1316/mediasources/croak-h.wav
number of samples: 8808 (and next: null)..).
> e2.setNext(e3)
> e1
SoundElement with sound: Sound file: D:/cs1316/mediasources/shh-a-h.wav
number of samples: 11004 (and next:
SoundElement with sound:
Sound file: D:/cs1316/mediasources/croak-h.wav
number of samples: 8808 (and next:
SoundElement with sound:
Sound file: D:/cs1316/mediasources/clap-q.wav
number of samples: 4584 (and next: null)..).
```

Where did all that text come from? When we try to print next and it is *not null*, then the object that next refers to is asked to convert itself to a string (`toString`). If that next object is also a SoundElement (as in the above examples), then the *same* `toString` method is executed (the one in `SceneElement`). The difference in the next call is that the object being asked to convert itself (`this`) is different.

In the above example, `e1` starts executing `toString`, which then tries to print next—which is `e2`. The node `e2` is then asked `toString`, and when it gets to next, the node `e3` is asked to convert itself `toString`. When `e3` is done (since its next is `null`), `e3.toString()` ends, then `e2.toString()` ends, and finally the original call to print `e1` ends. This process is called a *recursive traversal* of the linked list—we are *traversing* (walking along, visiting) each node of the linked list, by using the same method which repeatedly calls itself. In the case of `toString`, the calls are implicit—they happen when we try to print out next.

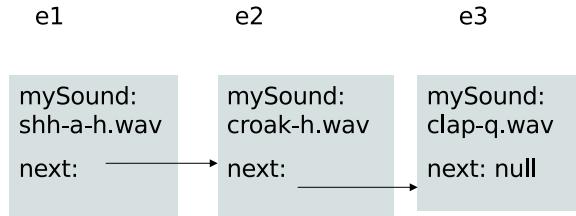


Figure 10.1: The initial SoundElement list

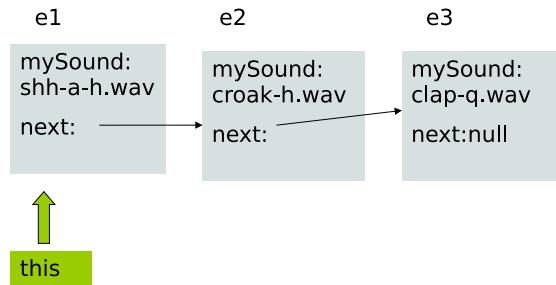


Figure 10.2: As we start executing playFromMeOn()

Tracing a Recursive Traversal

In the above example, if we then asked e1.playFromMeOn(), another recursive traversal would occur. Let's trace what happens. When we start the execution, the linked list looks like Figure 10.1.

The method playFromMeOn() is really short—it only says **this.collect().play();**. The method collect() does all the hard work. It collects the sounds from all the individual nodes into one big Sound instance, so that the big sound can finally be played. So, the interesting work occurs in collect().

So we start out by asking e1 (Figure 10.2), to execute the below code.

```

public Sound collect(){
    if (this.getNext() == null)
        {return mySound;}
  
```

Clearly, getNext() is not **null**, so we end up making the recursive call.

```

else
    {return mySound.append(this.getNext().collect());}
  
```

Our call to e1.collect() is now *frozen*. It's not ended, but it can't go any further yet. It now has to process **this.getNext().collect()**.

We are now asking e2 (which is what **this.getNext()** means) to collect() (Figure 10.3). The question of “Where is e1?” isn't relevant. From e2's

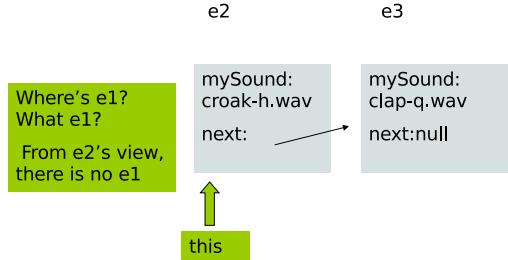


Figure 10.3: Calling e2.collect()

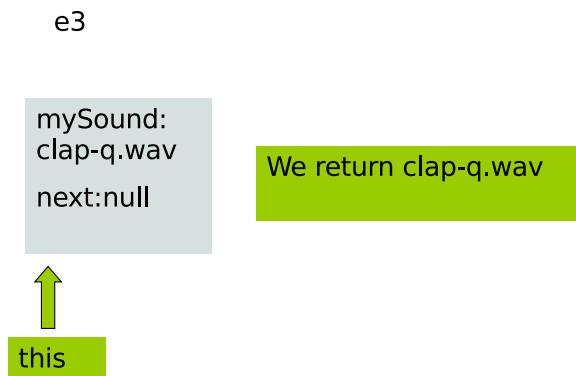


Figure 10.4: Finally, we can return a sound

perspective, e1 doesn't exist. We're simply asking e2.collect().

The node e2 also has a next, so we execute **else return mySound.append(this.getNext().collect());**. Now, we have the execution of e2.collect() *frozen*. Neither e1.collect() nor e2.collect() can end yet—they're waiting in limbo.

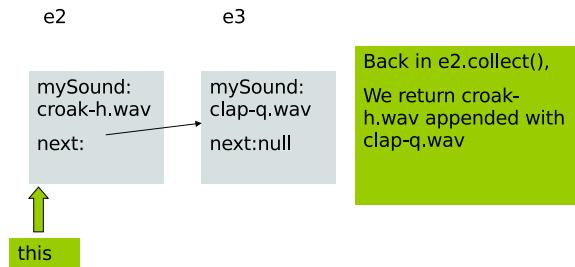
Now, we execute e3.collect() (Figure 10.4). Finally the first part of collect() is true—next is **null**. We return the sound in e3.

Now we can *unfreeze* e2.collect() (Figure 10.5). Now that we have **this.getNext().collect()**, we can append to it the sound of e2 and return back to e1.collect().

Finally, e1.collect() gets the sounds from e2 and e3 appended together, from its execution of **this.getNext().collect()**. The sound from e1 gets appended to the rest, and we finally return from e1.collect(). We return “shh-a-h.wav” appended with “croak-h.wav” appended with “clap-q.wav.” That's what finally gets played.

Making It Work Tip: Tracing your program

It is very important for you to be able to do what we just did in this section.

Figure 10.5: Ending `e2.collect()`

You should be able to trace through your programs when you need to. You don't always have to be able to trace through your programs in this much detail, but when something goes wrong, it helps enormously to be able to do this.

Testing and Replacing

Below is a program that constructs a linked list from the `SoundElement` class.

*Program
Example #79*

Example Java Code: **SoundListText: Constructing a SoundElement list**

```
public class SoundListTest {
    SoundElement root;

    public SoundElement root(){ return root; }

    public void setUp(){
        Sound s = new Sound(FileChooser.getMediaPath( "scratch-h.wav"));
        root = new SoundElement(s);

        s = new Sound(FileChooser.getMediaPath( "gonga-2.wav"));
        SoundElement one = new SoundElement(s);
        root.repeatNext(one,10);
        s = new Sound(FileChooser.getMediaPath( "scratches-q.wav"));
        SoundElement two = new SoundElement(s);
        root.weave(two,3,3);
        s = new Sound(FileChooser.getMediaPath( "clap-q.wav"));
        SoundElement three = new SoundElement(s);
        root.weave(three,5,2);
    }
}
```

```
    root.playFromMeOn();
}
```

Go ahead and try it. You'll notice that you get sounds between other sounds because of the use of repeatNext and weave for SoundElement. It looks much like the code that we developed back for MIDI.

Example Java Code: RepeatNext for SoundElement

*Program
Example #80*

```
/**
 * 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(SoundElement nextOne, int count) {
    SoundElement current = this; // Start from here
    SoundElement 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
    }
}
```

Example Java Code: Weave for SoundElement

*Program
Example #81*

```
/**
 * Weave the input sound count times every skipAmount elements
 * @param nextOne SoundElement 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(SoundElement nextOne, int count, int skipAmount)
{
    SoundElement current = this; // Start from here
    SoundElement copy; // Where we keep the one to be weaved in
```

```

SoundElement 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.getNext() != null) && (skipped < skipAmount))
    {
        current = current.getNext();
        skipped++;
    };

    oldNext = current.getNext(); // Save its next
    current.insertAfter(copy); // Insert the copy after this one
    current = oldNext; // Continue on with the rest
    if (current == null) // Did we actually get to the end early?
        break; // Leave the loop
}
}

```

In both of these methods, we create a *copy* of the input node, using the method *copyNode*. How does that work? What does it mean to copy a SoundElement node?

Program
Example #82

Example Java Code: **copyNode** for **SoundElement**

```

/**
 * copyNode returns a copy of this element
 * @return another element with the same sound
 */
public SoundElement copyNode(){
    Sound copySound;
    if (this.mySound.getFileName() == null)
        {copySound = this.mySound.scale(1.0);} // Does nothing — copies
    else
        {copySound = new Sound(this.mySound.getFileName());} // Copy from file
    SoundElement returnMe = new SoundElement(copySound);
    return returnMe;
}

```

* * *

How it works: What does it mean for one node to be a *copy* of another node? The most critical thing is for the node to have the same sound. It turns out that Sound instances have more than just a bunch of samples associated with them—they also know their filename. Sometimes they do. If a Sound instance is not created from a file, perhaps created from manipulating some other sound (by scaling or appending, for example), then the sound has no filename associated with it.

So, if we want a SoundElement node to have an identical structure, all the way down to the structure of its Sound, then we have to deal with two cases: (a) if there is no filename, and (b) if there is. If there is a filename, then creating the right sound is easy—we simply go create a new sound from the same filename. If there is no filename, we need some way to get a copy of that sound. This method uses one way: scale by 1.0. Scaling by 1.0 is simply a copy of the original sound.

A Problem and Its Solution: Removing a node without recreating the list

You should try that example in SoundListTest. It's really annoying. There are 10 gongs in there that seemingly keep going on and on and on. How can we get rid of them?

The obvious thing to do is to modify the SoundListTest class to use fewer (or zero!) gongs, recompile, and re-run it. What if we couldn't? Imagine that you have a long linked list that has been created through many different operations, and it would take too long or be too hard to recreate it.

What we can do is to walk through the list, find the gongs, and replace the sounds in those nodes with some other sound. We'll call it “de-gonging.”

Example Java Code: **De-gonging the list**

*Program
Example #83*

```
public void degong(){
    Sound gong = new Sound(FileChooser.getMediaPath( "gonga-2.wav" ));
    Sound snap = new Sound(FileChooser.getMediaPath( "snap-tenth.wav" ));
    root.replace(gong, snap);
}
```

We know how to write replace. We're simply going to iterate through the list, find the nodes that have the gong, and replace it with the snap.

However, it's not quite that easy. How do you know which sounds contain the gong sound? That's the trick. The gong sound in the SoundElement is *not* the same object as is input (e.g., it's not `==` to the sound `gong` above). We have to figure out what it means for two objects (the input sound and the sound in a node) to be *equivalent* even if they're not the exact same object.

*Program
Example #84*

Example Java Code: **replace** for SoundElement

```
/*
 * Replace the one sound with the other sound
 * in all the elements from me on.
 * Decide two sounds are equal if come from same filename
 * @param oldSound sound to be replaced
 * @param newSound sound to put in its place
 */
public void replace(Sound oldSound, Sound newSound){
    if (mySound.getFileName() != null)
        {if (mySound.getFileName().equals(oldSound.getFileName()))
            {mySound = newSound;}}
    if (next != null)
        {next.replace(oldSound, newSound);}
}
```

How it works: For our purposes, we'll use a very simple notion of equivalence. If the sound in the node has the same `fileName` as the input sound, it's the same. Obviously, this is a poor approach, e.g., if you have created the sound that you're looking for, you won't possibly make it. It will work for our de-gonging method, though, since we want to find and remove sounds made from the "gong" file.

Notice that, since we're comparing two strings (the filenames), we use the `equals` method. We can't use `==` method because that checks if two objects are *equal*, the exact same object. It's not the case that the two strings are the same. We need to check if the two strings have the same characters—that's what the `equals` method does. We have to check first that there's a `fileName` string at all. If there isn't (e.g., the `Sound` instance was created via `scale` or some other method that returns a `Sound` instance that was never associated with a file), then the `fileName` is `null`—and `null` does not understand `equals`.

The `replace` method has no explicit loop—no `for`, no `while`. Instead, it traverses the list using *recursion*. We call the method `replace` on each segment of the linked list that we want to check. Let's walk through how

that works. Imagine that we execute `e1.replace(croak,clap)` on the list in Figure 10.6.

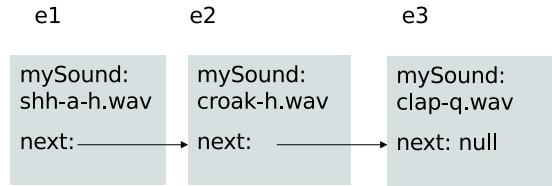


Figure 10.6: Starting out with `e1.replace(croak,clap)`

We start out executing the method `replace` in Figure 10.7:

```
public void replace(Sound oldSound, Sound newSound){
    if (mySound.getFileName() != null)
        { if (mySound.getFileName().equals(oldSound.getFileName()))
            {mySound = newSound;}}
```

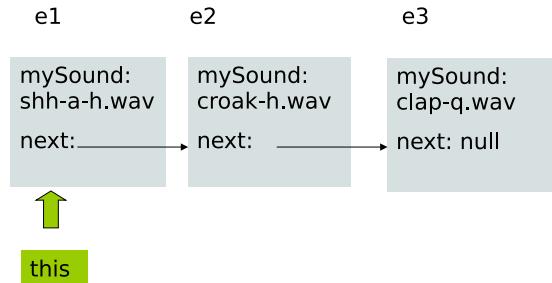


Figure 10.7: Checking the first node

It is clear that the “croak” sound is not the same as the “shh” sound, so we continue on.

```
if (next != null)
    {next.replace(oldSound,newSound);}
```

Now, `e2` is asked to replace(`croak,clap`) (Figure 10.8). Node `e2` *does* contain “croak” so we replace it with a “clap.” Since the `next` of `e2` is *not* **null**, we go on to call `e3.replace(croak,clap)`.

The node `e3` does not contain the “croak” sound, so we don’t replace anything (Figure 10.9). The `next` value on `e3` is **null**, so we return.

While it is invisible to us (no additional code is executed), after `e3.replace(croak,clap)` ends, the method execution for `e2.replace(croak,clap)` also ends. Finally,

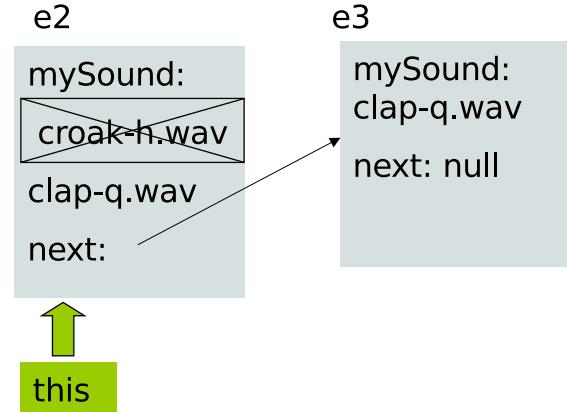


Figure 10.8: Replacing from e2 on

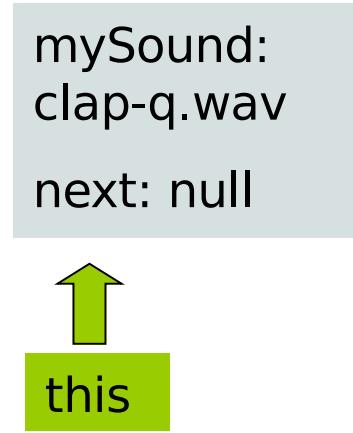


Figure 10.9: Finally, replace on node e3

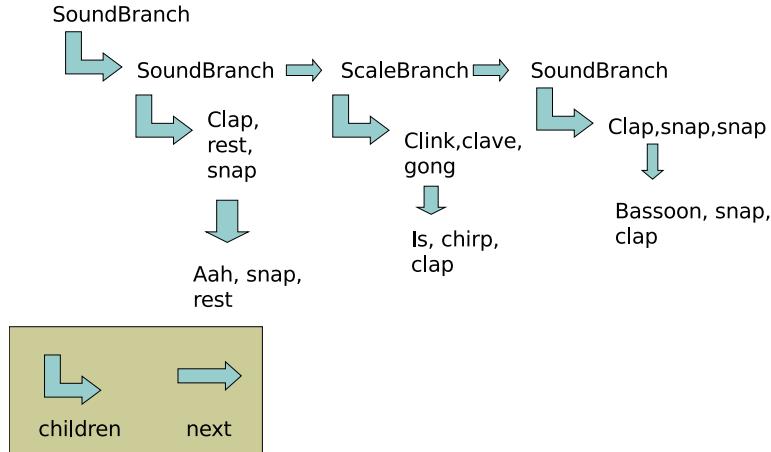


Figure 10.10: Our first sampled sound tree

e1.replace(croak,clap) ends. Those other method executions were awaiting the completion of the later method executions—they could be thought of as being “frozen” or “in limbo.”

10.2 Using Trees to Structure Sampled Sounds

Just as with MIDI phrases and images, we may want to structure our sampled sounds in compositions with a sense of clusters or hierarchy. A long, linear list of sampled sounds is hard to manipulate and think about. We naturally think about sound in clusters: motif expositions and recapitulations, or verses and chorus. So, just like with MIDI phrases and images, we can explore the use of trees to structure our sampled sounds.

Just like with our MIDI phrases and images, we might also embed operations in our branches. We might have branches that reverse the sound collected from the children (all appended together), or scale the children’s sounds, or normalize the volume of the children’s sound. Thus, we can create trees of sampled sounds that combine structure and behavior.

We are going to use an implementation of trees that is similar to the one that we created for images, the structure called a *linked list of lists*. All nodes will have a next, and branches will also have a children link (Figure 10.10). We will create a normal branch, and a ScaleBranch which changes the frequency of the sounds that are children of the branch.

We would create and play this example sound tree like this:

```
Welcome to DrJava.
> SoundTreeExample tree = new SoundTreeExample();
> tree.setUp();
> tree.play()
```

We might then change the scaling factor in our ScaleBranch, then replay the tree:

```
> tree.playScaled(2.0);
> tree.play();
```

Here's what our class SoundTreeExample looks like.

*Program
Example #85*

Example Java Code: **SoundTreeExample**

```
public class SoundTreeExample {

    // Declared here because
    // needed between methods
    ScaleBranch scaledBranch;

    SoundBranch root;
    public SoundBranch root() {return root;}

    public void setUp() {
        Sound clap = new Sound(
            FileChooser.getMediaPath("clap-q.wav"));
        Sound chirp = new Sound(
            FileChooser.getMediaPath("chirp-2.wav"));
        Sound rest = new Sound(
            FileChooser.getMediaPath("rest-1.wav"));
        Sound snap = new Sound(
            FileChooser.getMediaPath("snap-tenth.wav"));
        Sound clink = new Sound(
            FileChooser.getMediaPath("clink-tenth.wav"));
        Sound clave = new Sound(
            FileChooser.getMediaPath("clave-twentieth.wav"));
        Sound gong = new Sound(
            FileChooser.getMediaPath("gongb-2.wav"));
        Sound bassoon = new Sound(
            FileChooser.getMediaPath("bassoon-c4.wav"));
        Sound is = new Sound(
            FileChooser.getMediaPath("is.wav"));
        Sound aah = new Sound(
            FileChooser.getMediaPath("aah.wav"));

        // Build the root
        root = new SoundBranch();
        SoundNode sn;

        // Build the first branch
        SoundBranch branch1 = new SoundBranch();
        // One node, containing 3 sounds appended
```

```

// together to create ONE sound
sn = new SoundNode(clap.
    append(rest).append(snap));
branch1.addChild(sn);
sn = new SoundNode(aah.
    append(snap).append(rest));
branch1.addChild(sn);

// Build our scaled branch
scaledBranch = new ScaleBranch(1.0);
sn = new SoundNode(clink.
    append(clave).append(gong));
scaledBranch.addChild(sn);
sn = new SoundNode(is.
    append(chirp).append(clap));
scaledBranch.addChild(sn);

// Build a third branch, our second SoundBranch
SoundBranch branch2 =
    new SoundBranch();
sn = new SoundNode(clap.
    append(snap).append(snap));
branch2.addChild(sn);
sn = new SoundNode(bassoon.
    append(snap).append(clap));
branch2.addChild(sn);

// Assemble the whole tree
root.addChild(branch1);
root.addChild(scaledBranch);
root.addChild(branch2);
}

public void play(){
    root.playFromMeOn();
}

public void playScaled(double factor){
    scaledBranch.setFactor(factor);
    root.playFromMeOn();
}
}

```

How it works: There are a lot of similarities between this class and the WolfAttackMovie class. Each create a root variable that holds the main branch of the tree. Some branches that need to be accessed between methods are declared in variables in the class. The setUp() methods in both

classes have a very similar structure. Each creates a branch, creates the nodes that go in that branch, and then loads the nodes into the branch. SoundTreeExample may be a little confusing, in that each node *only* contains a single Sound, but in this example, each node's Sound is a collection of three appended sounds. It's still one Sound per SoundNode.

Once we create our tree, we can explore the structure of the tree by simply printing it out. All the objects in the sound tree have `toString` methods. Since we know the implementation of the tree, we can walk the children and `next` links to explore the whole tree.

```
> tree // Not very useful in itself
SoundTreeExample@92b1a1
> tree.root() // Way useful
SoundBranch (with child:
SoundBranch (with child:
SoundNode (with sound: Sound number of samples: 28568 and next:
SoundNode (with sound: Sound number of samples: 46034 and next:
null and next:
ScaleBranch (1.0) SoundBranch (with child:
SoundNode (with sound: Sound number of samples: 47392 and next:
SoundNode (with sound: Sound number of samples: 32126 and next:
null and next:
SoundBranch (with child:
SoundNode (with sound: Sound number of samples: 8452 and next:
SoundNode (with sound: Sound number of samples: 28568 and next:
null and next: No next)) and next: No next)
> tree.root().children // All the children of the root
SoundBranch (with child:
SoundNode (with sound:
Sound number of samples: 28568 and next:
SoundNode (with sound:
Sound number of samples: 46034 and
next: null and next:
ScaleBranch (1.0) SoundBranch (with child:
SoundNode (with sound:
Sound number of samples: 47392 and next:
SoundNode (with sound:
Sound number of samples: 32126 and next: null and next:
SoundBranch (with child: SoundNode (with sound:
Sound number of samples: 8452 and next:
SoundNode (with sound:
Sound number of samples: 28568 and next: null and next: No next)))
> tree.root().children.getNext() // Second branch on
ScaleBranch (1.0) SoundBranch (with child:
SoundNode (with sound:
Sound number of samples: 47392 and next:
SoundNode (with sound:
Sound number of samples: 32126 and next: null and next:
SoundBranch (with child:
SoundNode (with sound:
```

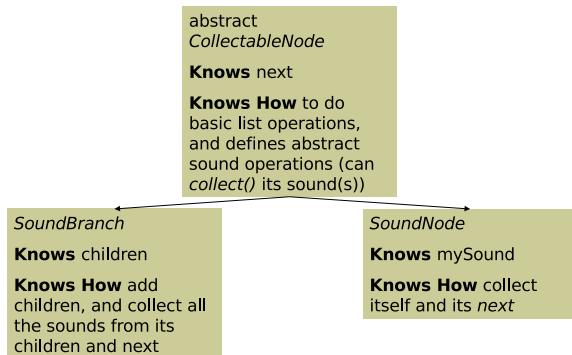


Figure 10.11: The core classes in the `CollectableNode` class hierarchy

```

Sound number of samples: 8452 and next:
SoundNode (with sound:
Sound number of samples: 28568 and
next: null and next: No next))
> tree.root().children.getNext().getNext() \\ 3rd Branch
SoundBranch (with child:
SoundNode (with sound:
Sound number of samples: 8452 and next:
SoundNode (with sound:
Sound number of samples: 28568 and next: null
and next: No next)
> tree.root().children.getNext().getNext().getNext() \\ 4th branch
null
  
```

Implementing a Sound Tree

How we build a tree of sounds is very much like how we built a tree of pictures.

- Set up a general *node* abstract superclass that all other node and branch classes inherit from. In our sound tree, this superclass is `CollectableNode`—a node from which a sound can be collected.
- Create a *leaf* node class that will store our data of interest (sounds). Here, that's a `SoundNode`
- Create a *branch* node that will store collections of leaves and references to other branches—a `SoundBranch` (Figure 10.11).
- Create (as we wish) *branch nodes with operations* that do things to the children of this branch. We have one example of a branch with operations, `ScaleBranch` (Figure 10.12).

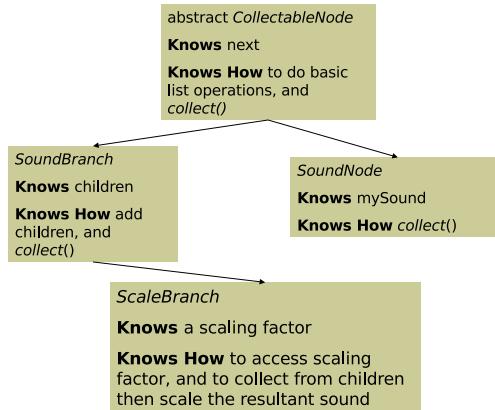


Figure 10.12: Extending the class hierarchy with ScaleBranch

Let's go through these classes in turn. We should note that much of the `CollectableNode` class hierarchy looks much like the code in the `DisplayableNode` class hierarchy.

Program
Example #86

Example Java Code: **CollectableNode**

```

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

    /**
     * Constructor for CollectableNode just sets
     * next to null
     */
    public CollectableNode(){
        next = null;
    }

    /**
     * Methods to set and get next elements
     * @param nextOne next element in list
     */
  
```

```

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

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

// All the rest of the linked list
// methods are there, too: insertAfter(), last()...

/***
 * Play the list of sound elements
 * after me
 ***/
public void playFromMeOn(){
    this.collect().play();
}

/***
 * Collect all the sounds from me on
 ***/
abstract public Sound collect();
}

```

How it works: For the most part, CollectableNode is simply another linked list node class. It has a next link and lots of linked list methods. In addition, it defines abstract methods for collect()—the method for collecting a Sound from the node. In the abstract superclass CollectableNode, the method does nothing at all. The method playFromMeOn() is defined here, too. It simply plays what gets collected.

Example Java Code: **SoundNode**

*Program
Example #87*

```

/*
 * SoundNode is a class representing a sound
 * node in a sound tree.
 */
public class SoundNode extends CollectableNode {
    /***
     * The sound I'm associated with
     ***/
    Sound mySound;
}

```

```

/*
 * Make me with this sound
 * @param sound the Sound I'm associated with
 */
public SoundNode(Sound sound){
    super(); // Call superclass constructor
    mySound = sound;
}

/**
 * Method to return a string with information
 * about this node
 */
public String toString()
{
    return "SoundNode (with sound: "+mySound+" and next: "+next;
}

/**
 * Collect all the sounds from me on,
 * recursively.
 */
public Sound collect(){
    if (this.getNext() == null)
        {return mySound;}
    else
        {return mySound.append(this.getNext().collect());}
}

```

How it works: The class SoundNode models leaf nodes in our sound trees. Instances of SoundNode contains a Sound. The most interesting thing about SoundNode is how it defines `toString()` and `collect()`.

- A SoundNode instance converts itself to a String by converting its Sound and its next to a String. That's easy to say, and that's why we define it that way. Tracing it takes some thinking about it.
- A SoundNode collects its sound by (a) if it has no next node (next is `null`), then returning its sound; (b) else, returning its sound appended with whatever its next will return when we ask it to `collect()`. Its this last part that is recursive. Most powerfully, though, the definition is easy to say and easy to understand—"When I'm told to collect, I grab me and whatever my buddy has to return." It's exactly how you collect papers when you're told to hand them to the end of the row: you stick yours on top of what your neighbor returns, then pass the whole stack to the next person.

* * *

Example Java Code: **SoundBranch**

*Program
Example #88*

```

public class SoundBranch extends CollectableNode {
    /*
     * A list of children to draw
     */
    public CollectableNode children;

    /*
     * Construct a branch with children and
     * next as null
     */
    public SoundBranch(){
        super(); // Call superclass constructor
        children = null;
    }
    /**
     * Method to add nodes to children
     */
    public void addChild(CollectableNode child){
        if (children != null)
            {children.add(child);}
        else
            {children = child;}
    }
    /**
     * Method to return a string with information
     * about this branch
     */
    public String toString()
    {
        String childString = "No children", nextString="No next";
        if (children != null)
            {childString = children.toString();}
        if (next != null)
            {nextString = next.toString();}

        return "SoundBranch (with child: "+childString+" and next: "+
               nextString+)";
    }

    /**
     * Collect all the sound from our children,
     * then collect from next.
     */
    public Sound collect(){

```

```

Sound childSound;

if (children != null)
{childSound = children.collect();}
else
{childSound = new Sound(1);}

// Collect from my next
if (this.getNext() != null)
{childSound=childSound.append(this.getNext().collect());}

return childSound;
}

```

How it works: The class SoundBranch is the generic class for branches in a sound tree. Again, there is a similarity with the Branch class that we used with images.

- The definition for adding a child (addChild) is exactly the same as in Branch—if there’s no children link, set it; else, add() to the children.
- The definition of toString() is a little complicated by the fact that we have to traverse both the children and next links to print a branch. We explicitly ask each variable to convert to toString() (being careful never to ask **null** to convert itself to a string—though it would be glad to do so). Then we return the string of these pieces concatenated together.
- How a SoundBranch collects all its sounds, in collect(), is straightforward to say. First, we want to collect all the children’s sounds. If the node children is empty (**null**), we create a very small (1/22,050th of a second) sound to serve as a dummy, empty sound. If there are children, collect() from the children. We then return the childSound appended with the collection of sounds from next. In short form, a SoundBranch collects sounds by collecting all of its children’s sounds (if there are any children) and all of its sibling’s sounds, and return them appended together.

*Program
Example #89*

Example Java Code: **ScaleBranch**

```

public class ScaleBranch extends SoundBranch {

    /** Amount to scale */
    double factor;
}

```

```


    /**
     * Construct a branch with this factor
     */
    public ScaleBranch(double nufactor){
        super(); // Call superclass constructor
        this.factor = nufactor;
    }

    /** Accessors */
    public double getFactor() {return this.factor;}
    public void setFactor(double nufactor) {this.factor = nufactor;}

    /**
     * Collect all the sound from our children,
     * then collect from next. Scale the children
     */
    public Sound collect(){

        Sound childSound;

        if (children != null)
            childSound = children.collect().scale(factor);
        else
            childSound = new Sound(1);

        // Collect from my next
        if (this.getNext() != null)
            {Sound nextSound=this.getNext().collect();
            childSound = childSound.append(nextSound);}

        return childSound;
    }

    /**
     * Method to return a string with information
     * about this branch
     */
    public String toString()
    {
        return "ScaleBranch ("+factor+") "+super.toString();
    }
}


```

How it works: The class ScaleBranch is a kind of SoundBranch, so it is a branch that knows how to add children and how to collect those children's sounds. In addition, a ScaleBranch has a factor that it uses to scale the

sound from its children—scaling it up in frequency or down. This change requires a slightly different constructor (for setting the scaling factor) and accessors. The biggest change is in the `collect()` method. Now, when children are gathered, they are also scaled:

```
if (children != null)
{childSound = children.collect().scale(factor);}
else
{childSound = new Sound(1);}
```

Tracing a Recursive Tree Traversal

We have now seen how we can build and use a tree of sounds, and how that tree of sounds is implemented. In this section, we aim to understand how the tree traversal occurs dynamically. How is it that this simple code in the `collect()` methods traverses the whole tree?

We start from this code:

```
Welcome to DrJava.
> SoundTreeExample tree =
      new SoundTreeExample();
> tree.setUp();
> tree.root().collect().play()
```

The object in `tree` has a `root()` (an instance of the class `SoundBranch`, as we saw earlier) which is created in `setUp()`. We then ask the root to `collect()` all its sounds into one big `Sound` which we can then `play()`. How does that sound get collected?

When we start executing `tree.root().collect()`, the tree and **this** looks like Figure 10.13. We start executing the below method for `collect()` with **this** pointing at the root.

```
public Sound collect(){
    Sound childSound;
    if (children != null)
        {childSound = children.collect();}
    else
        {childSound = new Sound(1);}

}
```

Since the root clearly has children, we ask the children to `collect()`. Figure 10.14. The call to `root().collect()` is frozen, as we have to wait to get `children.collect()` to return before the call to the root can complete. The same code gets executed as with the root, and since **this** is also a `SoundBranch`, the same code executes.

Again, **this** branch does have children, so we ask the `children.collect()` (Figure 10.15). Now, we're executing a different `collect()` method, because now **this** points at a `SoundNode`. So, we execute this version:

```
public Sound collect(){
    if (this.getNext() == null)
        return mySound;
```

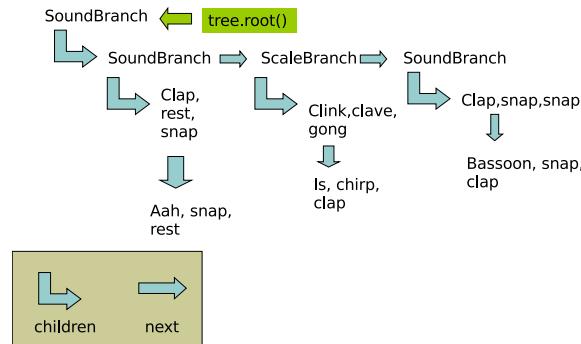


Figure 10.13: Starting out with tree.root().collect()

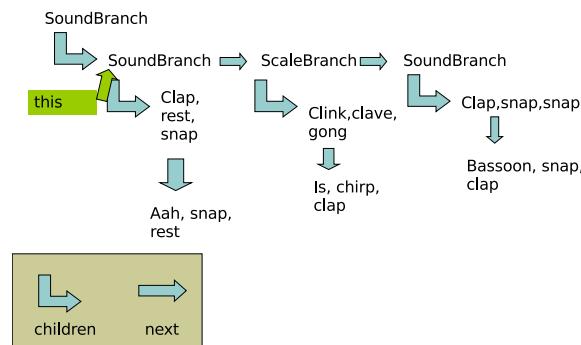


Figure 10.14: Asking the root's children to collect()

```

else
{return mySound.append(this.getNext().collect());}
}

```

The current **this** does have a next, so we need to collect() from there before we can finish this method. We move on to the next SoundNode (Figure 10.16). This one has no next, so we return “Aah, snap, rest” (all appended together as one Sound object).

At this point, we’re back in the position of Figure 10.15, but later in the method collect(). We now have collected the sound from the next, so we can append **this** sound. We now return “clap, snap, rest” with “aah, snap, rest.”

We’re now back at the position of Figure 10.14. We now have the children’s sound, so we can finish the SoundBranch’s collect() method.

```

// Collect from my next
if (this.getNext() != null)
{Sound nextSound=this.getNext().collect();
childSound = childSound.append(nextSound);}

```

258 CHAPTER 10. LISTS AND TREES FOR STRUCTURING SOUNDS

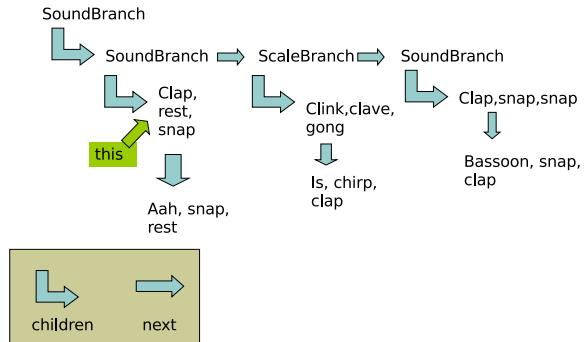


Figure 10.15: Asking the first SoundNode to collect()

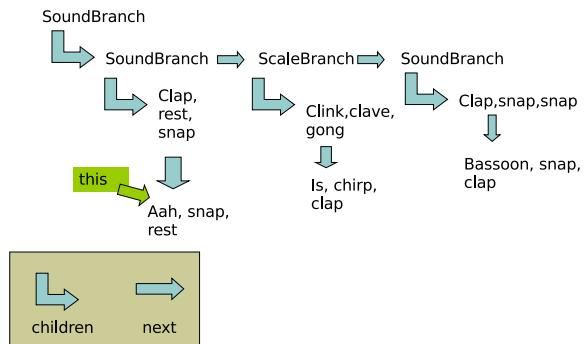


Figure 10.16: Asking the next SoundNode to collect()

```

return childSound;
}
  
```

This SoundBranch *does* have a next, so we again freeze this method invocation while we ask `this.getNext().collect()`. We are now collecting from the ScaleBranch that is next to the SoundBranch (Figure 10.17).

We know something about what will happen next. The `collect()` method in ScaleBranch will gather up the sound from the children list. Then that resultant sound will be scaled.

```

public Sound collect(){
    Sound childSound;
    if (children != null)
        {childSound = children.collect().scale(factor);}
    else
        {childSound = new Sound(1);}
  
```

We can see from the tree that we will gather “clink, clave, gong” appended to “is, chirp, clap.” That will get scaled to whatever factor is in the

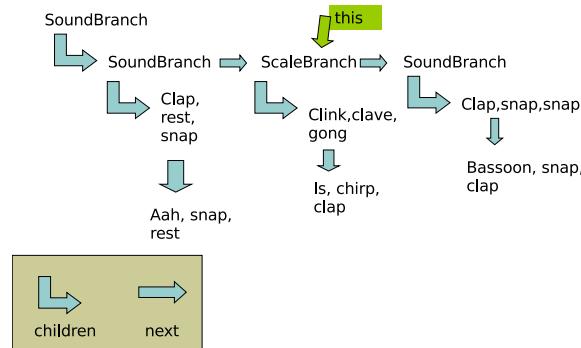


Figure 10.17: Collecting from the next of the SoundBranch

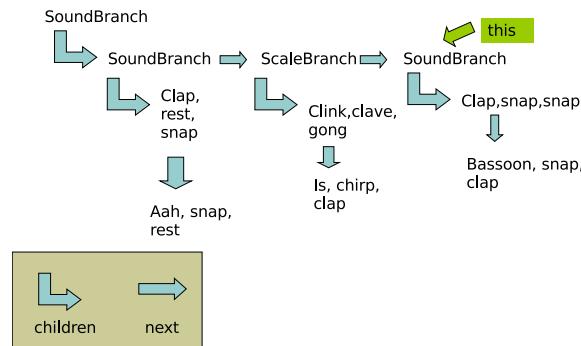


Figure 10.18: Collecting from the last SoundBranch

ScaleBranch. Then the ScaleBranch collect() method will finish:

```
// Collect from my next
if (this.getNext() != null)
{Sound nextSound=this.getNext().collect();
childSound = childSound.append(nextSound);}
return childSound;
}
```

This is the same as in the collect() method of SoundBranch, so we know what's going to happen here. The ScaleBranch invocation of collect() stops executing and waits for **this.getNext().collect()** to execute. We ask the last SoundBranch to collect() (Figure 10.18).

It's worthwhile to consider just what methods are paused, frozen, waiting for other methods to finish. We call that list of methods that are currently pending the *stack trace*. These are quite literally in a *stack*. The last one on must be the first one popped. So, as of Figure 10.18, we have methods in play:

- The currently executing method, the call to collect() on the last SoundBranch.
- The call to collect() on the ScaleBranch is waiting for the collection from its next.
- The call to collect() on the first SoundBranch is waiting for the collection from the ScaleBranch.
- And the call that started it all—the root SoundBranch is waiting for the collection from its children.

We know that calling collect() on the last SoundBranch will result from collecting its children's sounds: “clap, snap, snap” appended with “bassoon, snap, clap” (Figure 10.18). This last SoundBranch has no next! So this call simply returns the sound “clap, snap, snap, bassoon, snap, clap.”

We return to the state of Figure 10.17 and pop the method off the stack. We can now finish the call to collect() on the ScaleBranch. We return back up “clink,clave,gong,is,chirp,clap” scaled by the factor, appended with “clap, snap, snap, bassoon, snap, clap.”

We now pop the next method off the stack, and we're at Figure 10.14. We take the sound from the next (the ScaleBranch and the rest of the tree), and append the sound from its children. So it returns “clap,rest,snap,aah,snap,rest”, plus “clink,clave,gong,is,chirp,clap” scaled by the factor, plus “clap, snap, snap, bassoon, snap, clap.”

We pop the last method off the stack—we're at the collect() of the root. Obviously, the root has no next, so we simply return the sound from the root's children: “clap,rest,snap,aah,snap,rest”, plus “clink,clave,gong,is,chirp,clap” scaled by the factor, plus “clap, snap, snap, bassoon, snap, clap.” And that's the sound that plays.

What we just traced might be called an *in-order traversal*. We visited the children, did whatever the branch needed to do (e.g., scale), then processed the next. There are different orderings of traversals. Imagine a form of ScaleBranch that scaled the result from the next rather than the children. We will visit more of these traversals in the next chapter.

Where we're going next

Take a look at DrawableNode and CollectableNode. Pretty darn similar, aren't they? Duplicated code is a bad idea for several reasons:

- Duplicated code is hard to maintain. Let's say that we figured out a better way to do add() or reverse(). Right now, we have nearly identical code in two different classes, so we would need to fix both of those. What are the odds that we make a mistake at some point and forget to fix one of them when we fix the other?
- Duplicated code is a waste of space. Why should the exact same code be in memory in two different places.

- Duplicated code strains our notion of *responsibility-driven design* that is important for object-oriented programming. Whose responsibility is it to provide `add()`? Is it the responsibility of both `DrawableNode` and `CollectableNode`? That seems odd. Rather, it should be the responsibility of some third class, that both of these classes use. That's where we're going in the next chapter.

In the next chapter, we will correct this problem, and make it easier to create more linked lists and trees in the future. Later, we will find that graphical user interfaces are *also* trees, and that different kinds of traversals of the same graphical user interface tree creates different window appearances.

Ex. 14 — Change the `replace` method so that it actually compares the sounds (sample-by-sample) rather than simply comparing the filenames.

Ex. 15 — Implement a subclass of `SoundBranch` called `ReverseBranch` that reverses the sound returned from collecting the children's sounds. Build a sound tree example using your new class.

Ex. 16 — Implement a subclass of `SoundBranch` called `NormalizeBranch` that normalizes volume returned from collecting the children's sounds. Build a sound tree example using your new class.

Ex. 17 — Implement a subclass of `SoundBranch` called `VolumeChangeBranch` that remembers a factor for use in increasing or decreasing the volume returned from collecting the children's sounds. Build a sound tree example using your new class.

Ex. 18 — Create a new version of `ScaleBranch` that scales its next rather than its children. Build a sound tree example using your new class.

Ex. 19 — Record some individual sounds (singing perhaps?) that represents verses and chorus of some song. Use a `SoundTree` to organize these sounds into the complete song with the right structure.

11 Generalizing Lists and Trees

Chapter Learning Objectives

In the last two chapters, we used trees to create an animation (the `DrawableNode` class hierarchy) and a complex sound structure (the `CollectableNode` class hierarchy). In this chapter, we create a generalized class that represents any kind of linked list structure. We will then go on to explain how to create generalized tree structures that are particularly efficient for some tasks (like searching).

The computer science goals for this chapter are:

- To *factor* out common functionality from two classes to create a new abstract superclass.
- To develop strategies for how to respond to Java compiler errors.
- To understand the relative characteristics of arrays, linked lists, and trees for various manipulations.
- To use a binary tree structure and understand its value in a search tree.
- To explore different kinds of traversals on binary trees.

11.1 Refactoring a General Linked List Node Class

We are going to remove the duplication between `DrawableNode` and `CollectableNode` by creating a new class, `LLNode`, that has the responsibility of being a linked list node. `LLNode` will be an abstract superclass—we'll never want just an `LLNode` (for one thing, it will have no data—it will just be a `next`). There are several advantages to this structure:

- We remove the duplication of code between `DrawableNode` and `CollectableNode`.
- We push the responsibility of being a linked list node to a class that bears just that responsibility.
- Once we create an `LLNode` class, we can create a linked list of anything by simply subclassing `LLNode`.

Creating an LLNode class is pretty easy. Certainly, LLNode will need a next instance variable. We'll simply copy-paste the linked list code from either of DrawableNode or CollectableNode, then change all the appropriate variable types to LLNode.

Program
Example #90

Example Java Code: **LLNode, a generalized linked list node class**

```

abstract public class LLNode{
    /**
     * The next branch/node/whatever to process
     */
    public LLNode next;

    /**
     * Constructor for LLNode just sets
     * next to null
     */
    public LLNode(){
        next = null;
    }

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

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

    /**
     * Method to remove node from list , fixing links appropriately.
     * @param node element to remove from list.
     */
    public void remove(LLNode node)
    {
        if (node==this)
        {
            System.out.println("I can't remove the first node from the list.");
            return;
        };

        LLNode current = this;
        // While there are more nodes to consider
    }
}

```

```

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();
}
}

</**
* Insert the input node after this node.
* @param node element to insert after this.
**/
public void insertAfter(LLNode node){
    // Save what "this" currently points at
    LLNode oldNext = this.getNext();
    this.setNext(node);
    node.setNext(oldNext);
}

/**
* Return the last element in the list
**/
public LLNode last() {
    LLNode current;

    current = this;
    while (current.getNext() != null)
    {
        current = current.getNext();
    };
    return current;
}
/**
* Return the count of the elements in the list
**/
public int count() {
    LLNode current;
    int count = 1;

    current = this;
    while (current.getNext() != null)
    {
        count++;
        current = current.getNext();
    }
}

```

```

    };
    return count;
}

public void add(LLNode node){
    this.last().insertAfter(node);
}

public LLNode reverse() {
    LLNode 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;
}

```

Here's where it gets interesting. Can we now make CollectableNode and DrawableNode subclasses of LLNode, rip out all the replicated linked list code from CollectableNode and DrawableNode, *and then* make our WolfAttackMovie animation and sound tree examples work again? The process we're engaging in is called *refactoring*—we are moving replicated code *up* in the class

hierarchy, and making sure that everything still works afterward.

Making WolfAttackMovie work again

We'll start with the animation. We change `DrawableNode` in two ways:

- We change the class definition to extend `LLNode`.
- We remove the linked list code. There's a bit more to do here beyond simply deleting methods like `add()`. For example, we remove the definition of the `next` field, we change the constructor to simply call the `super`.

Example Java Code: **DrawableNode, with linked list code factored out** *Program Example #91*

```
/*
 * Stuff that all nodes and branches in the
 * scene tree know.
 */
abstract public class DrawableNode extends LLNode {

    /**
     * Constructor for DrawableNode just sets
     * next to null
     */
    public DrawableNode(){
        super();
    }

    /**
     * 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);
    }
}
```

This new version of DrawableNode is much smaller and seems more appropriate for its responsibility. If the class LLNode now has the responsibility of “Being a linked list node,” then class DrawableNode has the responsibility of “Being a linked list node that can be drawn.” The amount of code and the methods for that, as seen above, seems appropriate for that responsibility.

When we click COMPILE now, it doesn’t work. We get a lot of errors. Refactoring doesn’t come for free.

- We get two errors in HBranch. The first one looks like this:

```
HBranch.java:38: incompatible types
found   : LLNode
required: DrawableNode
```

It turns out that there are two errors in the method that the above error points at, and they’re both really the same thing.

```
public void drawWith(Turtle pen){
    DrawableNode current = children;

    // Have my children draw
    while (current != null){
        current.drawWith(pen);
        pen.moveTo(pen.getXPos() + gap, pen.getYPos());
        current = current.getNext(); // Error here, line 38
    }

    // Have my next draw
    if (this.getNext() != null)
    {current = this.getNext(); // Error on this line, too
     current.drawWith(pen);}
}
```

The problem is that we are going to try to call drawWith() with the object referenced in current which comes from getNext(). The method getNext() now returns type LLNode, yet current is of type DrawableNode. We *need* current to be a DrawableNode—general linked lists don’t know how to drawWith(). That’s what’s meant by INCOMPATIBLE TYPES: getNext() returns an LLNode, and we’re stuffing it into a DrawableNode variable. Now, we know that this will always work—all the nodes linked up to a DrawableNode will, in fact, be kinds of Drawables. We have to tell Java that, by *casting*.

```
public void drawWith(Turtle pen){
    DrawableNode current = children;

    // Have my children draw
```

```

while (current != null){
    current.drawWith(pen);
    pen.moveTo(pen.getXPos() + gap, pen.getYPos());
    current = (DrawableNode) current.getNext();
}

// Have my next draw
if (this.getNext() != null)
{current = (DrawableNode) this.getNext();
    current.drawWith(pen);}
}
}

```

- The second set of errors is identical to the first, just now in VBranch rather than in HBranch. (This does suggest that the drawWith code in those two classes is nearly identical, and we could further refactor these two classes.) We similarly have incompatible types in drawWith() because our current variable is an instance of DrawableNode but getNext() returns an LLNode. The fix is the same—we add casting.

No more compiler errors, so let's try to make our animation.

```
> WolfAttackMovie wam = new WolfAttackMovie()
> wam.setUp()
```

We get a new error, at *runtime*.

```
NoSuchMethodError: DrawableNode.add(LDrawableNode;)V
at Branch.addChild(Branch.java:37)
at WolfAttackMovie.setUp(WolfAttackMovie.java:39)
```

That is undeniably a strange error. Let's recompile Branch. Now, we get an error that we have seen before:

```
Branch.java:53: incompatible types
found   : LLNode
required: DrawableNode
```

It turns out to be the exact same error in the exact same place: drawWith() uses a variable current of type DrawableNode, but getNext() returns an LLNode. So, we have to add the casts here, too—same places, in fact. (Again, this suggests the need for refactoring in the branch classes.)

We can try again with WolfAttackMovie, as we did in a previous chapter. This time, we are rewarded with scenes of doggies attacking a village until the brave hero appears. We have successfully refactored the DrawableNode class hierarchy.

Making sound trees work again

Now, let's shift our attention to sound trees. We start with class CollectableNode. Just like with DrawableNode, we take two steps:

- We change the class definition to extend LLNode.
- We remove the linked list code.

Program
Example #92

Example Java Code: CollectableNode, with linked list code factored out

```
/*
 * Stuff that all nodes and branches in the
 * sound tree know.
 */
abstract public class CollectableNode extends LLNode {

    /**
     * Constructor for CollectableNode just sets
     * next to null
     */
    public CollectableNode(){
        super();
    }

    /**
     * Play the list of sound elements
     * after me
     */
    public void playFromMeOn(){
        this.collect().play();
    }

    /**
     * Collect all the sounds from me on
     */
    abstract public Sound collect();
}
```

The new version of CollectableNode is even smaller than the new version of DrawableNode. For the most part, all CollectableNode really does is to define playFromMeOn() (which is only a single line) and declare the abstract method collect().

Perhaps surprisingly, when we compile the new CollectableNode and SoundTreeExample, no errors arise! The error occurs at runtime, when we try to use it:

```
> SoundTreeExample ste = new SoundTreeExample()
> ste.setUp()
```

```
NoSuchMethodError: CollectableNode.add(LCollectableNode;)V
at SoundBranch.addChild(SoundBranch.java:37)
at SoundTreeExample.setUp(SoundTreeExample.java:27)
```

So, we open up `SoundBranch` and recompile that. Now, we get a compiler error in `SoundBranch`—one we've seen before.

```
SoundBranch.java:58: incompatible types
found   : LLNode
required: CollectableNode
```

The method at line 58 is in `collect()`.

```
public Sound collect(){

    Sound childSound;
    CollectableNode node;

    if (children != null)
        {childSound = children.collect();}
    else
        {childSound = new Sound(1);}

    // Collect from my next
    if (this.getNext() != null)
        { node=this.getNext(); // ERROR IS HERE
            childSound=childSound.append(node.collect()); }

    return childSound;
}
```

The variable `node` has type `CollectableNode`, and `getNext()` (now) returns an `LLNode`. We can repair this with a simple cast: `node=(CollectableNode) this.getNext()`.

When we try to execute a sound tree example, we get the next error, at runtime.

```
> SoundTreeExample ste=new SoundTreeExample()
> ste.setUp()
> ste.play()
NoSuchMethodError: SoundNode.getNext()LCollectableNode;
at SoundNode.collect(SoundNode.java:34)
at SoundBranch.collect(SoundBranch.java:52)
```

We'll use a similar strategy as we did last time. The error suggests that the error is in `SoundNode`. Let's recompile that class. Yes, now we do get a compile-time error.

```
SoundNode.java:37: incompatible types
found   : LLNode
required: SoundNode
```

Line 37 is in `collect()`.

```
public Sound collect(){
    SoundNode nextNode;
    if (this.getNext() == null)
    {return mySound;}
    else
    {nextNode = this.getNext(); // ERROR HERE
     return mySound.append(nextNode.collect());}
}
```

Same error here as we have seen before, with the same fix: nextNode = (SoundNode) **this.getNext()**; We try again with a compilation and an execution.

```
> SoundTreeExample ste=new SoundTreeExample()
> ste.setUp()
> ste.play()
NoSuchMethodError: ScaleBranch.getNext () LCollectableNode;
at ScaleBranch.collect (ScaleBranch.java:45)
at SoundBranch.collect (SoundBranch.java:59)
```

The error is in ScaleBranch to be sure, and we can probably guess where the error will be. When we compile ScaleBranch, we get a slightly different error:

```
ScaleBranch.java:46: cannot find symbol
symbol  : method collect()
location: class LLNode
```

What this error says is that our code currently asks an LLNode to collect(). CollectableNode instances know how to collect(), not LLNode instances. Here's what the actual line looks like.

```
public Sound collect(){
    Sound childSound;

    if (children != null)
    {childSound = children.collect().scale(factor);}
    else
    {childSound = new Sound(1);}

    // Collect from my next
    if (this.getNext() != null)
    {Sound nextSound=(this.getNext()).collect(); // ERROR HERE
     childSound = childSound.append(nextSound);}

    return childSound;
}
```

This is actually a similar problem as we had earlier, and we need a similar fix. **this.getNext()** returns an LLNode, and LLNode instances don't know how to collect(). We need a cast. Because of the statement, the cast

looks a little more complex. The working line with a cast looks like this:
 Sound nextSound=((CollectableNode) **this**.getNext()).collect();

Now, finally, we can execute:

```
> SoundTreeExample ste=new SoundTreeExample()
> ste.setUp()
> ste.play()
```

We are rewarded with a gong-ing mess of noise. We have successfully refactored both class hierarchies. We have defined a generalized linked list node, removed the linked list content from both class hierarchies, and made everything work again.

11.2 Making a New Kind of List

Now that we have a generalized LLNode class, we can create new kinds of linked lists easily. We never have to write add() or remove() again. Instead, we simply subclass LLNode. Our subclass should define the data that we want to store in the linked list.

As an example, let's use the Student class that we defined back in Chapter 2 and create a linked list of students. We define a StudentNode class that extends LLNode and stores a Student instance for each node.

Example Java Code: **StudentNode** class

*Program
Example #93*

```
public class StudentNode extends LLNode {

    // Constructor
    public StudentNode(Student pupil){
        super();
        this.setStudent(pupil);
    }

    private Student me;

    public Student getStudent(){
        return me;
    }

    public void setStudent(Student someone){
        me = someone;
    }

    public static void main(String[] args){
        // Make our first student
        Student fred = new Student("Fred");
    }
}
```

```
// Create the students list with the first student
StudentNode students = new StudentNode(fred);

/* Add several more students */
students.add(new StudentNode(new Student("Wilma")));
students.add(new StudentNode(new Student("Barney")));
students.add(new StudentNode(new Student("Betty")));

/* Print out first student */
System.out.println(students.getStudent());
/* Print out third student */
System.out.println(students.getNext().getNext().getStudent());
}
}
```

How it works: The class declaration was obvious: `class StudentNode extends LLNode`. We declare a **private** variable `me` which has a type `Student`. Our constructor for `StudentNode` sets the instance variable to the input `student`, using the setter that we also define for manipulating the `Student` in the node.

The `main()` method simply creates a few nodes (with a few students), then prints them out for testing. However, the simple code above won't work. When we compile it, we get the error:

```
StudentNode.java:33: cannot find symbol
symbol  : method getStudent()
location: class LLNode
```

We have seen an error like this previously. It is saying that the class `LLNode` does not understand `getStudent()`. That's obvious—the question is, “Where are we asking an `LLNode` to `getStudent()`?” The error is in the last line of `main()`:

```
System.out.println(students.getNext().getNext().getStudent());
```

While `students` has type `StudentNode`, we recall that `getNext()` is in `LLNode`. It returns type `LLNode`. The solution, of course, is to cast. We need to cast the result of `students.getNext().getNext()` to `StudentNode`. Here's the re-written line, spaced out over several lines to make the parentheses clear.

```
System.out.println(
    (StudentNode)
        (students.getNext().getNext())
    )
    .getStudent());
```

11.3 The Uses and Characteristics of Arrays, Lists, and Trees

At this point, we can start to summarize some of the characteristics of the various data elements that we have been discussing up until now. We have learned about arrays, linked lists, and trees.

Arrays are more compact than linked lists or trees. Arrays are just element after element after element in memory. There is no wasted space. Linked lists use additional memory to hold references to the next elements. Trees are even worse because they contain both next and children links.

If arrays are more compact, why would you ever want to use linked lists or arrays? Here are several reasons:

- If you don't know the maximum size of the collection of things *a priori*. A linked list or a tree can grow to any size—that's why they are often called *dynamic data structures*.
- If you want to be able to insert and delete into the middle of the collection easily. It is complicated and computationally expensive (in other words, it takes a lot of time) to move lots of elements around in an array.
- You don't need to have fast access to any particular element.

What applications have these characteristics? Lots!

- Order of elements on a slide in PowerPoint. You can SEND TO BACK or BRING FORWARD to change the order of any element in the linked list of elements on the screen.
- Order of video segments when you do non-linear video editing, as in iMovie or Windows Movie Maker. Think about the amount of memory required to store frames in a video—all those pixels, all that sound. For you to be able to drag and drop groups of frames so easily, it cannot be that all the frames are in a big array. It would take much more time to move frames around if that were true.
- Items in a toolbar. Have you ever re-configured your toolbar in an application? You simply drag and drop these icons into the list of icons. As easily as they are rearranged, inserted, and deleted, it is likely that items in a toolbar are stored in a linked list.
- Slides in a PowerPoint presentation. Just as it's too easy to drag segments of video around, it's too easy to drag around sets of slides in the slide organizer in Powerpoint. If slides were stored as an array, you would expect more of a delay as all those pixels and text are moved around, yet it takes not time at all. It's likely that slides are stored in a linked list.

The last point in that list of strengths of linked lists is particularly important: arrays are fast for accessing any element. To access the fifth element of an array is no faster than accessing the 105th. On the other hand, accessing the n th element of a linked list requires $O(n)$ accesses—you have to just walk one element to its next.

How about searching? What if you want to find a particular item in an array? If there is no order to the array or list, it is an $O(n)$ process to find the element—you simply search one piece after another. However, if the array is in a sorted order, you can use a *binary search*. You probably learned about a binary search in your first computing course.

Searching through a dictionary is a good way to depict a binary search. If you want to find a word (“eggplant” for example), you could simply check one page after the next. That’s an $O(n)$ search—you just keep checking element after element for n elements. (On average, the word you are looking for will be halfway through the dictionary.) There’s a smarter way.

- Open up the dictionary halfway through. Is the word you want on those pages? If not, is it before or after the halfway point? We can only answer this question because we know that a dictionary is in sorted order—“A” is at the beginning and “Z” is at the end.
- Take the first or second half of the dictionary, whichever way the desired word lays, and split that half in half. Ask the same questions: On this page, before, or after?

In general, a binary search lets you find the word in $O(\log_2 n)$ tries. Since you split n in half each time, it’s at most $\log_2 n$ tries to get all the way down to the page where the word is. That’s a lot faster than $O(n)$.

You can’t get any real efficiencies in searching a linked list. Even if the linked list is in sorted order, you can’t get to the middle one (for example) any faster than just checking each node one-at-a-time from the beginning. Thus, there’s no good answer for linked lists.

There is some hope for trees, however. If there is no order to the tree, then searching a tree for some element is just as slow as a linked list. There is a way of structuring trees so that they are as fast to search as a binary search on an array. More on that in the next subsection.

Examples of Tree Uses

What are trees good for, if they are less compact than arrays and no faster than linked lists? Trees have two huge benefits:

- Trees represent structure. Whether we consider that structure to be a hierarchy or just clustering, the branching character of a tree allows us to represent something that a linked list doesn’t.
- Branches in a tree can represent something apart from data, like the operations in our sound and image trees. That is a powerful ability to encode both structure and behavior in the same computational entity.

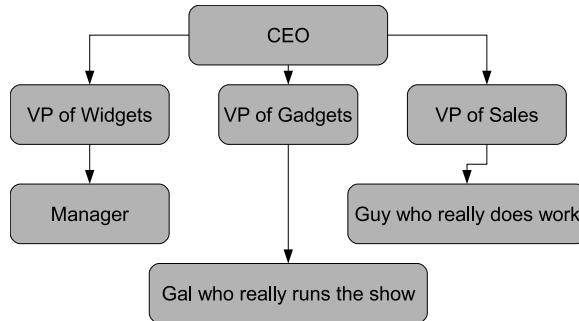


Figure 11.1: An example organization chart

Some of the things that we can represent in a tree include:

- Representing how parts of music assemble to form a whole—we saw that earlier.
- Representing the elements of a scene (*a scene graph*)—again, we saw this earlier, and we know that that’s how professional 3-D animators depict scenes.
- Representing the inheritance relationships among classes (*a class hierarchy*). You may not have thought about that when we were describing our classes. Look again at the descriptions of the `DrawableNode` or `CollectableNode` class hierarchies, and you see a clear tree structure.
- Files and directories on your hard disk—directories are essentially branches, files are leaves.
- Elements in an HTML page. If you know HTML, you know that a whole file splits into `<head>` and `<body>`. A `<head>` can contain (for example), a `<title>`. A `<body>` can have any number of sub-components, such as a `<p>` paragraph which can contain `` bold text or `<i>` italicized text. Thus, a document has two main branches, and sub-branches within those branches. That sounds like a tree.
- An organization chart (“orgchart”) is clearly a tree (Figure 11.1)—a manager has some number of employees, and there are levels (hierarchies) of management.

An example that might be surprising is that a tree can represent an equation (Figure 11.2). Operators go in the branches and operands (numbers, variables) are in the leaves. In fact, this is how equations are often represented internally within a computer. The nice thing about representing equations with trees is that a simple *in-order traversal* of the tree

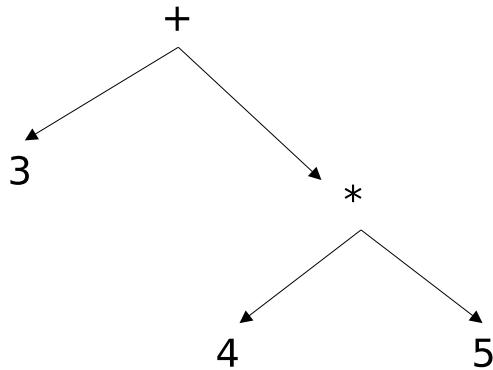
3 + 4 * 5

Figure 11.2: An equation represented as a tree

recreates the equations *and* in the order that the operations should be performed. For example, in Figure 11.2, we would do the multiplication before the addition, which laws of mathematics would require us to do.

The links in a tree can have meanings. In the examples we have seen already, the links from parent node to child node have different meanings. In a class hierarchy, they indicate that the class represented by the parent node is a superclass of the class represented by the child node. In an organization chart, they indicate the parent is the boss and the children are the employees. In an equation tree, the links indicate that the result from the children will be used in applying the operation in the parent.

We can also use trees to represent *meaning*, where the links represent “the child is similar, but a specific case, of the parent.” Consider a tree of words or phrases that represent meaning differences. Figure 11.3 represents a kind of taxonomy, an organization of meanings associated with the concept of “price.” The same word “price” might represent what the customer pays versus what another company might pay.

Where might you use such a tree of meanings? Imagine that you want to create a website that “*crawls*” (visits and gathers information) various shopping or catalog sites to gather prices, so that you can compare the price of the same product at different sites. Maybe one store calls the price the “price” and the other one calls it the “consumer cost,” and you want to avoid the store that talks about “retail price.” A tree of meanings like this can be used to work out how your *web crawler* program should respond to these different human terms.

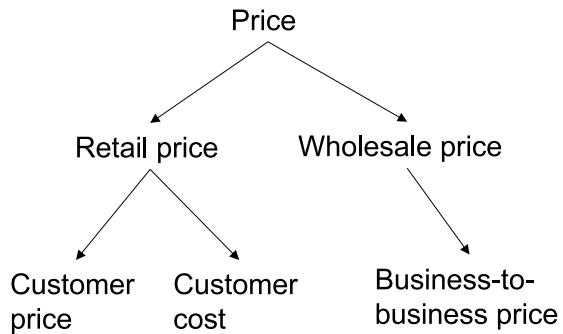


Figure 11.3: A tree of meanings

My cat ate a fat worm.

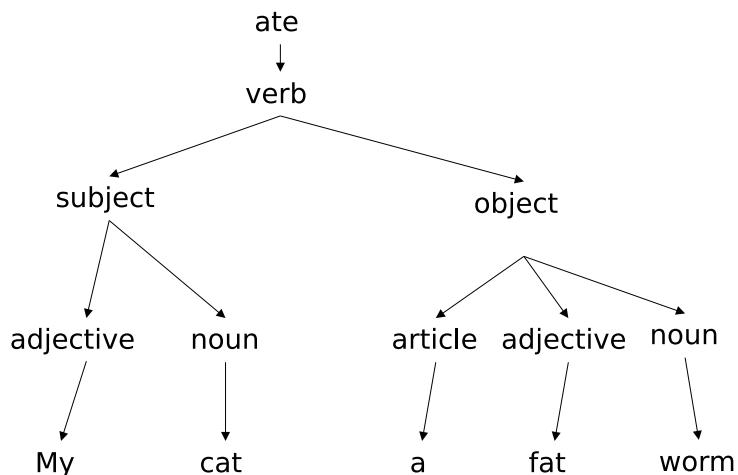


Figure 11.4: A sample sentence diagram

For many of you readers, the notion of representing words in trees may be familiar from *sentence diagrams* that you may have used in your primary schooling (Figure 11.4)¹. A sentence diagram explicitly labels the parts of the sentence (e.g., verb, object, subject, predicate phrase, and so on) with the actual words at the leaves or root of the tree—the boundaries.

A sentence diagram actually has computational advantages, too. Imagine that you work for an organization that is concerned with watching for signs of terrorist activity in the United States from captured text mes-

¹There are a wide variety of sentence diagramming techniques and forms. The advantage of this one is that it is certainly unlike any that anyone actually uses.

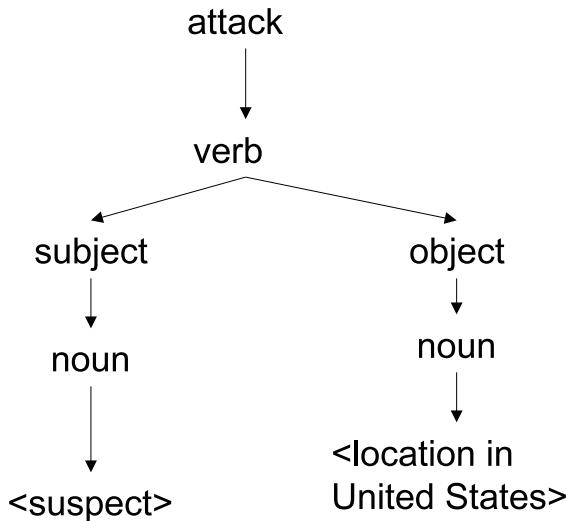


Figure 11.5: A query for a collection of sentence trees

sages. Given the amount of captured text messages today, there are probably megabytes and volumes of messages. Do you just search for a phrase like “is going to attack”? How do you deal with synonyms of the word? What you really want to know is if the phrase refers to a location in the United States (and what that location is) as opposed to “is going to attack my homework this evening.”

This problem is actually solvable with trees of the form that we have been talking about in this section. Imagine that you were able to take those enormous text message collections and figure out the meanings of the words so that you could build trees of all the sentences like Figure 11.4. You might represent what you want to know *also* as a tree, as in Figure 11.5. This query says that you want to know if “someone” (and that <angle brackets> indicates that you want to know whatever lands in this spot) is going to “attack” somewhere in the United States. What if the terrorists use another word for attack? That’s where meaning taxonomies as in Figure 11.3 come in. As we find verbs that are similar to attack, we can check them out in a taxonomy, and if the other pieces fit (e.g., that the target is somewhere in the US), we declare a match.

Doing this kind of matching of pieces of various trees is called a *unification* algorithm. Unification is very powerful. The programming language *Prolog* is actually a language for specifying trees like these, functions as rules on trees, and a powerful form of unification.

Here is one last tree that you use all the time, though you may not have thought of it as a tree. A user interface in a modern computer window is a tree (Figure 11.6). The whole window is usually broken into parts

11.4. BINARY SEARCH TREES: TREES THAT ARE FAST TO SEARCH

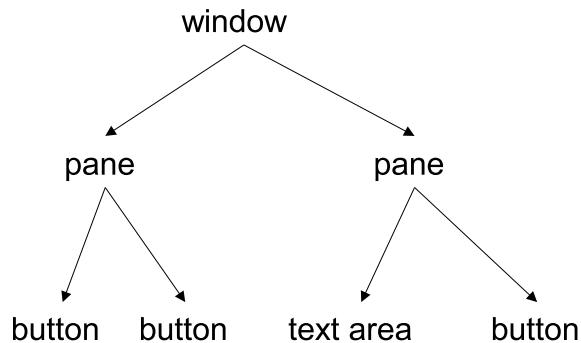


Figure 11.6: A user interface is a tree

we call *panels*. Those panels might have a row of buttons (maybe for a toolbar), and another may hold a text input area and a SAVE button. That is actually a tree. You can actually take the same user interface tree and create a variety of different appearances for the window. Different *layout managers* will render the same user interface tree in different ways—we will see that in the next chapter.

11.4 Binary Search Trees: Trees that are fast to search

As we said in the previous section, we can structure arrays so that we can search them for something particular (some element) in $O(\log_2 n)$. Lists are always $O(n)$ to search. We *can* structure trees in such a way that they are also $O(\log_2 n)$ to search. We construct a *binary search tree* out of a *binary tree*.

A binary tree is so-named because every branch has at most *two* children (Figure 11.7). We explicitly label the links *left* and *right*. What data is in each of these nodes is left completely open—maybe it's some general string, some objects, some images or sounds, whatever.

The really interesting thing about a binary tree, particularly compared to our trees so-far, is that *any* node can be a branch. *Every* node has data associated with it, and a left link, and a right link. That means that any parent node can also be a child node (Figure 11.8). Any branch of a tree, then, is also a tree. It's always the same kinds of objects all the way down. That level of consistency or uniformity can be quite powerful in computation.

Binary trees have a bunch of interesting characteristics that computer scientists have studied over the years. Let's say that you have n nodes in a tree. What is the maximum number of levels or generations in that tree?

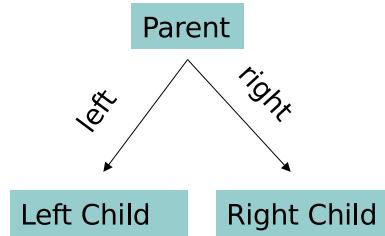


Figure 11.7: Simple binary tree

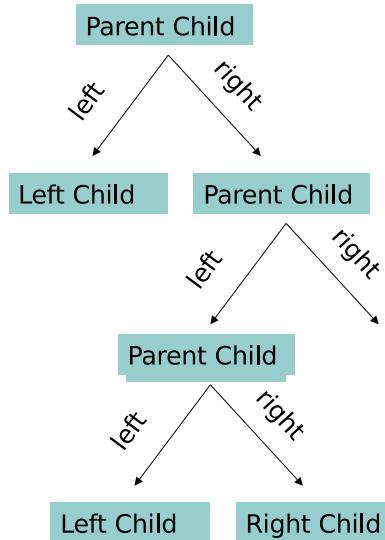


Figure 11.8: More complex binary tree

(Think of the number of levels or generations as the number of nodes you pass through going from the root of the tree to its farthest away leaf.) Each node could only be linked by the left links, or all by the right links — it's just a linked list. Then the number of levels is n . The *minimum* number of levels, though, is $\log_2 n + 1$. Try it—see if you can structure n nodes in fewer levels.

A binary tree is a great example of an ADT. Given what we have seen previously, you can probably imagine much of the definition of a binary tree.

- There is really only one class to define—some kind of `TreeNode`, since we know that all nodes are exactly the same, completely uniform, throughout the tree.
- There must be some way to `getLeft` and `setLeft`, and similarly `getRight`

11.4. BINARY SEARCH TREES: TREES THAT ARE FAST TO SEARCH

and setRight.

- There must be some way to getData and setData—whatever those data might be. One would probably imagine that the constructor for the node class would take data as input.
- Given that we are going to use this tree for storing data that we can find quickly, we might imagine that there will be a insert method that puts a piece of data into the right place in the tree—or maybe we call it an insertInOrder method that thus makes it clear that the data inserts in order. We would then (reflectively) expect a find method that finds the node that has a particular piece of data.

Given the above, one could start writing programs right now that could use a binary tree—even without seeing the implementation. There actually are several different kinds of implementations of trees. For example, there is an implementation of trees that works in a plain old array by computing indices in a crafty way². The implementation doesn't really matter, as long as the above methods are there and work the way that you expect.

Of course, we are going to implement a binary tree as a TreeNode class with object references.

Example Java Code: **TreeNode, a simple binary tree**

*Program
Example #94*

```
public class TreeNode {  
    private String data;  
    private TreeNode left;  
    private TreeNode right;  
  
    public TreeNode(String something){  
        data = something;  
        left = null;  
        right = null;  
    }  
    // Accessors  
    public String getData() {return data;}  
    public void setData(String something){data = something;}  
  
    public TreeNode getLeft() {return left;}  
    public TreeNode getRight() {return right;}  
  
    public void setLeft(TreeNode newleft){left = newleft;}  
    public void setRight(TreeNode newright){right = newright;}}
```

²Basically, have an array of strings (for example), and store the left child of index n at $2n$ and the right child at $2n + 1$.

```
// Recursive traversal of the tree
public String toString()
{return
    "This:" +this.getData() +
    " Left: "+this.getLeft() +
    " Right: "+this.getRight();}
```

Let's test by building a small tree (just two nodes), and see if it prints correctly.

```
> TreeNode node1 = new TreeNode("george");
> node1
This:george Left: null Right: null
> TreeNode node1b = new TreeNode("alvin")
> node1.setLeft(node1b)
> node1
This:george Left: This:alvin Left: null Right: null Right: null
```

That generally is what we would expect. Now, let's use this binary tree to structure data so that we can search and find things very quickly. The structuring rule for a *binary search tree* is very simple: **The left side data is less than the data in the parent node, and the right side data is greater than or equal to the data in the parent node**. Structured like that, a binary search tree is like our dictionary—at each branch, we split the dictionary (tree) in half. At least, that's true if the tree is *well-formed*. We say more about a well-formed, *balanced* binary search tree a bit later.

Given the definition of how we want a binary search tree structured, the algorithm for inserting a new piece of data into a binary search tree is about searching for where the data *should* be, if it were there already, then putting it into place. The method `insertInOrder` is recursive. Because the whole tree is uniform (each parent node is a root of another tree, just a sub-tree of the whole tree's root), we can just pass the buck to the other nodes to do the right thing. The result is that the code is quite short.

How it works: First we ask, “Is the value to insert less than the current node's value?” If that's true, we look to see if there is a left branch. If there is, we ask the left branch to insert the data into place—we make a recursive call to `insertInOrder`. If there is no left branch, we know where to put the data that we want to insert—we insert it on the left. If the value to insert is greater than or equal to the current nodes value (which it must be if it's not less-than), then we check to see if there is something on the right branch. If there is, we ask that right branch node to do the `insertInOrder`. If there isn't, bingo! We put the node on the right.

Since we are working on strings here, we are going to have to compare strings in alphabetical order. There is a method on Strings to do this: `compareTo()`. The method `compareTo` is sent to a string, and takes a second string as input to the method. The method returns a zero if the two

11.4. BINARY SEARCH TREES: TREES THAT ARE FAST TO SEARCH

strings are equal (have the same characters), positive if the input string comes before the string that the method is called on, and negative if the input string comes after. The value has to do with the number of letters (characters) between the two strings.

```
> "abc".compareTo("abc")
0
> "abc".compareTo("aaa")
1
> "abc".compareTo("bbb")
-1
> "bear".compareTo("bear")
0
> "bear".compareTo("beat")
-2
```

Example Java Code: **insertInOrder for a binary search tree**

*Program
Example #95*

```
// Insert in order
public void insert(TreeNode newOne){
    if (this.data.compareTo(newOne.data) > 0){
        if (this.getLeft() == null)
            this.setLeft(newOne);
        else
            this.getLeft().insert(newOne); }
    else {
        if (this.getRight() == null)
            this.setRight(newOne);
        else
            this.getRight().insert(newOne); }
}
```

Let's try it out:

```
> TreeNode node1 = new TreeNode("george");
> TreeNode node2 = new TreeNode("betty");
> TreeNode node3 = new TreeNode("joseph");
> TreeNode node4 = new TreeNode("zach");
> node1
This:george Left: null Right: null
> node1.insert(node2)
> node1
This:george Left: This:betty Left: null Right: null Right: null
> node1.insert(node3)
> node1
```

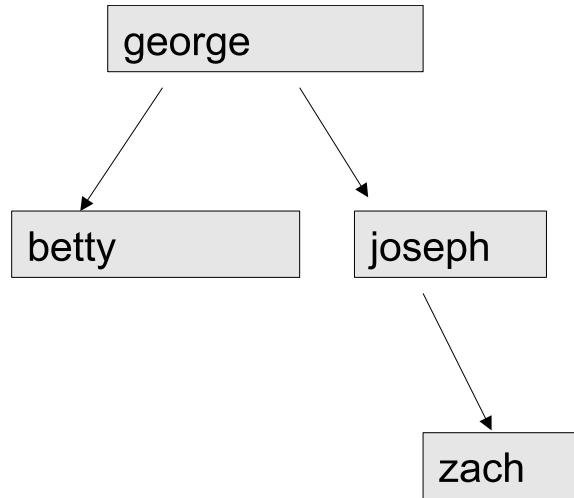


Figure 11.9: Tree formed by the names example

```

This:george Left: This:betty Left: null Right: null Right: This:joseph Left: null Right: null
> node1.insert(node4)
> node1
This:george Left: This:betty Left: null Right: null Right: This:joseph Left: null Right: null
  
```

The printing of node1 at the very end describes the whole tree. It may be hard to understand as such. You might compare it to the drawing of the tree in Figure 11.9.

Now that we have a tree that is formed well for scripting, we can try to write find() for this tree. The basic algorithm depends on the same structuring that we talked about earlier: smaller things down the left side, and bigger things down the right. Again, it's recursive because it's working on the uniform structure of the tree.

How it works: The method find() takes an input string to find. The method asks “Is the input less than ‘me’ (`this.getData()`)?” If so, then stop and return this node. If not, then we want to compare the input string to the current data. If it’s less, we find() down the left branch, and if greater than or equal, we search down the right branch. However, if the left or right node isn’t there (e.g., we want to find() down the left and there is no left node), then we return `null` to indicate that the data was not found.

*Program
Example #96*

Example Java Code: **find, for a binary search tree**

```

public TreeNode find(String someValue){
    if (this.getData().compareTo(someValue) == 0)
        return this;
    else if (this.getData() < someValue)
        return this.left.find(someValue);
    else
        return this.right.find(someValue);
}
  
```

11.4. BINARY SEARCH TREES: TREES THAT ARE FAST TO SEARCH

```

{return this;}
if (this.data.compareTo(someValue) > 0){
    if (this.getLeft() == null)
        {return null;}
    else
        {return this.getLeft().find(someValue);}}
else {
    if (this.getRight() == null)
        {return null;}
    else
        {return this.getRight().find(someValue);}}
}

```

Let's try it on the tree in Figure 11.9:

```

> node1.find("betty")
This:betty Left: null Right: null
> node1.find("mark")
null

```

If tree is well-ordered and well-structured, then searching in the tree should be an $O(\log_2(n))$ process. Each decision splits the amount of data in half. From the top to the bottom of the tree should not take more than $1 + \log_2(n)$ steps, so no more than that many checks.

We call that kind of tree as being *balanced*. A completely valid binary search tree might *not* be balanced—that is, it's *unbalanced*. Figure 11.10 is completely a valid binary search tree: no node has more than two children, left is less than the parent, and right is greater than the parent. This is the same tree as in Figure 11.9. How many searches will it take to find "zach" in this tree using `find()`? Just as many nodes as in the tree, n steps (where $n = 4$ here). Since everything is on the right, we just keep checking one node after the other—this is the same as searching a linked list.

What we want is a way of converting an unbalanced tree into a balanced tree, where roughly half of each sub-tree is in the left branch of the sub-tree and the other half is in the right branch of the sub-tree. We are not going to build this algorithm here. The key idea is that we have to *rotate* our nodes. If we were to rotate the right branch off "betty" in Figure 11.10, we would "rotate up" the node "joseph," moving "george" to the left and leaving "zach" on the right. That part isn't hard to understand or even implement. Completely balancing the tree involves rotating left and right to go from Figure 11.10 to Figure 11.9—for example, notice that the node "george" has to become the root of the tree, so another kind of rotation occurs from Figure 11.11 from Figure 11.10.

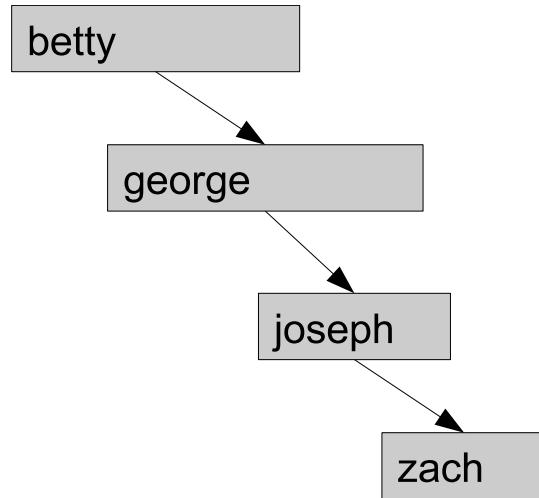


Figure 11.10: An unbalanced form of the last binary search tree

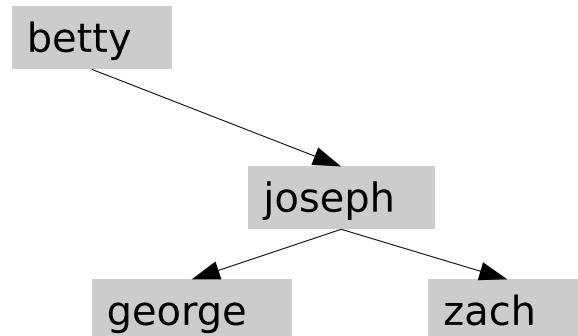


Figure 11.11: Rotating the right branch off “betty”

Traversals of Trees

We can take a binary search tree and print it out in alphabetical order. We use a process called *inorder traversal*—traverse the tree in (alphabetical) order. It's really pretty simple, if we are willing to do it recursively. From each node, the left side has to come before the current (**this**) node, and then the right side should be printed. That would be the order because left is less than **this** data, and right is greater than **this**.

*Program
Example #97*

Example Java Code: **traverse, a binary tree in-order**

11.4. BINARY SEARCH TREES: TREES THAT ARE FAST TO SEARCH

```
//In-Order Traversal
public String traverse(){
    String returnValue = "";
    // Visit left
    if (this.getLeft() != null)
        {returnValue += " "+this.getLeft().traverse();}
    // Visit me
    returnValue += " "+this.getData();
    // Visit right
    if (this.getRight() != null)
        {returnValue += " "+this.getRight().traverse();}
    return returnValue;
}
```

Let's try it on our binary tree from Figure 11.9:

```
> node1.traverse()
" betty george joseph zach"
```

Do we have to do it that way? Of course not! We could visit ourselves, then the left and then right. That's called *pre-order traversal*. We could also visit the left and then the right, and then ourselves. That's called *post-order traversal*. We can think about other orderings as well.

Doing an in-order traversal of a tree makes sense. One can imagine wanting a list of everything in a tree. When we are working with trees of names, doing a pre-order or post-order traversal doesn't make much sense. However, it does make sense when there are certain other things in the tree, like the equation tree we saw earlier (Figure 11.2).

Here's an example of the power of different traversals on this kind of tree. If you do an in-order traversal of the equation tree Figure 11.12, you get $(3 * 4) + (x * y)$. That is fine and useful—that is an equation that makes sense. If you do a pre-order traversal of the same tree, you get $34 * xy * +$. That may look like gibberish *unless* you have ever used a *reverse Polish notation (RPN)* calculator, like some of Hewlett-Packard's popular calculators. The equation $34 * xy * +$ means “Push a 3 on the *stack*, then a 4, multiply two things on the stack and push the result back on the stack. Push the value of x on the stack, then y , multiply the two things on the stack and push the result back. Finally, add the two things on the stack and push the result back on.” This turns out to be an efficient way of doing the same computation. A pre-order traversal of the same equation tree gives you the RPN form of the equation.

Trees can do anything

We now know a bunch of things that trees can do well:

- It is easy to do useful things with them.

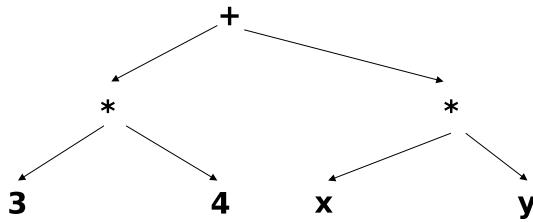


Figure 11.12: An equation tree for different kinds of traversals

- Searching is quick.
- We can use them to evaluate equations.
- Operations on trees are easily written, small, and recursive.
- We know interesting facts about them.

Binary trees can actually do just about anything. A binary tree can represent an n-ary tree, like the ones we used in earlier chapters. A binary tree can even be a list.

Imagine that we want a list (an ordered group of items) where we can add things to the front or end of the list. We can implement that with a linked list, of course. We can also implement this same structure with a binary tree. We make the structuring assumption that *earlier* items are to the left, and *later* items go to the right—a similar assumption to how we made a binary search tree. Here are the methods for doing that, `addFirst()` and `addLast()`.

Program
Example #98

Example Java Code: **addFirst and addLast, treating a tree as a list**

```

public void addFirst(TreeNode newOne){
    if (this.getLeft() == null)
        {this.setLeft(newOne);}
    else
        {this.getLeft().addFirst(newOne);}
}

public void addLast(TreeNode newOne){
    if (this.getRight() == null)
        {this.setRight(newOne);}
    else
        {this.getRight().addLast(newOne);}
}
  
```

* * *

11.4. BINARY SEARCH TREES: TREES THAT ARE FAST TO SEARCH

```
> TreeNode node1 = new TreeNode("the")
> node1.addFirst(new TreeNode("George of"))
> node1.addLast(new TreeNode("jungle"))
> node1.traverse()
" George of the jungle"
```

While binary trees are simple, they are actually quite powerful. There is great power in having a uniform structure—it allows us to write code in short form that actually works. By simply choosing the meaning of our left and right links, we can represent a wide variety of things. This chapter explains some of these.

Ex. 20 — How would you rewrite VBranch and HBranch so that there is less duplicated code between them? Could you have a general LayoutBranch that calls some method for changing the (x, y) positions, then subclass that to create VBranch and HBranch? Refactor these methods so that there is less duplicated code.

Ex. 21 — Use the same approach for rewriting SoundBranch and ScaleBranch so that there is less duplicated code.

Ex. 22 — (Advanced) Investigate the Java notion of a **interface**. Can you come up with a LLBranch interface that all our branches (Branch, VBranch, HBranch, SoundBranch, and ScaleBranch) might implement so as to reduce the duplication of code? You might also investigate the *design pattern* called “Visitor.” Can you use that to make tree traversals more common?

Ex. 23 — Trace out insertInOrder and how it walks through an example tree in order to insert a new piece of data.

Ex. 24 — Trace out the right calls to insertInOrder for a set of 8 words that result (a) in an unbalanced tree with all words down the right branch, (b) in an unbalanced tree with all words down the left branch, and (c) a balanced tree.

Ex. 25 — (Advanced) Write the method balance() for a generic binary search tree that produces a balanced tree.

Ex. 26 — Write addFirst() and addLast() for LLNode so that all our linked lists can add to the front or end.

12 Circular Linked Lists and Graphs: Lists and Trees That Loop

Chapter Learning Objectives

Lists can loop—a latter node can have as its next point to an earlier node. We use a *circular linked list* sometimes to create circular lists, as when you are representing the cells in an animation, such as in the Nintendo video game *Mario Brothers*.

Trees can also have loops, while children of different subtrees can be linked together. We call those kinds of trees *graphs*.

The computer science goals for this chapter are:

- To create linked lists that have loops in them, and how to avoid the dangerous parts of it.
- To explore the use of graphs, and how to traverse them.

The media learning goals for this chapter are:

- To create a looping cell animation, as used in older video games.

12.1 Making Cell Animation with Circular Linked Lists

The older reader may recall Nintendo's *Super Mario Brothers* (Figure 12.1). The Mario Brothers characters seemed to run and jump under the control of the user in exploring a virtual world. As Mario and Luigi "ran," their arms and legs seemed to move as they moved. The style of animation being used in those style games is called *cell animation*.

In a cell animation, characters are represented by a series of still images. In one image, the right leg might be raised and in front of the body. In the next image, the right leg might be on the ground. In a following image, the left leg might be raised ahead of the body, and so on. By rapidly showing different of these images, in sequence, the illusion of moving body parts is created. By moving where the images are displayed, the illusion is

CHAPTER 12. CIRCULAR LINKED LISTS AND GRAPHS: LISTS AND
294 TREES THAT LOOP



Figure 12.1: Scenes from Super Mario Brothers



gal1-rightface.jpg



gal1-right2.jpg



gal1-right1.jpg

Figure 12.2: Three images to be used in a cell animation

created that the moving body parts are actually driving the character. In the end, the character seems to run.

Let's create a version of cell animation. The wildebeests in *The Lion King* and the villagers in *The Hunchback of Notre Dame* are not examples of cell animation. In those cases, the body parts did move and were controlled at a fine level of detail. However, a cell animation *could* have been used to represent the characters whose positions are specified by a simulation. It is a particularly easy mechanism to be used in your own animations. Cell animations were invented for use on lower-powered processors (like the early Nintendo video games), so if you were to do an animation on a computer with little power (e.g., perhaps a cellphone), then cell animation would be a reasonable choice.

In your MediaSources folder on the CD, you will find some pictures of one of our daughter's dolls. These pictures are positioned (as best we could) to represent different positions in walking (Figure 12.2). We took the pictures against a blue background so that they could be used with chromakey.

We can arrange these images in a sequence to give the appearance of

12.1. MAKING CELL ANIMATION WITH CIRCULAR LINKED LIST 295



Figure 12.3: A sequence of images arranged to give the appearance of walking

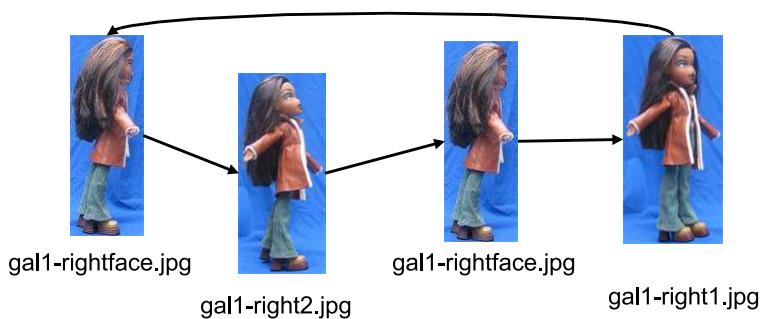


Figure 12.4: A circular linked list of images

walking, were they to be displayed one right after the other (Figure 12.3). These images are particularly stiff and give more of the impression of Frankenstein’s monster walking—hopefully the point is made. Imagine these pictures in a flipbook, so that if you flipped the pages quickly, the images would seem to move.

One structure for arranging these images in the right order for display in a cell animation is a *circular linked list*. A circular linked list has at least one node (typically, the last node) whose next refers to a node earlier in the list (often, the first node). By arranging a series of picture nodes in a circular linked list gives us a simple way of defining the sequence of images for a cell animation (Figure 12.4).

A circular linked list is really useful for modeling lots of real things in the world. There are many things that contain loops: electrical circuits, pipe systems, maps of roads (e.g., there’s always more than one way to go between two spots). The tricky aspect of a circular linked list is *never* to try to traverse the circular linked list to “the end,” that is, where next is **null**. In a circular linked list, there is no **null** references.

Here’s what we’ll execute to create a walking doll (Figure 12.5):

CHAPTER 12. CIRCULAR LINKED LISTS AND GRAPHS: LISTS AND
296 TREES THAT LOOP



Figure 12.5: Frames of the walking doll

```
> WalkingDoll gal = new WalkingDoll(); gal.setUp();
> gal.steps(10);
```

How it works: To make our WalkingDoll class work, we keep track of a current node in the circular linked list and where the (x, y) position should be. To take a step, we display the current node's picture, then set the current equal to its next and update the position instance variables. Now, when the next step comes along, we will be displaying a new picture.

Program
Example #99

Example Java Code: **WalkingDoll**

```
public class WalkingDoll {

    /**
     * Which character node position are we at?
     */
    public CharNode current;

    /**
     * Starting position for new walking.
     */
    public CharNode start;

    /**
     * Position for the character
     */
    public int x, y;

    public int getX() {return x;}
    public int getY() {return y;}
    public void setLoc(int nux, int nuy){x=nux; y=nuy;}

    /**
     * FrameSequence for the display
     */
    FrameSequence frames;
}
```

12.1. MAKING CELL ANIMATION WITH CIRCULAR LINKED LIST 297

```
* We'll do the list setup in the constructor
*/
public WalkingDoll(){
    Picture p = null; // For loading up images

    p = new Picture(FileChooser.getMediaPath("gal1-rightface.jpg"));
    start = new CharNode(p);
    p = new Picture(FileChooser.getMediaPath("gal1-right2.jpg"));
    CharNode rightfoot = new CharNode(p);
    p = new Picture(FileChooser.getMediaPath("gal1-rightface.jpg"));
    CharNode center = new CharNode(p);
    p = new Picture(FileChooser.getMediaPath("gal1-right1.jpg"));
    CharNode leftfoot = new CharNode(p);
    start.setNext(rightfoot); rightfoot.setNext(center);
    center.setNext(leftfoot);
    // Now the scary one
    leftfoot.setNext(start);

    frames = new FrameSequence("D:/Temp/");
}

/**
 * Setup to display walking left to right
 */
public void setUp(){
    x = 0; // Left side
    y = 300; // 300 pixels down
    frames.show();
    this.start();
}
/**
 * Start a walking sequence
 */
public void start() {
    current = start;
    this.draw();
}

/**
 * Draw the current character
 */
public void draw() {
    Picture bg = new Picture(400,400);
    Turtle pen = new Turtle(bg);
    pen.setPenDown(false); pen.moveTo(x,y);
    current.drawWith(pen);
    frames.addFrame(bg);
}
/**
 * Draw the next step

```

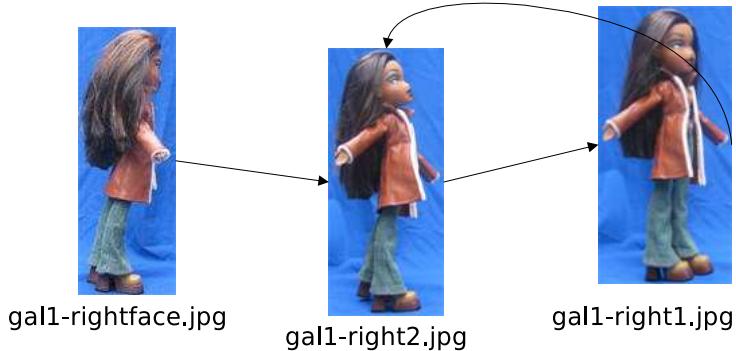


Figure 12.6: A partial circular linked list

```
 */
public void step(){
    current = (CharNode) current.getNext();
    x=x+10; // We'll try this
    this.draw();
}

/**
 * Draw a few steps
 */
public void steps(int num){
    for (int i=0; i < num; i++) {this.step();}
}
```

While this works, the walking doll looks terrible. Like we said, it looks like Frankenstein's monster. We could fix this problem with better images. We might also remove the image where the legs come together between steps. We can model this by a circular linked list where the loop doesn't go all the way back to the beginning (Figure 12.6). The list starts with a single, standing image, then loops just on the movement of the legs.

12.2 Generalizing a Circular Linked List

Let's create a class like `LLNode` that makes it easy to create circular linked lists. The trick is never to look for `null`. This simple version simply refuses to do the things that would normally require a traversal to the `null` at the end of the list—no last and no remove.

```
> CharNode ch = new CharNode(new Picture(20,20));
> ch.last()
Don't try to find last() from a circular list!
```

```
CharNode with picture: Picture, filename null height 20 width 20
> ch.remove(new CharNode(new Picture(20,20)))
Very dangerous to try to remove a node from this list!
```

Program
Example Java Code: CharNode, a class for representing characters *Example #100*
in cell animations

```
/*
 * CharNode is a class representing a drawn picture
 * that is one in a sequence of Pictures to
 * use for a given character. Don't ever try to traverse this one!
 */
public class CharNode extends LLNode {
    /**
     * The picture I'm associated with
     */
    public Picture myPict;

    /*
     * Make me with this picture
     * @param pict the Picture I'm associated with
     */
    public CharNode(Picture pict){
        super(); // Call superclass constructor
        myPict = pict;
    }
    /**
     * Don't try to remove() from a circular list!
     */
    public void remove(LLNode node){
        System.out.println("Very dangerous to try to remove a node from this list!");
    }

    /**
     * Don't try to get the last() from a circular list!
     */
    public LLNode last(){
        System.out.println("Don't try to find last() from a circular list!");
        return this;
    }

    /**
     * Method to return a string with information
     * about this node. A NON-recursive one.
     */
    public String toString()
```

CHAPTER 12. CIRCULAR LINKED LISTS AND GRAPHS: LISTS AND TREES THAT LOOP

300

```
{  
    return "CharNode with picture: "+myPict;  
}  
  
/*  
 * Use the given turtle to draw oneself  
 * @param pen the Turtle to draw with  
 **/  
public void drawWith(Turtle pen){  
    // Assume that we're at the lower-left corner  
    pen.setHeading(0); pen.forward(myPict.getHeight());  
    Picture bg = pen.getPicture();  
    myPict.bluescreen(bg,pen.getXPos(),pen.getYPos());  
}  
}
```

How it works: While CharNode works, it works by taking the easy way out. The methods `last()`, `remove()`, and `toString` in `LLNode` involve traversing the whole list until `next` is `null`. CharNode simply prevents those methods from executing, by overriding them and displaying error messages.

There are better ways of implementing circular linked lists. How can we traverse a circular linked list without going on infinitely, looking for a end to a circle? There are several ways to do it—here are a couple:

- We could add another instance variable, a boolean named `visited`. As we visit a node, we mark it `true`. When we traverse the list, we mark each node that we print or whatever by setting `visited` to `true`. Then, rather than looking for `next` equal to `null`, we look for `visited` equal to `true`. When that's true, we've looped around and can stop. We need a reset method, too, that sets all `visited` flags to `false` (and keeps going until it finds a `visited` flag that is already `false`).
- We could have *two* `next` links in each node. One points to the next node created, and the other points to the next node to be traversed when displaying the cells. When we want to traverse all nodes (or `add()` or `remove()`), we use the first `next`. When we want to draw, we use the second `next`.

12.3 Graphs: Trees with Loops

In the previous sections, we have shown that there are some uses for linked lists with loops in them. How about trees? Is it useful to have nodes that link to previous nodes, e.g., children that point to their grandparents in the tree? With that kind of data structure, we can model all kinds of interest things that appear in the world.

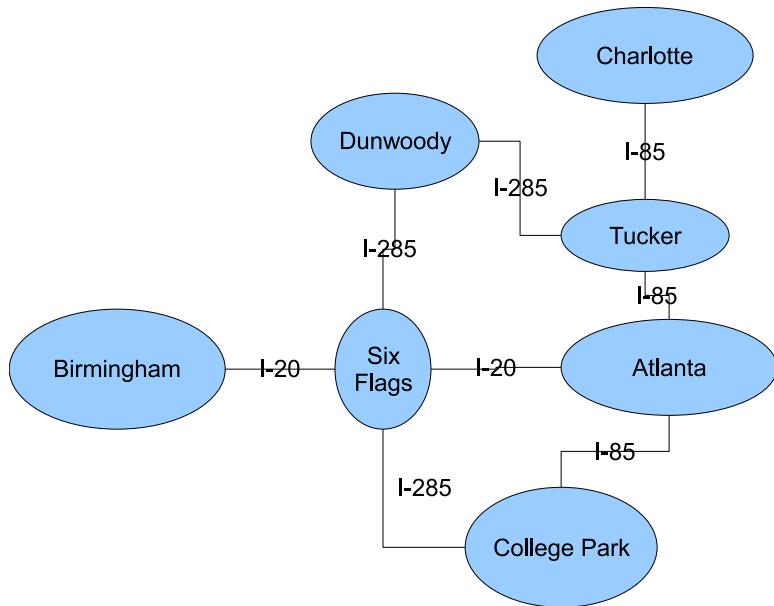


Figure 12.7: A map as a graph

We call these general structures where loops (or *cycles*) are allowed, *graphs*. A graph is a series of points or *vertices* connected with lines or *edges*. In some graphs, there is a directionality to the line or edge—node1 points to node3, but not vice-versa. We call those graphs a *directed graph*. Without that directionality, we call the graph *undirected*.

From this perspective, a tree is an unusual kind of graph. It's a graph without cycles. A tree is a kind of an *acyclic graph*.

Graphs are useful to model structures in the real world that have loops in them.

- Think about modeling a human circulatory system. There are clearly cycles in this graph—your blood doesn't flow to the end of your body and then out your fingers and toes. Your blood cycles back to the heart.
- A subway system is another example of a graph that has cycles. Each station is a vertex. Subway lines between stations are edges. Clearly there are cycles—on most subway systems, it's possible to go out one way and come back another way, or to find more than one way to get from one place to another.
- A map is another kind of graph (Figure 12.7). Cities (or even intersections or exits, depending on the detail you want) are vertices, and roads are edges. A map is a good example of a graph that has *costs*

CHAPTER 12. CIRCULAR LINKED LISTS AND GRAPHS: LISTS AND TREES THAT LOOP
302

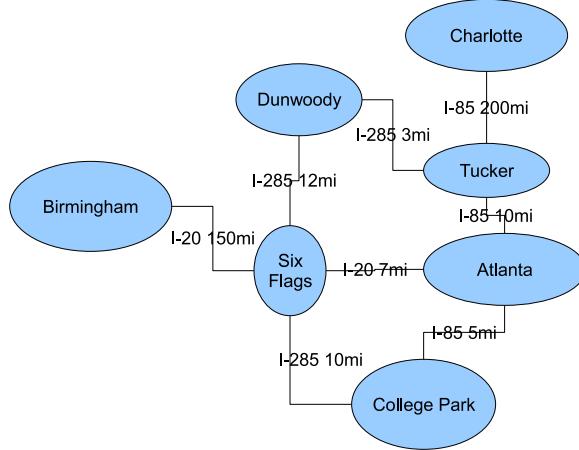


Figure 12.8: Apply weights to a graph—distances on a map

associated with edges. For many uses of maps, you care about the distance between any two vertices—that can be the cost of the edge (Figure 12.8).

Traversing graphs (doing something to every node) is particularly hard to do. As you can imagine from these examples, each node or vertex can actually be associated with a number of edges, not just one (as in a linked list) or two (as in binary tree). This means that a traversal has to make sure that every vertex is included, which may involve traveling down every edge.

The strategies for traversing graphs are similar to the strategies for traversing circular linked lists. We can add a visited flag to tell us whether or not we've visited a vertex. Then we can travel down every edge until we are sure that every node has visited equal to `true`. We can also keep a separate linked list of all the nodes so that we can check each node, regardless of how it's connected up into the graph.

A powerful tool for traversing a graph is to create a *spanning tree*. A spanning tree is the same graph (i.e., includes all the same vertices) without cycles. Once you have a spanning tree, you can visit all nodes using standard tree traversal techniques—you don't have to deal with the cycles.

One approach to computing a spanning tree is to use a *greedy algorithm*. A greedy algorithm, when having to make a choice (say, between which edge to follow from a given vertex) takes the shortest or easiest paths. That's being “greedy.” It turns out that a greedy algorithm for creating a spanning tree on a graph actually results in a *minimal spanning tree*—a spanning tree that covers the least ground (has the lowest total cost).

Let's imagine that something bad has happened in the United States—

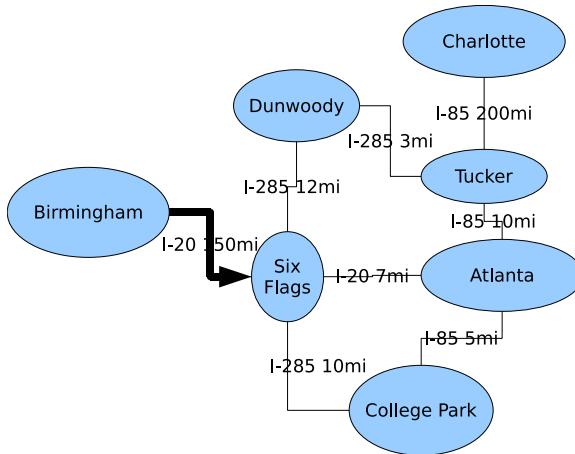


Figure 12.9: Traversing a graph to create a spanning tree

an invasion or an outbreak of an epidemic disease. In Birmingham on our map (Figure 12.7, there are troops or a vaccine. How do we visit all the cities on the map as quickly as possible, traveling the shortest path as possible. What we want is a minimal spanning tree.

The first step is obvious. We have to travel from Birmingham to Six Flags (Figure 12.9). From Six Flags we have several choices. We add Atlanta because it's the cheapest (lowest cost, shortest) path out of Six Flags (Figure 12.10). From Atlanta, we move to College Park as the next cheapest edge (Figure 12.11). We then have a problem. The only untraversed link out of College Park goes to Six Flags, where we've already been. We now *backtrack*, revisiting an earlier node to see if there are more paths from there. We backtrack to Atlanta where we do have a next-cheapest link (after the one to College Park) up to Tucker (Figure 12.12). From Tucker, we add the link to Dunwoody since it's obviously the cheapest (Figure 12.13). Dunwoody has no untraversed nodes (since we can't add Six Flags back in), so we backtrack again to Tucker, and then finish our minimal spanning tree at Charlotte (Figure 12.14).

You might be wondering how the algorithm figures out to which node to backtrack. It's pretty easy—it's another use for *stacks*. Each node, as it is visited, is pushed onto the stack. If you can't find another edge to follow from the node you're currently at, you pop off the top node from the stack, then see if there are more edges that need exploring from that node. You keep going until the stack is empty. The stack being empty is actually when you know that the algorithm is done.

Ex. 27 — Implement the circular linked list where the loop doesn't go all the way back to the front of the loop.

CHAPTER 12. CIRCULAR LINKED LISTS AND GRAPHS: LISTS AND TREES THAT LOOP
304

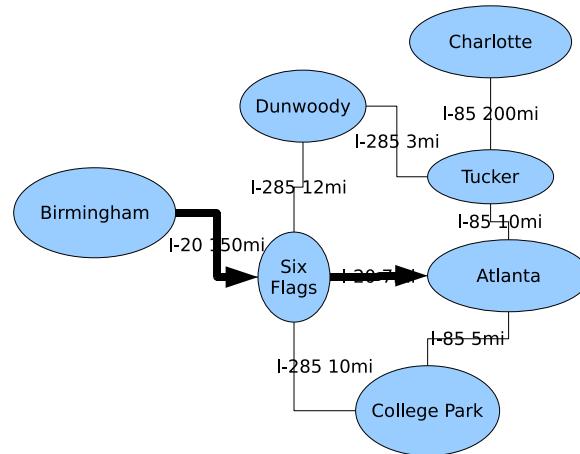


Figure 12.10: Choosing the cheapest path out of Six Flags

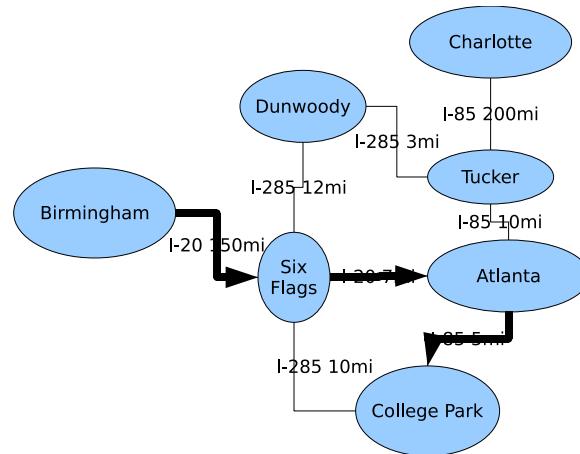


Figure 12.11: Going to College Park

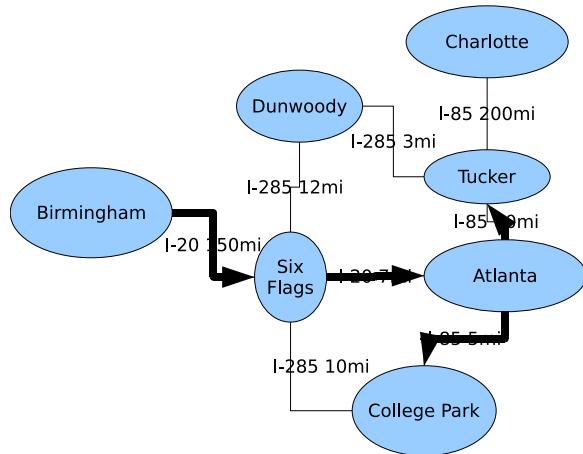


Figure 12.12: Backtracking to avoid re-visiting Six Flags

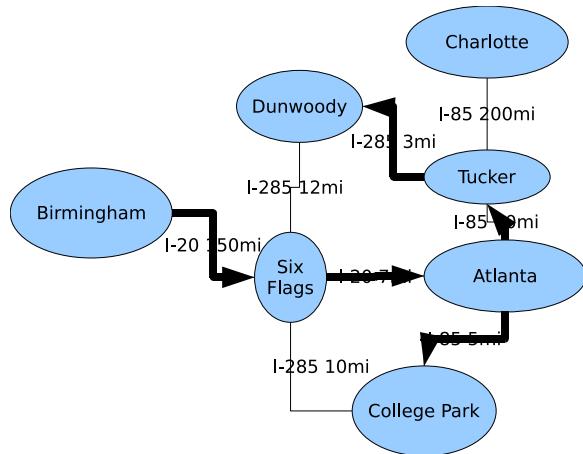


Figure 12.13: Adding Dunwoody, the obviously cheaper path

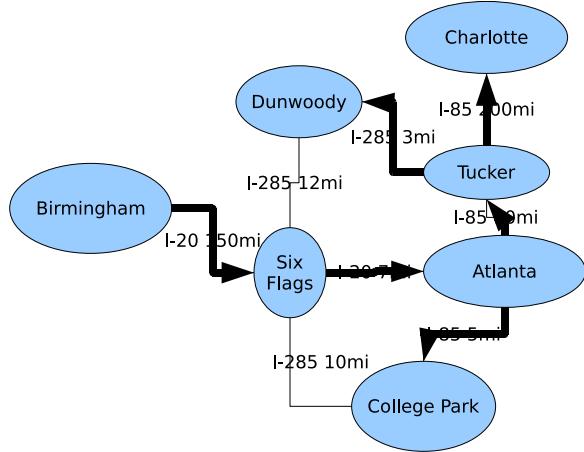


Figure 12.14: Finishing up in Charlotte

Ex. 28 — Implement `walkBackwards` for `WalkingDoll` class, so that the *x* position goes right to left, and the order of images swaps.

Ex. 29 — Implement `walkToLeft` for `WalkingDoll` class, where the images are flipped and *x* position goes right-to-left.

Ex. 30 — Implement a better `CharacterNode` class that uses one of the strategies described in this method to allow us to traverse a circular linked list without simply blocking dangerous methods.

13 User Interface Structures

We are all familiar with the basic pieces of a *graphical user interface (GUI)*: windows, menus, lists, buttons, scrollbars, and the like. As programmers, we can see that these elements are actually constructed using the lists and trees that we've seen in previous chapters. A window contains *panes* that in turn contain components such as *buttons* and *lists*. It's all a hierarchy, as might be represented by a tree. Different *layout managers* are essentially rendering the interface component tree via different traversals.

Part IV

Simulations: Problem Solving with Data Structures

14 Using an Existing Simulation Package

We may put a chapter here that uses Greenfoot or Gridworld. The idea is to *use* a simulation package, before exploring how one is built and how to pull an animation out of one.

15 Introducing UML and Continuous Simulations

Chapter Learning Objectives

We're now starting on the third major theme of this book. The first two were programming media in Java and structuring media using dynamic structures (e.g., linked lists and trees). The third theme is simulations, and here's where we use all of the above to create our villagers and wildebeests.

The problem being addressed in this chapter is how to model dynamic situations, and then, how to simulate those models.

The computer science goals for this chapter are:

- To be able to use the basic terminology of simulations: Discrete event vs. continuous, resources, and queues.
- To describe linked lists as a *head* and a *tail* (or *rest*).
- To learn generalization and aggregation as two mechanisms for modelling with objects.
- To use UML class diagrams for describing the class structure of increasingly sophisticated object models.
- To implement a simple predator-prey simulation.
- To write numeric data to a file for later manipulation.

The media learning goals for this chapter are:

- To describe (and modify) behavior of agents in order to create different graphical simulations (like the Wildebeests and Villagers).
- To use spreadsheets (like Excel) for analyzing the results of graphical simulations.

15.1 Introducing Simulations

A simulation is a representation of a system of objects in a real or fantasy world. The purpose of creating a computer simula-

tion is to provide a framework in which to understand the simulated situation, for example, to understand the behavior of a waiting line, the workload of clerks, or the timeliness of service to customers. A computer simulation makes it possible to collect statistics about these situations, and to test out new ideas about their organization.

The above quote is by Adele Goldberg and Dave Robson from their 1989 book in which they introduced the programming language *Smalltalk* to the world [Goldberg and Robson, 1989]. Smalltalk is important for being the first language explicitly called “object-oriented,” and it was the language in which the desktop user interface (overlapping windows, icons, menus, and a mouse pointer) was invented. And Smalltalk was invented, in part, in order to create *simulations*.

Simulations are representations of the world (models) that are executed (made to behave like things in the world). The idea of objects in Smalltalk were based on a programming language called *Simula*, which was entirely invented to build simulations. Object-oriented programming makes it easier to build simulations, because objects were designed to model real-world objects.

- In the real world, things *know* stuff and they *know how to do* stuff. We don’t mean to anthropomorphize the world, but there is a sense in which real world objects *know* and *know how*. Blood cells *know* the oxygen that they carry, and they *know how* to pass it through to other cells through permeable membranes. Students know the courses that they want, and Registrars know the course catalog.
- Objects get things done by *asking* each other to do things, not by demanding or controlling other things. The important point is that there is an *interface* between objects that defines how they interact with each other. Blood cells don’t force their oxygen into other cells. Students register for classes by requesting a seat from a registrar—it’s not often that a student gets away with registering by placing themselves on a class roll.
- Objects decide what they share and what they don’t share. The Registrar doesn’t know what a student wants to enroll for, and the student won’t get the class she wants until the desired course is shared with the Registrar.

Object-oriented programming was invented to make simulations easier, but not *just* to build simulations. Alan Kay, who was one of the key thinkers behind Smalltalk and object-oriented programming, had the insight that simulations were a great way to think about *all* kinds of programs. A course registration system is actually a simulation of a model of how a campus works. A spreadsheet is a simulation of the physical paper books in which accountants would do their totals and account tracking.

There are two main kinds of simulations. Continuous simulations represent every moment of the simulated world. Most video games can be thought of as continuous simulations. Weather simulations and simulations of nuclear blasts tend to be continuous because you have to track everything at every moment. Discrete event simulations, on the other hand, do not represent *every* moment of time in a simulation.

Discrete event simulations only simulate the moments when something interesting happens. Discrete event simulations are often the most useful in professional situations. If you want to use a simulation of a factory floor, in order to determine the optimal number of machines and the layout of those machines, then you really don't care about simulating the product when it's in the stamping machine, cutter, or polisher. You only care about noting when material enters and leaves those machines—and having some way to measure how much the material was *probably* in the machine for. Similarly, if you wanted to simulate Napoleon's march to Russia (maybe to explore what would have happened if they'd taken a different route, or if the weather was 10 degrees warmer), you care about how many people marched each day, and consider some notion of how many might succumb to the cold each day. But you really don't need to simulate every foot-dragging, miserable moment—just the ones that really matter.

The real trick of a discrete event simulation, then, is to figure out when you should simulate—when something interesting should happen, so that you can jump right to those moments. We're going to find that there are several important parts of a discrete event simulation that will enable us to do that. For example, we will have an *event queue* that will keep track of what are the important points that we know about so-far, and when are they supposed to happen.

Discrete event simulations (and sometimes continuous simulations) tend to involve *resources*. Resources are what the active, working beings (or *agents*) in the simulation strive for. A resource might be a book in a library, or a teller in a bank, or a car at a rental agency. We say that some resources are *fixed*—there's only so much of it (like cars in a rental car), and no more is created even if more is needed. Other resources are *produced* and *consumed*, like jelly beans or chips (just keep crunching, we'll make more).

We can also think of resources as points of coordination in a simulation. Imagine that you are simulating a hospital where both doctors and patients are agents being simulated. You want to simulate that, during some procedure (say, an operation), both the doctor and patient have to be at the same place and can't do anything else until the procedure is done. In that case, the operating table might be the *coordinated resource*, and when both the doctor and patient access that resource, they're both stuck until done.

If an agent can't get the resource it wants when it wants it, we say that the agent enters a *queue*. In the United Kingdom, people know that word well—in the United States, we simply call it “a line” and being in a queue is

“getting in line.” A queue is an ordered collection (that is, there is a first, and a second, and eventually, a last) where the first one into the line is the first one that comes out of the line (and the second one in is the second one out, and so on). It’s a basic notion of “fairness.” We sometimes call a queue a *FIFO list*—a list of items that are first-in-first-out. If an agent can’t get the resource that she wants, she enters a queue waiting for more resource to be produced or for a resource to be returned by some other agent (if it’s a fixed resource).

The interesting question of any simulation is *Is the model right?* Do the agents interact in the way that describes the real world correctly? Do the agents request the right resources in the right way? And then, how do we *implement* those models? To get started, let’s build one model and simulate that model.

15.2 Our First Model and Simulation: Wolves and Deer

We are going to explore a few different kinds of continuous simulations in this chapter. We’ll be using our `Turtle` class to represent individuals in our simulated worlds. The first simulation that we’re going to build is a simulation of wolves chasing and eating deer (Figure 15.1). Wolves and deer is an instance of a common form of continuous simulation called *predator and prey* simulations.

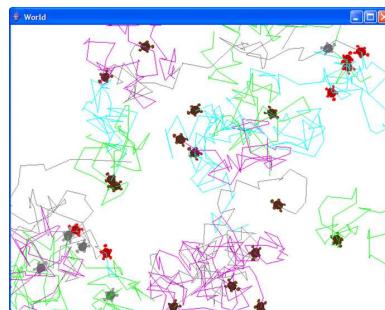


Figure 15.1: An execution of our wolves and deer simulation

The name of this class is `WolfDeerSimulation`. We can start an execution like this:

```
Welcome to DrJava. & WolfDeerSimulation wds = new WolfDeer-
Simulation() & wds.run() && Timestep: 0 Wolves left: 5 Deer
left: 20 && Timestep: 1 Wolves left: 5 Deer left: 20 ¡SIGH!&
A deer died... && Timestep: 2 Wolves left: 5 Deer left: 19 &&
Timestep: 3 Wolves left: 5 Deer left: 19 ¡SIGH!& A deer died...
&& Timestep: 4 Wolves left: 5 Deer left: 18
```

What we see is Figure 15.1. Wolves (in gray) move around and (occasionally) catch deer (in brown), at which point the deer turn red to indicate their death (depicted above with a <SIGH>!). This is a very simple model, but we're going to grow it further in the book.

WolfDeerSimulation is a continuous simulation. Each moment in time is simulated. There are no resources in this simulation. It is an example of a predator-prey simulation, which is a common real world (ecological) situation. In these kinds of simulations, there are parameters (variables or rules) to change to explore under what conditions predators and prey survive and in what numbers. You can see in this example that we are showing how many wolves and deer survive at each *time step*—that is, after each moment in the simulation's notion of time. (It's up to the modeler to decide if a moment stands for a nanosecond or a hundred years.)

Modelling the Wolves and Deer

Simulations will require many more classes than the past projects that we have done. Figure 15.2 describes the relationships in the Wolves and Deer simulation.

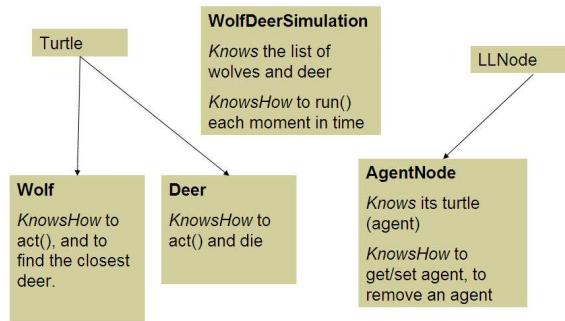


Figure 15.2: The class relationships in the Wolves and Deer simulation

- We are going to use our **Turtle** class to model the wolves and deer. The class **Wolf** and the class **Deer** are subclasses of the class **Turtle**. Simulation agents will be told to `act()` once per timestep—it's in that method wolves and deer will do whatever they are told to do. **Deer** instances also know how to `die()`, and **Wolf** instances also know how to find the closest deer (in order to eat it).
- We will use the handy-dandy **LLNode** class in order to create a linked list of agents (which are in our case kinds of **Turtle**) through the **AgentNode** class. The **AgentNode** class knows how to get and set its agents (**Turtle** instances) and to remove an agent from the list. Removing an agent from the list of agents is a little more complicated

than simply removing a node from a linked list—the first thing we have to do is to find the node containing the input turtle and *then* remove the node.

- The overall WolfDeerSimulation keeps track of all the wolves and deer. It also knows how to run() the simulation. To run a simulation has a few basic parts:
 - The world must be set up and populated with wolves and deer.
 - In a loop (often called a *event loop* or *time loop*), time is incremented.
 - At each moment in time, each agent in the world is told to do whatever it needs to do (e.g., act()). Then the world display is updated.

Now, while that may seem complex, the reality is that Figure 15.2 doesn't capture all the relationships between the different classes in this simulation. For example, how does the WolfDeerSimulation keep track of the wolves and deer? You can probably figure out that it must have instance variables that hold AgentNode instances. But this figure doesn't reflect the entire *object model*. Let's talk about how to describe object models using a software industry standard that captures more of the details in how the classes in the simulation interact.

15.3 Modelling in Objects

As we said at the beginning of this chapter, object-oriented programming was invented to make simulations easier to build. The individual objects are clearly connected to real-world objects, but there are also techniques for thinking about how classes and objects relate to one another that helps to capture how objects in the real world relate to one another. Using these techniques is referred to as *modelling* in terms of objects, or *object modelling*. We call the process of studying a situation and coming up with the appropriate object-oriented model *object-oriented analysis*.

Computer Science Idea: The relationship between objects is meant to model reality

When an object modeler sets up relationships between objects, she is making a statement about how she sees the real world works.

Two of the kinds of object relationships that we use in modelling are *generalization-specialization* and *aggregation*.

- When we create a *generalization-specialization* relationship, we are saying that one class “is a kind of” the other class. Generalization-specialization relationships occur in the real world. Think of muscle and blood cells as specializations of the general concept of a cell. This relationship is typically implemented as a *subclass-superclass relationship*. When we created the Student class by extending Person, we were saying that a student is a kind of person. A student is a *specialization* of a person. A person is a *generalization* of a student.
- Another common object relationship is *aggregation*. This is simply the idea that objects exist within one another. Within you are organs (heart, lungs, brain), and each of these are individualized cells. If we were to model these organs and cells as objects, we would have objects inside of other objects.

Expert object modelers make a further distinction in aggregation between what we might call “*has-a*” relationship (sometimes called an *association relationship* or simply a *reference relationship*), in contrast with an aggregation where the parts make up the whole. Imagine that you were modelling a human being. You could model the human as two arms and hands, two legs and feet, a torso, and neck and head. In that case, all those parts make up the whole of a human—that’s the latter kind of aggregation. On the other hand, if you were modelling the cardiovascular system, you might model just hearts and lungs for each human in your simulation—even though, we know that humans have more than just hearts and lungs in them. A human “*has-a*” heart and two lungs, but that’s not all that they are. Modelers sometimes call that an *association relationship*—humans *is associated with* a heart object and two lung objects.

We can describe many kinds of object models with just these two kinds of relationships. But if we were to spell out sentences like “A Person and a Student have a generalization-specialization relationship” for every relationship in our models, they would go on for pages. Just look at what we wrote in the last section for the wolves and deer simulation, and that wasn’t even all the relationships in that model!

Object modelers use graphical notations like the *Unified Modelling Language* for describing their models. The Unified Modelling Language (*UML*) has several different kinds of diagrams in it, such as diagrams for describing the order of operations in different objects over time (a *collaboration diagram* or *sequence diagram*) or describing the different states (values of variables) that an object can be in during a particular process (a *state diagram*). The diagram that we’re going to use is the *class diagram* that describes the relationships, methods, and fields in an object model.

Figure 15.3 is a UML class diagram describing the classes in our wolves and deer simulation. There are lots of relationships described in this diagram. While it may look complex, there really are two only kinds of rela-

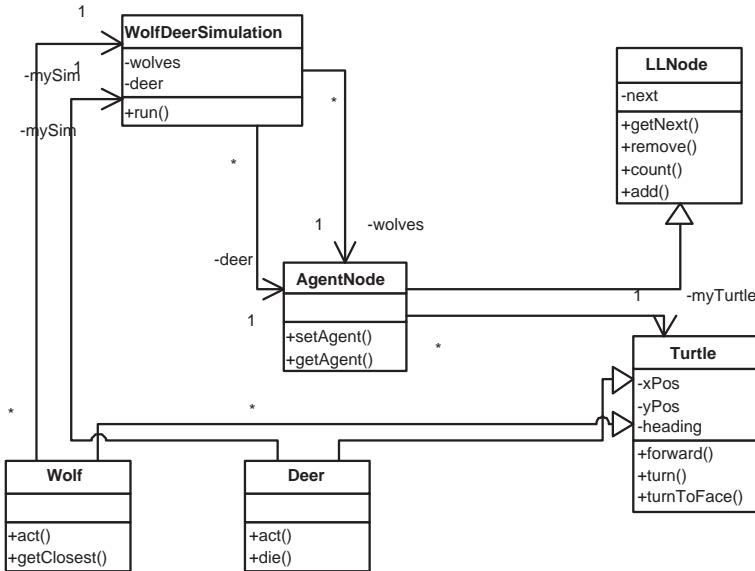


Figure 15.3: A UML class diagram for the wolves and deer simulation

tionships going on here, and the rest are things that you already know a lot about.

The boxes are the individual UML classes (Figure 15.4). They are split into thirds. The top part lists the name of the class. The middle part lists the instance variables (also called *fields*) for this class. Besides the name, sometimes the type of the variable is also listed (e.g., what kind of objects are stored in this variable?). The symbols in front of the names of the fields indicate the accessibility. A '+' indicates a public field, a '-' indicates a private field, and a '#' indicates a protected field. Finally, the bottom lists the methods or operations for this class. Like the fields, the accessibility is also indicated with a prefix on the method name.

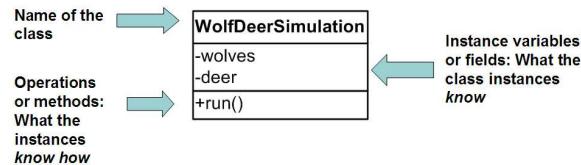


Figure 15.4: One UML class

Some fields may not appear in the class box. Instead, they might appear as a name on a reference relationship. Figure 15.5 pulls out just the

reference relationship from the overall diagram. Reference relationships (“has-a” relationship) have open arrow points, and they indicate that one kind of object contains a reference to the other object. In this example, we see that the class Wolf contains a reference to its WolfDeerSimulation. The name of this reference is mySim. This means that Wolf contains an additional field named mySim that doesn’t have to appear in the class box. You’re also seeing a “1” on one end of the reference link, and a “*” on the other end. This means that each Wolf references exactly one WolfDeerSimulation, but many (that’s what “*” means – anywhere from 0 to infinity) wolves might be in one simulation. The arrowhead could actually be on both sides. If a WolfDeerSimulation referenced at least one Wolf, then the arrows would go both way.

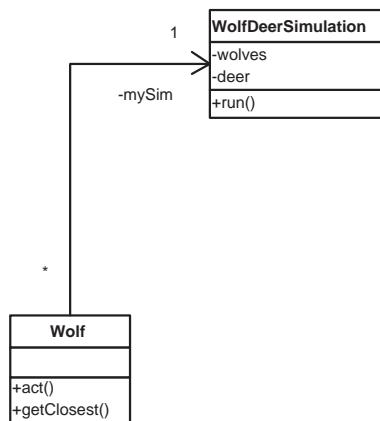


Figure 15.5: A Reference Relationship

At this point, you might be wondering, “Huh? I thought that there were wolves in this simulation?” There are other, but not direction from the simulation object to the wolf object. There are other objects in there. Follow the lines in Figure 15.3. WolfDeerSimulation contains an AgentNode named wolves. (See that “1” in there? Exactly one direct reference.) AgentNode contains a turtle named myTurtle. That’s how a simulation contains wolves and deer.

Because, odd as it seems, this diagram claims that wolves and deer are kinds of turtles (Figure 15.6). The lines that have closed arrows at their ends are depicting generalization-specialization (*gen-spec*) relationships. The arrow points toward the generalization (superclass). Figure 15.6 says that deers are kinds of turtles. (Are you imagining small turtles with antlers pasted onto their heads? Or maybe with gray or brown fur glued onto the shells?) This is what is sometimes called *implementation inheritance*—we want Deer instances to behave like Turtles in terms of movement

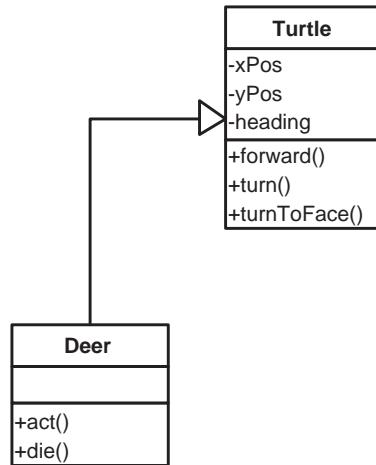


Figure 15.6: A Gen-Spec (Generalization-Specialization) relationship

and appearance. But from a modelling perspective, it's pretty silly to say that a deer is a kind of turtle. We'll fix that later.

15.4 Implementing the Simulation Class

The whole WolfDeerSimulation can be found at Program Program Example #102 (page 352). Let's walk through the key parts here.

```

public class WolfDeerSimulation {

    /* Linked lists for tracking wolves and deer */
    private AgentNode wolves;
    private AgentNode deer;

    /** Accessors for wolves and deer */
    public AgentNode getWolves(){return wolves;}
    public AgentNode getDeer(){return deer;}
  
```

Why do we declare our `wolves` and `deer` references to be **private**? Because we don't want them to be access except through accessors that we provide. Just imagine some rogue hacker wolf, gaining access to the positions of *all* the deer! No, of course, that's not the idea. But access to data is an important part of the model. Wolves *can* get deer's positions—by seeing them! If that's the only way that wolves find deer in the real world, then we should make sure that that's the only way it happens in our model, and we'll hide information that the wolf shouldn't have access to. If all the data were public, then a programmer might accidentally access data that one part of the model isn't supposed to access.

Now let's start looking at the run() method.

```
public void run()
{
    World w = new World();
    w.setAutoRepaint(false);

    // Start the lists
    wolves = new AgentNode();
    deer = new AgentNode();

    // create some deer
    int numDeer = 20;
    for (int i = 0; i < numDeer; i++)
    {
        deer.add(new AgentNode(new Deer(w, this)));
    }

    // create some wolves
    int numWolves = 5;
    for (int i = 0; i < numWolves; i++)
    {
        wolves.add(new AgentNode(new Wolf(w, this)));
    }
}
```

The first part of this method is saying that we don't want the World doesn't update (repaint) until we tell it to. Within a single time step, everything is supposed to be happening at the same moment, but we have to tell each agent to act() separately. We won't tell the World to update during each turtle (er, wolf and deer) movement. So we'll tell it to wait.

The rest of the example above is creating the wolves and deer lists. Notice that the deer variable references an AgentNode that is *empty*—there's no deer in there! Same for wolves. Then in the loops, we create each additional AgentNode with a Deer or a Wolf. Each Deer and Wolf takes as input the world w and the simulation **this**. The new AgentNodes get added to the respective linked lists.

Why the empty node at the front? This is actually a much more common linked list structure than the one that we've used up until now. Figure 15.7 describes what the structure looks like. This is sometimes called a *head-tail* or *head-rest* structure. What we're doing is setting up a node for wolves and deer to point at that does *not* itself contain a wolf or deer. Why? Recall our remove() code for removing a node from a list—it can't remove the first node in the list. How could it—we can't change what the variables wolves and deer point at within the method. If the nodes that wolves and deer reference actually contained a wolf and a deer, those would be invulnerable objects—they could never die and thus be removed from the list of living wolves or deer. The first node is immortal! Since that's probably a highly unusual situation to have immortal wolves and deer, we'll use a head-tail structure so that we can remove animals from our list.



Figure 15.7: The structure of the wolves linked list

Here's the next part of the run() method, where we invite our wolves and deer act.

```

// declare a wolf and deer
Wolf currentWolf = null;
Deer currentDeer = null;
AgentNode currentNode = null;

// loop for a set number of timesteps (50 here)
for (int t = 0; t < 50; t++)
{
    // loop through all the wolves
    currentNode = (AgentNode) wolves.getNext();
    while (currentNode != null)
    {
        currentWolf = (Wolf) currentNode.getAgent();
        currentWolf.act();
        currentNode = (AgentNode) currentNode.getNext();
    }
    // loop through all the deer
    currentNode = (AgentNode) deer.getNext();
    while (currentNode != null)
    {
        currentDeer = (Deer) currentNode.getAgent();
        currentDeer.act();
        currentNode = (AgentNode) currentNode.getNext();
    }
}
  
```

What's going on in here is that, within a **for** loop that counts up to 50 time steps, we traverse the wolves and then deer linked lists, inviting each one to act(). But the code probably looks more complicated than that, because of the *casting* going on.

- `currentNode = (AgentNode) wolves.getNext();`—remember that AgentNode is a subclass of LLNode. `getNext()` returns an LLNode, so we have to cast it to an AgentNode to be able to do AgentNode-specific stuff, like getting at the agent.
- `currentWolf = (Wolf) currentNode.getAgent();`—remember that AgentNodes contain Turtles, but we need a Wolf to get it to act(). So, we have to cast again.

Same things are going on in the Deer part of the loop.

Finally, the end of the run() method in WolfDeerSimulation.

```
// repaint the world to show the movement
w.repaint();

// Let's figure out where we stand...
System.out.println(">>> Timestep: "+t);
System.out.println("Wolves left: "+wolves.getNext().count());
System.out.println("Deer left: "+deer.getNext().count());

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

}
```

First, we tell the world “Everyone’s had a chance to update! Show the world!” We then print the current statistics about the world—how many deer and wolves are left, by counting. Our count() method is very simple and doesn’t understand about head-tail structures, so we’ll call count() on the tail (the getNext()) to avoid counting the empty *head* as one wolf or deer. If you have a really fast computer and the world is updating faster than you can really see (one can dream), you might want to uncomment the line Thread.sleep(). That causes the execution to pause for 1000 milliseconds—a whole second, so that you can see the screen before it updates again.

15.5 Implementing a Wolf

The complete Wolf class can be found at Program Program Example #103 (page 355). Here’s how it starts.

```
import java.awt.Color;
import java.util.Random;
import java.util.Iterator;

/**
 * Class that represents a wolf. The wolf class
 * tracks all the living wolves with a linked list.
 *
 * @author Barb Ericson ericson@cc.gatech.edu
 */
public class Wolf extends Turtle
{
    /////////////// fields //////////////////////

    /** class constant for the color */
    private static final Color grey = new Color(153,153,153);
    /** class constant for probability of NOT turning */
    protected static final double PROB_OF_STAY = 1/10;
    /** class constant for top speed (max num steps can move in a timestep) */
    protected static final int maxSpeed = 60;
```

```
/** My simulation */
protected WolfDeerSimulation mySim;

/** random number generator */
protected static Random randNumGen = new Random();
```

There's a new term we've never seen here before: **final**. A **final** variable is one that can't actually ever vary—it's value is stuck right from the beginning. It's a *constant*. You never *have* to say **final**, but there are advantages to using it. Java can be more efficient in its use of memory and generate even a little faster code if you declare things that will never change **final**.

Making It Work Tip: Names of constants are capitalized

Java discourse rules say that you capitalize values that are **final**, that will never change, in order to highlight them and set them apart.

We're using **final** values here to set the value of the wolves and the probability that they will *not* turn in any given time step. We really only need one copy of these variables for the whole class (e.g., we don't need another copy of the color grey for each and every wolf), so we're declaring them **static**, too. We declare them **protected** because we might want to create new kinds of wolves (specializations of wolves) that will want to access these. Remember that **protected** fields can be accessed by the class or any of its subclasses¹.

In the past, when we needed a random value, we simply accessed the method `Math.random()`. That method returns a **double** between 0 and 1 where all values are equally likely. That all numbers are “equally likely” is called a *uniform distribution*. That's a problem because relatively few things are uniformly distributed. Think about heights in your room or in your school. Let's say that you have someone who is 6 foot 10 inches tall and someone else 4 foot 11 inches tall. Does that mean that there are just as many people 6-10 as there are 4-11, and 5-0, 5-1, 5-2, and so on? We know that that's not how heights work. Most of the people are near the average height, and only a few people are at the maximum or minimum. That's a *normal distribution*, so-named because it is so, well, *normal*!

Instances of the class `Random` know how to generate random values on a normal distribution, as well as on a uniform distribution. We'll see those methods later, but we'll get started creating instances of `Random` now to get ready for that.

¹In Java, **protected** fields can actually be accessed from any class in the same package. Did you notice us creating any new packages so far? No? Then by default, all the code we've created can access any **protected** field. Not particularly **protected**, so we don't use **protected** much.

The next part of Wolf are the constructors. The constructors for Wolf are fairly complicated. They have to match Turtle's constructors which have to do with things like ModelDisplay which is an **interface** that World obeys. We will also have them call init() in order to *initialize* the agent. Here's what they look like.

```
//////////////////// Constructors //////////////////////////////

/*
 * Constructor that takes the model display (the original
 * position will be randomly assigned)
 * @param modelDisplayer thing that displays the model
 * @param mySim my simulation
 */
public Wolf (ModelDisplay modelDisplayer ,WolfDeerSimulation thisSim)
{
    super(randNumGen.nextInt(modelDisplayer.getWidth()) ,
          randNumGen.nextInt(modelDisplayer.getHeight()) ,
          modelDisplayer);
    init(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 mySim my simulation
 */
public Wolf (int x, int y, ModelDisplay modelDisplayer ,
             WolfDeerSimulation thisSim)
{
    // let the parent constructor handle it
    super(x,y,modelDisplayer);
    init(thisSim );
}
```

Initializing a Wolf is fairly simple.

```
////////////////// methods //////////////////////////////

/*
 * Method to initialize the new wolf object
 */
public void init(WolfDeerSimulation thisSim)
{
    // set the color of this wolf
    setColor(grey);

    // turn some random direction
```

```
this.turn(randNumGen.nextInt(360));

// set my simulation
mySim = thisSim;
}
```

Here, we are setting the wolf's color to grey, making it point in some random direction, and setting its reference mySim back up to the simulation that was passed in via the constructor. The method nextInt on Random returns a random integer between 0 and one less than the number provided as input. So randNumGen.nextInt(360) then returns a random number between 0 and 359.

Next comes a very important method, especially if you are a wolf. How do we figure out if there's a deer near enough to eat?

```
public AgentNode getClosest(double distance,AgentNode list)
{
    // get the head of the deer linked list
    AgentNode head = list;
    AgentNode curr = head;
    AgentNode closest = null;
    Deer thisDeer;
    double closestDistance = 0;
    double currDistance = 0;

    // loop through the linked list looking for the closest deer
    while (curr != null)
    {
        thisDeer = (Deer) curr.getAgent();
        currDistance = thisDeer.getDistance(
            this.getXPos(),this.getYPos());
        if (currDistance < distance)
        {
            if (closest == null || currDistance < closestDistance)
            {
                closest = curr;
                closestDistance = currDistance;
            }
        }
        curr = (AgentNode) curr.getNext();
    }
    return closest;
}
```

The method getClosest() searches through the given list to see if there's a deer-agent in the list that is within distance (the input parameter) of **this** wolf. Wolves can only see or hear or smell within some range or distance, according to our model. So, we'll look for the closest deer within the range. If the closest deer is outside the range, this method will just return **null**.

For the most part, this is just a traversal of the linked list. We walk the list of AgentNodes, and grab thisDeer out of the current node curr. We then compute the distance between thisDeer and this wolf's position (`this.getXPos()`, `this.getYPos()`). If the distance to this deer is within our range distance, then we consider if it's the closest. The two vertical bars (||) mean logical “or.” If we have no closest deer yet (`closest == null`) or if this is closer than our current closest deer (`currDistance < closestDistance`), then we say that the current AgentNode is the closest, and that this currDistance is the new closestDistance. At the end, we return the closest AgentNode.

Now that we know how wolves will go about finding something to eat, we can see how they will actually behave when told to `act()`.

```
/*
 * 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 some specified distance
    AgentNode closeDeer = getClosest(30,
                                    (AgentNode) mySim.getDeer().getNext());

    if (closeDeer != null)
    {
        Deer thisDeer = (Deer) closeDeer.getAgent();
        this.moveTo(thisDeer.getXPos(),
                    thisDeer.getYPos());
        thisDeer.die();
    }
    else // if we can't eat, then move
    {

        // 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(maxSpeed));
    }
}
```

Here's what our Wolf does:

- The very first thing a wolf does is to see if there's something to eat! It checks to see if there's a close deer within its sensing range (30). If there is, the wolf gets the deer out of the agent (via `getAgent`), move to the position of that deer, and eat the deer (tell it to die).

- If the wolf can't eat, it moves. It generates a random number (with `nextFloat()`) which returns a uniform number between 0 and 1), and if that random number is greater than the probability of just keeping our current heading (`PROB_OF_STAY`), then we turn some random amount. We then move forward at some random value less than a wolf's maximum speed.

15.6 Implementing Deer

The Deer class is at Program Program

Example #104 (page 358). There's not much new in the declarations of Deer (e.g., we declare a **static final** value for brown instead of grey) or in the constructors. When Deer instances `act()`, they don't eat anything in this model, so they don't have to hunt for closest anything. Instead, they just run around randomly. Note that they don't even look for wolves and try to get away from them yet.

```
/*
 * Method to act during a time step
 * pick a random direction and move some random amount up to top speed
 */
public void act()
{
    // 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(maxSpeed));
}
```

The interesting thing that Deer instances do in contrast to Wolf instances is to die.

```
/*
 * Method that handles when a deer dies
 */
public void die()
{
    // Leave a mark on the world where I died...
    this.setBodyColor(Color.red);

    // Remove me from the "live" list
    mySim.getDeer().remove(this);

    // ask the model display to remove this
    // Think of this as "ask the viewable world to remove this turtle"
```

```

// this.getModelDisplay().remove(this);

System.out.println("<SIGH!> A deer died... ");
}

```

When a Deer instance dies, we set its body color to Color.red. We then remove the deer from the list of living deer. We could, if we wished, remove the body of the dead deer from the screen, by removing the turtle from the list of turtle's in the world. That's what `this.getModelDisplay().remove(this);` does. Finally, we print the deer's obituary to the screen.

15.7 Implementing AgentNode

The full class AgentNode is at Program Program Example #105 (page 361). As we know from our earlier use of LLNode, there's not much to AgentNode—it's fairly easy to create a linked list of agents (Turtle instances) by subclassing LLNode.

The interesting part of AgentNode is the removal method that takes a Deer as input and removes the AgentNode that contains the input Deer instance.

```

/**
 * Remove the node where this turtle is found.
 */
public void remove(Turtle myTurtle) {
    // Assume we're calling on the head
    AgentNode head = this;
    AgentNode current = (AgentNode) this.getNext();

    while (current != null) {
        if (current.getAgent() == myTurtle)
            // If found the turtle, remove that node
            head.remove(current);
    }

    current = (AgentNode) current.getNext();
}

```

In this method, we have a linked list traversal where we're looking for the node whose agent is the input Turtle—`current.getAgent() == myTurtle`. Once we find the right node, we call the normal linked list `remove()` on that node.

This is a generally useful method—it removes a node based on the *content* of the node. Could we add this method to LLNode? Do we have to subclass and create AgentNode in order to make this work? Actually we could create a general linked list class that could hold anything. There is a class named Object that is the superclass of everything—even if you don't

say **extends**, you are implicitly subclassing Object in Java. If you had an instance variable that was declared Object, it could hold any kind of content: A Picture, a Turtle, a Student—anything.

15.8 Extending the Simulation

There are lots of things that we might change in the simulation. We might have wolves that are hungry sometimes and not hungry other times. We could have wolves chase deer, and have deer run from wolves. We'll implement these variations to see how we change simulations and implement different models. That is how people use simulations to answer questions. But to get answers, we need to do more than simply run the simulation and watch the pictures go by. We need to be able to get data out of it. We'll do that by generating files that can be read into Excel and analyzed, e.g., with graphs.

Making Hungry Wolves

Let's start out by creating a subclass of Wolf whose instances are sometimes hungry and sometimes satisfied. What's involved in making that happen? The whole class is at Program Program Example #106 (page 363).

```
/**  
 * A class that extends the Wolf to have a Hunger level.  
 * Wolves only eat when they're hungry  
 */  
public class HungryWolf extends Wolf {  
    /**  
     * Number of cycles before I'll eat again  
     */  
    private int satisfied;  
  
    /** class constant for number of turns before hungry */  
    private static final int MAX_SATISFIED = 3;
```

Obviously, we need to subclass Wolf. We will also need to add another field, satisfied, that will model how hungry or satisfied the HungryWolf is. We will have a new constant that indicates just how satisfied the HungryWolf instance is.

Here's how we will model satisfaction or hunger. When a HungryWolf eats, we will set the satisfied state to the MAX_SATISFIED. But each time that a time step passes, we decrement the satisfied state—making the wolf less satisfied. The wolf will only eat, then, if the wolf's satisfaction drops below zero.

The constructors for HungryWolf look just like Wolf's, which is as they must be.

```


    /**
     * Constructor that takes the model display (the original
     * position will be randomly assigned)
     * @param modelDisplayer thing that displays the model
     * @param mySim my simulation
     */
    public HungryWolf (ModelDisplay modelDisplayer ,WolfDeerSimulation thisSim)
    {
        super(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 mySim my simulation
     */
    public HungryWolf (int x, int y, ModelDisplay modelDisplayer ,
                       WolfDeerSimulation thisSim )
    {
        // let the parent constructor handle it
        super(x,y,modelDisplayer ,thisSim );
    }


```

The initialization method for HungryWolf does not do much, but nor does it need to. By simply calling upon its superclass, the HungryWolf only has to do what it must do as a specialization—start out satisfied.

```


    /**
     * Method to initialize the hungry wolf object
     */
    public void init(WolfDeerSimulation thisSim)
    {
        super.init(thisSim);

        satisfied = MAXSATISFIED;
    }


```

How a HungryWolf acts must also change slightly compared with how a Wolf. The differences require us to rewrite act()—we can't simply inherit and specialize as we did with init().

```


    /**
     * Method to act during a time step
     * pick a random direction and move some random amount up to top speed
     */
    public void act()
    {
        // Decrease satisfied time, until hungry again
        satisfied--;
    }


```

```

// get the closest deer within some specified distance
AgentNode closeDeer = getClosest(30,
                                (AgentNode) mySim.getDeer().getNext());

if (closeDeer != null)
{ // Even if deer close, only eat it if you're hungry.
  if (satisfied <= 0)
    {Deer thisDeer = (Deer) closeDeer.getAgent();
     this.moveTo(thisDeer.getXPos(),
                  thisDeer.getYPos());
     thisDeer.die();
     satisfied = MAX_SATISFIED;

    }
}
else
{
  // if the randome number is > prob of turning then turn
  if (randNumGen.nextFloat() > PROB_OF_TURN)
  {
    this.turn(randNumGen.nextInt(360));
  }

  // go forward some random amount
  forward(randNumGen.nextInt(maxSpeed));
}

}

```

The difference between HungryWolf and Wolf instances is that hungry wolves will eat, but satisfied ones will not. So after the HungryWolf finds a close deer, it considers whether it's hungry. If so, it eats the deer. If not, that's the end of act() for the HungryWolf. The Hungry Wolf only wanders aimlessly if it finds no deer.

How do we make the simulation work with HungryWolf? We only have to change the code in the run() method so that hungry wolves are created instead of normal wolves. *Everything else just works!* Because a HungryWolf is a kind of Wolf, all references to wolves work for hungry wolves.

```

// create some wolves
int numWolves = 5;
for (int i = 0; i < numWolves; i++)
{
  wolves.add(new AgentNode(new HungryWolf(w, this)));
}

```

Writing results to a file

As we start making changes to our simulations, we would like to get a sense for what effects our changes are making. Certainly, we can see the number of wolves and deer left at the end of the simulation, but what if we care about more subtle changes than that. What if we want to know, for example, how *quickly* the deer die? Is it all at once, or over time? Is it all at first, or all at the end?

But even more important than being able to compare different runs of our simulation, we may care about the results of any given simulation. Maybe we are ecologists who are trying to understand a particular setting for wolves and deer. Maybe we are trying to make predictions about what will happen in a given situation. If your simulation is a video game, you just want to watch it go by. But if your simulation is answering a question for you, you probably want to get the answers to your questions.

There are three parts to creating a file of data that we can open up and analyze in Excel.

1. We need to open a *stream* to a file. A stream is a data structure that efficiently handles data that flows – goes in only one direction.
2. We need to be able to write strings to that stream.
3. We need to be able to handle exceptional events, like the disk becoming full or the filename being wrong.

Java handles all file input and output as a stream. It turns out that streams are useful for more than just files. For example, you can create large, sophisticated strings, say, of HTML, by assembling the string using output streams. An input stream might be coming from a file, but might also be coming from a network connection, for example.

It turns out that you've been using streams already. There is a stream associated with where you can print known as System.out. Thus, when you use the method System.out.println(), you are actually sending a string to System.out by printing it using println. There is also a stream that you might have used called System.err where errors are expected to be printed. And as you might expect, there is a stream called System.in for taking in input from the keyboard. All of this suggests one way of handling the second task: We can get strings to our stream simply by using println. We can also use a method named write().

To get a stream on a file, we use a technique called *chaining*. Basically, it's wrapping one object in another so that you get the kind of access you want. To get a stream for reading from a file, you'll need a BufferedReader to create a stream, and a FileReader for accessing the file. It would look something like `new BufferedReader(new FileReader(fileName));`.

Thus, there are three parts to writing to a file.

1. Open up the stream. `writer = new BufferedWriter(new FileWriter(fileName));`

2. We write to the stream. writer.write(data);
3. When done, we close the stream (and the file). writer.write(data);

Now, it's not enough to have a stream. Java requires you to deal with the *exceptions* that might arise when dealing with input and output (I/O). Exceptions are disruptions in the normal flow of a program. There is actually a Java class named `Exception` that is used in handling exceptions. Things can go wrong when dealing with output to a stream. What happens if the disk fills? What if the filename is bad? In other programming languages, there are ways for programmers to check if something bad has happened—and most programmers don't check. Java *requires* the programmer handle exceptions. The programmer can be specific (e.g., “If the disk fails, do this. If the filename is bad, do that.”) or general (e.g., “If *anything* bad happens, here's how to bail out.”).

There is a special Java statement just for handling exceptions. It's called **try–catch**. It looks something like this:

```
try {
    \\code that can cause exceptions
} catch (ExceptionClassName varName) {
    \\code to handle this exception
} catch (ExceptionClassName varName) {
    \\code to handle that exception
}
```

You can deal with (“catch”) several exceptions at once, of different types. If you do try to distinguish between exceptions in a single statement like that, put the most general one last, e.g., catch the general `Exception`, and maybe one like `FileNotFoundException` (for trying to open a file for reading that doesn't exist) earlier in the list. All those other exceptions are subclasses of `Exception`, so catching `Exception` will handle all others. A general exception handling **try–catch** might look like this:

```
try {
    \\code that can throw the exception
} catch (Exception e) {
    System.err.println( Exception:      + e.getMessage());
    System.err.println( Stack Trace is:   );
    e.printStackTrace();
}
```

That `e` in the above code is a variable that will be a reference to the exception object, if an exception occurs. Exceptions know how to do several different methods. One returns (as a String) the error message associated with the exception: `e.getMessage()`. Another prints out what all methods were currently running and at what lines when the exception occurred: `e.printStackTrace()`.

There's an interesting variant on the **try–catch** that you should know about. You can specify a **finally** clause, that will always be executed *whether or not* exceptions occur.

```
try {
    \\\code that can cause the exception
} catch (FileNotFoundException ex) {
    \\\code to handle when the file isn't found
} finally {
    \\\code to always be executed
}
```

Putting it all together, here's how you would *read* from a file in Java.

```
BufferedReader reader = null;
String line = null;

// try to read the file
try {

    // create the buffered reader
    reader = new BufferedReader(new FileReader(fileName));

    // loop reading lines till the line is null (end of file)
    while ((line = reader.readLine()) != null)
    {
        // do something with the line
    }

    // close the buffered reader
    reader.close();
}

} catch (Exception ex) {
    // handle exception
}
```

Now let's add a file output capability to our simulation. We want to be able to do this:

```
Welcome to DrJava.
> WolfDeerSimulation wds = new WolfDeerSimulation();
> wds.openFile("D:/cs1316/wds-run1.txt")
> wds.run();
```

The idea is that the simulation instance (named `wds` above) should be able to run with or without an open file. If there is a file open, then text lines should be written the text file—one per each time step. And after running the simulation timing loop, the file should be automatically closed.

The first step is to create a new instance variable for `WolfDeerSimulation` that knows a `BufferedWriter` instance. By default (in the constructor), the output file should be `null`.

```
/* A BufferedWriter for writing to */
public BufferedWriter output;

/**
```

```
* Constructor to set output to null
*/
public WolfDeerSimulation() {
    output = null;
}
```

We're going to use the idea of the output file being **null** to *mean something*. If the file is **null**, we'll presume that there is no file to be written to. While that may be obvious, it's important to consider with respect to those pesky exceptions discussed earlier in this section. What happens if we *try* to write to the file but something bad happens? If anything happens untoward to the file processing, we'll simply set output to file. Then, the rest of the code will simply presume that there is no file to write—even though there *was* one once.

For example, we'll give WDSimulation instances the knowledge of how to openFile()—but if anything bad happens, output goes back to **null**.

```
/*
 * Open the input file and
 * set the BufferedWriter to speak to it.
*/
public void openFile(String filename){
    // Try to open the file
    try {

        // create a writer
        output = new BufferedWriter(
            new FileWriter(filename));

    } catch (Exception ex) {
        System.out.println(
            "Trouble opening the file " + filename);
        // If any problem, make it null again
        output = null;
    }
}
```

We need to change the bottom of the timing loop, too. We need to write to the file. But we *only* write to the file if output is not **null**. If an exception gets thrown, we set output back to **null**.

```
// Let's write out where we stand...
System.out.println(">>> Timestep: "+t);
System.out.println("Wolves left: "+wolves.getNext().count());
System.out.println("Deer left: "+deer.getNext().count());

// If we have an open file, write the counts to it
if (output != null) {
    // Try it
    try{
        output.write(wolves.getNext().count()+

```

```

        "\t"+deer.getNext().count());
        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;
    }
}
}

```

Check out the above for just a moment: Why are we saying wolves.getNext().count()? Why aren't we just saying wolves.count()? Remember that our count method counts every node from **this**. We haven't updated it yet for our head-rest list structure. Since the head of the list is empty, we don't want to include it in our count, so we start from its *next*.

After the timing loop, we need to close the file (if output is not **null**).

```

// If we have an open file, close it and null the variable
if (output != null){
    try{
        output.close();
    } catch (Exception ex)
    {System.out.println("Something went wrong closing the file");}
    finally {
        // No matter what, mark the file as not-there
        output = null;
    }
}

```

Getting results from a simulation

We can use our newly developed ability to write out results of a simulation a text file in order to analyze the results of the simulation. From DrJava, using our new methods, we can write out a text file like this.

```

> WolfDeerSimulation wds = new WolfDeerSimulation();
> wds.openFile("D:/cs1316/wds-run1.txt")
> wds.run();

```

Our file is made up of lines with a number, a tab, and another number. Excel can interpret that as two columns in a spreadsheet.

Exercises

1. Change the Deer so that there is no random amount that it moves—it always zips around at maximum speed. Do you think that that would make more Deer survive, since they'll be moving so fast? Try it! Can you figure out why it works the way that it does? Here's a hint (that gives away what you can expect): Notice where the dead deer bodies pile up.

2. There is an inefficiency to this simulation, in that we return the closest AgentNode, but then we just pull the Wolf or Deer out of it. And then when a Deer dies, we get the Deer out of the AgentNode and call remove()—but the first thing that AgentNode’s remove() does is to figure out the node containing the input Deer! These kinds of inefficiencies can arise when designing programs, but once you take a global perspective (considering what all the methods are and when they’re getting called), we can improve the methods and make them less efficient. Try fixing both of these problems in this simulation.
3. Build the LinkedListNode class that can contain any kind of object. Make sure that AgentNode’s remove works in that class.
4. The HungryWolf checks to see if there’s a close deer *and then* decides whether or not it’s hungry. Doesn’t that seem silly? Change the HungryWolf act() method so that it checks if it’s hungry *first*.

16 Abstracting Simulations: Creating a Simulation Package

Finally, let's make those villagers. We're going to do it in two steps:

- First, let's create a set of classes to make it easier to build simulations. We don't want to go to all that effort for every simulation we want to build. We'll build a few simulations using our new set of classes.
- Then, we'll map our turtles to characters in order to create simulations, like the wildebeests charging over the ridge.

17 Discrete Event Simulation

The difference between continuous and *discrete event simulations* is that the latter only represent *some* moments of time—the ones where something important happens. Discrete event simulations are very powerful for describing situations such as supermarkets and factory floors.

17.1 Distributions and Events

How do we represent how real things move and act in the real world? It's *random*, yes, but there are different kinds of random.

And once we make things happen randomly, we have to make sure that we keep true to *time order*—first things come first, and next things come next. We need to *sort* events in time order so that we deal with things accurately. We can also use *binary trees* and insertion into an *ordered list* to keep track of event order.

A MIDI Instrument names in JMusic

AAH	BREATHNOISE	EL_BASS
ABASS	BRIGHT_ACOUSTIC	EL_GUITAR
AC_GUITAR	BRIGHTNESS	ELECTRIC_BASS
ACCORDION	CALLOPE	ELECTRIC_GRAND
ACOUSTIC_BASS	CELESTA	ELECTRIC_GUITAR
ACOUSTIC_GRAND	CELESTE	ELECTRIC_ORGAN
ACOUSTIC_GUITAR	CELLO	ELECTRIC_PIANO
AGOGO	CGUITAR	ELPIANO
AHHS	CHARANG	ENGLISH_HORN
ALTO	CHIFFER	EPIANO
ALTO_SAX	CHIFFER LEAD	EPIANO2
ALTO_SAXOPHONE	CHOIR	FANTASIA
APPLAUSE	CHURCH_ORGAN	FBASS
ATMOSPHERE	CLAR	FIDDLE
BAG_PIPES	CLARINET	FINGERED_BASS
BAGPIPE	CLAV	FLUTE
BAGPIPES	CLAVINET	FRENCH_HORN
BANDNEON	CLEAN_GUITAR	FRET
BANJO	CONCERTINA	FRET_NOISE
BARI	CONTRA_BASS	FRETLESS
BARI_SAX	CONTRABASS	FRETLESS_BASS
BARITONE	CRYSTAL	FRETNOISE
BARITONE_SAX	CYMBAL	FRETS
BARITONE_SAXOPHONE	DGUITAR	GLOCK
BASS	DIST_GUITAR	GLOCKENSPIEL
BASSOON	DISTORTED_GUITAR	GMSAW_WAVE
BELL	DOUBLE_BASS	GMSQUARE_WAVE
BELLS	DROPS	GOBLIN
BIRD	DRUM	GT_HARMONICS
BOTTLE	DX_EPIANO	GUITAR
BOTTLE_BLOW	EBASS	GUITAR_HARMONICS
BOWED_GLASS	ECHO	HALO
BRASS	ECHO_DROP	HALO_PAD
BREATH	ECHO_DROPS	HAMMOND_ORGAN

Table A.1: JMusic constants in JMC for MIDI program changes, Part 1

HARMONICA	PANFLUTE	SLAP
HARMONICS	PBASS	SLAP_BASS
HARP	PHONE	SLOW_STRINGS
HARPSICHORD	PIANO	SOLO_VOX
HELICOPTER	PIANO_ACCORDION	SOP
HONKYTONK	PIC	SOPRANO
HONKYTONK PIANO	PICC	SOPRANO_SAX
HORN	PICCOLO	SOPRANO_SAXOPHONE
ICE RAIN	PICKED_BASS	SOUNDEFECTS
ICERAIN	PIPE_ORGAN	SOUNDFX
JAZZ_GUITAR	PIPES	SOUNDTRACK
JAZZ_ORGAN	PITZ	SPACE_VOICE
JGUITAR	PIZZ	SQUARE
KALIMBA	PIZZICATO_STRINGS	STAR_THEME
KOTO	POLY_SYNTH	STEEL_DRUM
MARIMBA	POLYSYNTH	STEEL_DRUMS
METAL_PAD	PSTRINGS	STEEL_GUITAR
MGUITAR	RAIN	STEELDRUM
MUSIC_BOX	RECORDER	STEELDRUMS
MUTED_GUITAR	REED_ORGAN	STR
MUTED_TRUMPET	REVERSE_CYMBAL	STREAM
NGUITAR	RHODES	STRINGS
NYLON_GUITAR	SAW	SWEEP
OBOE	SAWTOOTH	SWEEP_PAD
OCARINA	SAX	SYN_CALLIOPE
OGUITAR	SAXOPHONE	SYN_STRINGS
OOH	SBASS	SYNTH_BASS
OOHS	SEA	SYNTH_BRASS
ORCHESTRA_HIT	SEASHORE	SYNTH_CALLIOPE
ORGAN	SFX	SYNTH_DRUM
ORGAN2	SGUITAR	SYNTH_DRUMS
ORGAN3	SHAKUHACHI	SYNTH_STRINGS
OVERDRIVE_GUITAR	SHAMISEN	SYNVOX
PAD	SHANNAI	TAIKO
PAN_FLUTE	SITAR	TELEPHONE

Table A.2: JMusic constants in JMC for MIDI program changes, Part 2

TENOR		
TENOR_SAX		
TENOR_SAXOPHONE		
THUMB_PIANO		
THUNDER		
TIMP		
TIMPANI		
TINKLE_BELL		
TOM		
TOM_TOM		
TOM_TOMS		
TOMS		
TREMOLO		
TREMOLO_STRINGS		
TROMBONE		
TRUMPET		
TUBA		
TUBULAR_BELL		
TUBULAR_BELLS		
VIBES		
VIBRAPHONE		
VIOLA		
VIOLIN		
VIOLIN_CELLO		
VOICE		
VOX		
WARM_PAD		
WHISTLE		
WIND		
WOODBLOCK		
WOODBLOCKS		
XYLOPHONE		

Table A.3: JMusic constants in JMC for MIDI program changes, Part 3

B Whole Class Listings

Utility #2: Turtle

Utility Program

```
/*
 * Creates a Turtle on an input
 *
 ****
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import javax.swing.*;
import java.awt.image.*;  
  
public class Turtle {  
  
    private Picture myPicture;           // the picture that we're drawing on  
    private Graphics2D myGraphics;  
    JFrame myWindow;  
  
    private double x = 0.0, y = 0.0;        // turtle is at coordinate (x, y)  
    private int height, width;  
    private double heading = 180.0;         // facing this many degrees counterclockwise  
    private Color foreground = Color.black; // foreground color  
    private boolean penDown = true;  
  
    // turtles are created on pictures  
    public Turtle(Picture newPicture) {  
        myPicture = newPicture;  
        myGraphics = (Graphics2D) myPicture.getBufferedImage().createGraphics();  
        myGraphics.setColor(foreground);  
    }  
}
```

```
height = myPicture.getHeight();
width = myPicture.getWidth();
};

// accessor methods
public double x() { return x; }
public double y() { return y; }

public double heading() { return heading; }
public void setHeading(double newhead) {
    heading = newhead;
}

public void setColor(Color color) {
    foreground = color;
    myGraphics.setColor(foreground);
}

//Pen Stuff
public void penUp() {
    penDown = false;
}

public void penDown() {
    penDown = true;
}

public boolean pen() {
    return penDown;
}

public float getPenWidth() {
    BasicStroke bs = (BasicStroke) myGraphics.getStroke();
    return bs.getLineWidth();
}

public void setPenWidth(float width) {
    BasicStroke newStroke = new BasicStroke(width);
    myGraphics.setStroke(newStroke);
};

public void go(double x, double y) {
    if (penDown)
        myGraphics.draw(new Line2D.Double(this.x, this.y, x, y));
    this.x = x;
    this.y = y;
}
```

```

}

// draw w-by-h rectangle, centered at current location
public void spot(double w, double h) {
    myGraphics.fill(new Rectangle2D.Double(x - w/2, y - h/2, w, h));
}

// draw circle of diameter d, centered at current location
public void spot(double d) {
    if (d <= 1) myGraphics.drawRect((int) x, (int) y, 1, 1);
    else myGraphics.fill(new Ellipse2D.Double(x - d/2, y - d/2, d, d));
}

// draw spot using jpeg/gif - fix to be at (x, y)
public void spot(String s) {
    Picture spotPicture = new Picture(s);
    Image image = spotPicture.getImage();

    int w = image.getWidth(null);
    int h = image.getHeight(null);

    myGraphics.rotate(Math.toRadians(heading), x, y);
    myGraphics.drawImage(image, (int) x, (int) y, null);
    myGraphics.rotate(Math.toRadians(heading), x, y);
}

// draw spot using gif, left corner on (x, y), scaled of size w-by-h
public void spot(String s, double w, double h) {
    Picture spotPicture = new Picture(s);
    Image image = spotPicture.getImage();

    myGraphics.rotate(Math.toRadians(heading), x, y);
    myGraphics.drawImage(image, (int) x, (int) y,
                         (int) w, (int) h, null);
    myGraphics.rotate(Math.toRadians(heading), x, y);
}

public void pixel(int x, int y) {
    myGraphics.drawRect(x, y, 1, 1);
}

// rotate counterclockwise in degrees
public void turn(double angle) { heading = (heading + angle) % 360; }

// walk forward

```

```

        public void forward(double d) {
            double oldx = x;
            double oldy = y;
            x += d * -Math.cos(Math.toRadians(heading));
            y += d * Math.sin(Math.toRadians(heading));
            if (penDown)
                myGraphics.draw(new Line2D.Double(x, y, oldx, oldy));
        }

        // write the given string in the current font
        public void write(String s) {
            FontMetrics metrics = myGraphics.getFontMetrics();
            int w = metrics.stringWidth(s);
            int h = metrics.getHeight();
            myGraphics.drawString(s, (float) (x - w/2.0), (float) (y + h/2.0));
        }

        // write the given string in the given font
        public void write(String s, Font f) {
            myGraphics.setFont(f);
            write(s);
        }

    }
}

```

*Program
Example #101*

Example Java Code: WolfDeerSimulation.java

```

import java.io.*; // For BufferedWriter
2
public class WolfDeerSimulation {
4
    /* Linked lists for tracking wolves and deer */
6    private AgentNode wolves;
    private AgentNode deer;
8
    /** Accessors for wolves and deer */
10   public AgentNode getWolves(){return wolves;}
11   public AgentNode getDeer(){return deer;}
12
    /* A BufferedWriter for writing to */
14   public BufferedWriter output;
}

```

```

16  /**
17   * Constructor to set output to null
18   */
19  public WolfDeerSimulation() {
20      output = null;
21  }
22
23  /**
24   * Open the input file and set the BufferedWriter to speak to it.
25   */
26  public void openFile(String filename){
27      // Try to open the file
28      try {
29          // create a writer
30          output = new BufferedWriter(new FileWriter(filename));
31
32      } catch (Exception ex) {
33          System.out.println("Trouble opening the file " + filename);
34          // If any problem, make it null again
35          output = null;
36      }
37
38  }
39
40  public void run()
41  {
42      World w = new World();
43      w.setAutoRepaint(false);
44
45      // Start the lists
46      wolves = new AgentNode();
47      deer = new AgentNode();
48
49      // create some deer
50      int numDeer = 20;
51      for (int i = 0; i < numDeer; i++)
52      {
53          deer.add(new AgentNode(new Deer(w, this)));
54      }
55
56      // create some wolves
57      int numWolves = 5;
58      for (int i = 0; i < numWolves; i++)
59      {
60          wolves.add(new AgentNode(new HungryWolf(w, this)));
61      }
62
63      // declare a wolf and deer
64      Wolf currentWolf = null;

```

```

66     Deer currentDeer = null;
67     AgentNode currentNode = null;
68
69     // loop for a set number of timesteps (50 here)
70     for (int t = 0; t < 50; t++)
71     {
72         // loop through all the wolves
73         currentNode = (AgentNode) wolves.getNext();
74         while (currentNode != null)
75         {
76             currentWolf = (Wolf) currentNode.getAgent();
77             currentWolf.act();
78             currentNode = (AgentNode) currentNode.getNext();
79         }
80
81         // loop through all the deer
82         currentNode = (AgentNode) deer.getNext();
83         while (currentNode != null)
84         {
85             currentDeer = (Deer) currentNode.getAgent();
86             currentDeer.act();
87             currentNode = (AgentNode) currentNode.getNext();
88         }
89
90         // repaint the world to show the movement
91         w.repaint();
92
93         // Let's figure out where we stand...
94         System.out.println(">>> Timestep: "+t);
95         System.out.println("Wolves left: "+wolves.getNext().count());
96         System.out.println("Deer left: "+deer.getNext().count());
97
98         // If we have an open file , write the counts to it
99         if (output != null) {
100             // Try it
101             try{
102                 output.write(wolves.getNext().count()+"\t"+deer.getNext().count());
103                 output.newLine();
104             } catch (Exception ex) {
105                 System.out.println("Couldn't write the data!");
106                 System.out.println(ex.getMessage());
107                 // Make output null so that we don't keep trying
108                 output = null;
109             }
110         }
111
112         // Wait for one second
113         //Thread.sleep(1000);
114     }

```

```

116    // If we have an open file , close it and null the variable
117    if (output != null){
118        try{
119            output.close();
120        catch (Exception ex)
121            {System.out.println("Something went wrong closing the file");}
122        finally {
123            // No matter what, mark the file as not-there
124            output = null;
125        }
126    }
}

```

Example Java Code: **Wolf.java**

*Program
Example #102*

```

1  import java.awt.Color; import java.util.Random; import
   java.util.Iterator;
2
3 /**
4  * Class that represents a wolf.  The wolf class
5  * tracks all the living wolves with a linked list.
6  *
7  * @author Barb Ericson ericson@cc.gatech.edu
8  */
9 public class Wolf extends Turtle {
10
11    ////////////////// fields ///////////////////
12
13    /** class constant for the color */
14    private static final Color grey = new Color(153,153,153);
15
16    /** class constant for probability of NOT turning */
17    protected static final double PROB_OF_STAY = 1/10;
18
19    /** class constant for top speed (max num steps can move in a timestep) */
20    protected static final int maxSpeed = 40;
21
22    /** class constant for how far wolf can smell */
23    private static final double SMELL RANGE = 50;
24
25    /** class constant for how close before wolf can attack */
26    private static final double ATTACK RANGE = 30;
27
28    /** My simulation */
29    protected WolfDeerSimulation mySim;
}

```

```

31      /** random number generator */
33  protected static Random randNumGen = new Random();
35  ////////////////////////////////////////////////////////////////// Constructors //////////////////////////////////////////////////////////////////
37  /**
38   * Constructor that takes the model display (the original
39   * position will be randomly assigned)
40   * @param modelDisplayer thing that displays the model
41   * @param thisSim my simulation
42   */
43  public Wolf (ModelDisplay modelDisplayer, WolfDeerSimulation thisSim)
44  {
45      super(randNumGen.nextInt(modelDisplayer.getWidth()) ,
46            randNumGen.nextInt(modelDisplayer.getHeight()) ,
47            modelDisplayer);
48      init(thisSim);
49  }
50
51  /** Constructor that takes the x and y and a model
52   * display to draw it on
53   * @param x the starting x position
54   * @param y the starting y position
55   * @param modelDisplayer the thing that displays the model
56   * @param thisSim my simulation
57   */
58  public Wolf (int x, int y, ModelDisplay modelDisplayer,
59              WolfDeerSimulation thisSim)
60  {
61      // let the parent constructor handle it
62      super(x,y,modelDisplayer);
63      init(thisSim);
64  }
65
66  ////////////////////////////////////////////////////////////////// methods //////////////////////////////////////////////////////////////////
67
68
69  /**
70   * Method to initialize the new wolf object
71   */
72  public void init(WolfDeerSimulation thisSim)
73  {
74      // set the color of this wolf
75      setColor(grey);
76
77      // turn some random direction
78      this.turn(randNumGen.nextInt(360));
79
80      // set my simulation

```

```

81     mySim = thisSim;
82 }
83 /**
84 * Method to get the closest deer within the passed distance
85 * to this wolf. We'll search the input list of the kind
86 * of objects to compare to.
87 * @param distance the distance to look within
88 * @param list the list of agents to look at
89 * @return the closest agent in the given distance or null
90 */
91 public AgentNode getClosest(double distance,AgentNode list)
92 {
93     // get the head of the deer linked list
94     AgentNode head = list;
95     AgentNode curr = head;
96     AgentNode closest = null;
97     Deer thisDeer;
98     double closestDistance = 999;
99     double currDistance = 0;
100
101
102 // loop through the linked list looking for the closest deer
103 while (curr != null)
104 {
105     thisDeer = (Deer) curr.getAgent();
106     currDistance = thisDeer.getDistance(this.getXPos(),this.getYPos());
107     if (currDistance < distance)
108     {
109         if (closest == null || currDistance < closestDistance)
110         {
111             closest = curr;
112             closestDistance = currDistance;
113         }
114     }
115     curr = (AgentNode) curr.getNext();
116 }
117 return closest;
118 }
119
120 /**
121 * Method to act during a time step
122 * pick a random direction and move some random amount up to top speed
123 */
124 public void act()
125 {
126     // get the closest deer within some specified distance
127     AgentNode closeDeer = getClosest(30,
128                                     (AgentNode) mySim.getDeer().getNext());

```

```

131     if (closeDeer != null)
133     {
135         Deer thisDeer = (Deer) closeDeer.getAgent();
136         this.moveTo(thisDeer.getXPos(),
137                     thisDeer.getYPos());
138         thisDeer.die();
139     }
140
141     else
142     {
143         // if the random number is > prob of NOT turning then turn
144         if (randNumGen.nextFloat() > PROB_OF_STAY)
145         {
146             this.turn(randNumGen.nextInt(360));
147         }
148
149         // go forward some random amount
150         forward(randNumGen.nextInt(maxSpeed));
151     }
152 }
153 }
```

*Program
Example #103*

Example Java Code: **Deer.java**

```

import java.awt.Color; import java.util.Random;
2
 /**
4  * Class that represents a deer.  The deer class
5   * tracks all living deer with a linked list.
6   *
7   * @author Barb Ericson ericson@cc.gatech.edu
8 */
public class Deer extends Turtle {
10
11    //////////////////// fields ///////////////////
12
13    /** class constant for the color */
14    private static final Color brown = new Color(116,64,35);
15
16    /** class constant for probability of NOT turning */
17    private static final double PROB_OF_STAY = 1/5;
18}
```

```

15  /**
16   * class constant for how far deer can smell */
17  private static final double SMELLRANGE = 50;
18
19  /**
20   * class constant for top speed (max num steps can move in a timestep) */
21  private static final int maxSpeed = 30;
22
23  /**
24   * random number generator */
25  private static Random randNumGen = new Random();
26
27  /**
28   * the simulation I'm in */
29  private WolfDeerSimulation mySim;
30
31  //////////////////// Constructors ///////////////////
32
33  /**
34   * Constructor that takes the model display (the original
35   * position will be randomly assigned
36   * @param modelDisplayer thing which will display the model
37   */
38  public Deer (ModelDisplay modelDisplayer ,WolfDeerSimulation thisSim)
39  {
40      super(randNumGen.nextInt(modelDisplayer.getWidth()) ,
41             randNumGen.nextInt(modelDisplayer.getHeight()) ,
42             modelDisplayer);
43      init(thisSim);
44  }
45
46  /**
47   * Constructor that takes the x and y and a model
48   * display to draw it on
49   * @param x the starting x position
50   * @param y the starting y position
51   * @param modelDisplayer the thing that displays the model
52   */
53  public Deer (int x, int y, ModelDisplay modelDisplayer ,
54              WolfDeerSimulation thisSim)
55  {
56      // let the parent constructor handle it
57      super(x,y,modelDisplayer);
58      init(thisSim);
59  }
60
61  //////////////////// methods ///////////////////
62
63  /**
64   * Method to initialize the new deer object
65   */
66  public void init(WolfDeerSimulation thisSim)
67  {
68      // set the color of this deer
69      setColor(brown);
70  }

```

```

70      // turn some random direction
71      this.turn(randNumGen.nextInt(360));
72
73      // know my simulation
74      mySim = thisSim;
75
76  }
77
78  /**
79   * Method to get the closest wolf within the passed distance
80   * to this deer. We'll search the input list of the kind
81   * of objects to compare to.
82   * @param distance the distance to look within
83   * @param list the list of agents to look at
84   * @return the closest agent in the given distance or null
85   */
86  public AgentNode getClosest(double distance,AgentNode list)
87  {
88      // get the head of the deer linked list
89      AgentNode head = list;
90      AgentNode curr = head;
91      AgentNode closest = null;
92      Wolf thisWolf;
93      double closestDistance = 999;
94      double currDistance = 0;
95
96      // loop through the linked list looking for the closest deer
97      while (curr != null)
98      {
99          thisWolf = (Wolf) curr.getAgent();
100         currDistance = thisWolf.getDistance(this.getXPos(),this.getYPos());
101         if (currDistance < distance)
102         {
103             if (closest == null || currDistance < closestDistance)
104             {
105                 closest = curr;
106                 closestDistance = currDistance;
107             }
108         }
109         curr = (AgentNode) curr.getNext();
110     }
111     return closest;
112 }
113
114 /**
115  * Method to act during a time step
116  * pick a random direction and move some random amount up to top speed
117  */
118

```

```

120   public void act()
121   {
122     if (randNumGen.nextFloat() > PROB_OF_STAY)
123     {
124       this.turn(randNumGen.nextInt(360));
125     }
126
127     // go forward some random amount
128     forward(randNumGen.nextInt(maxSpeed));
129
130   }
131
132   /**
133    * Method that handles when a deer dies
134    */
135   public void die()
136   {
137     // Leave a mark on the world where I died...
138     this.setBodyColor(Color.red);
139
140     // Remove me from the "live" list
141     mySim.getDeer().remove(this);
142
143     // ask the model display to remove this
144     // Think of this as "ask the viewable world to remove this turtle"
145     //getModelDisplay().remove(this);
146
147     System.out.println("<SIGH!> A deer died... ");
148   }
149
150 }
```

Example Java Code: AgentNode

*Program
Example #104*

```

1  /**
2   * Class to implement a linked list of Turtle-like characters.
3   * (Maybe "agents"?)  

4   */
5   public class AgentNode extends LLNode {
6     /**
7      * The Turtle being held
8      */
9     private Turtle myTurtle;
```

```

11   /** Two constructors: One for creating the head of the list
12    * , with no agent
13   */
14   public AgentNode() {super();}
15
16   /**
17    * One constructor for creating a node with an agent
18    */
19   public AgentNode(Turtle agent){
20     super();
21     this.setAgent(agent);
22   }
23
24   /**
25    * Make a printable form
26    */
27   public String toString() {
28     return "AgentNode with agent (" + myTurtle + ") and next: " +
29           (AgentNode) getNext();
30   }
31
32   /**
33    * Setter for the turtle
34    */
35   public void setAgent(Turtle agent){
36     myTurtle = agent;
37   }
38
39   /**
40    * Getter for the turtle
41    */
42   public Turtle getAgent(){return myTurtle;}
43
44   /**
45    * Remove the node where this turtle is found.
46    */
47   public void remove(Turtle myTurtle) {
48     // Assume we're calling on the head
49     AgentNode head = this;
50     AgentNode current = (AgentNode) this.getNext();
51
52     while (current != null) {
53       if (current.getAgent() == myTurtle)
54         {// If found the turtle, remove that node
55          head.remove(current);
56        }
57
58        current = (AgentNode) current.getNext();
59      }

```

```
61    }
```

Example Java Code: HungryWolf

*Program
Example #105*

```

1  /**
2   * A class that extends the Wolf to have a Hunger level.
3   * Wolves only eat when they're hungry
4   */
5  public class HungryWolf extends Wolf {
6      /**
7       * Number of cycles before I'll eat again
8       */
9      private int satisfied;
10
11     /** class constant for number of turns before hungry */
12     private static final int MAXSATISFIED = 3;
13
14     ////////////////////////////// Constructors /////////////////////
15
16     /**
17      * Constructor that takes the model display (the original
18      * position will be randomly assigned)
19      * @param modelDisplayer thing that displays the model
20      * @param thisSim my simulation
21      */
22     public HungryWolf (ModelDisplay modelDisplayer ,WolfDeerSimulation thisSim)
23     {
24         super(modelDisplayer ,thisSim );
25     }
26
27     /**
28      * Constructor that takes the x and y and a model
29      * display to draw it on
30      * @param x the starting x position
31      * @param y the starting y position
32      * @param modelDisplayer the thing that displays the model
33      * @param thisSim my simulation
34      */
35     public HungryWolf (int x, int y, ModelDisplay modelDisplayer ,
36                         WolfDeerSimulation thisSim)
37     {
38         // let the parent constructor handle it
39         super(x,y,modelDisplayer ,thisSim );
40     }

```

```

41      /**
42      * Method to initialize the hungry wolf object
43      */
44  public void init(WolfDeerSimulation thisSim)
45  {
46      super.init(thisSim);
47
48      satisfied = MAX_SATISFIED;
49  }
50
51      /**
52      * Method to act during a time step
53      * pick a random direction and move some random amount up to top speed
54      */
55  public void act()
56  {
57      // Decrease satisfied time, until hungry again
58      satisfied--;
59
60      // get the closest deer within some specified distance
61      AgentNode closeDeer = getClosest(30,
62                                      (AgentNode) mySim.getDeer().getNext());
63
64      if (closeDeer != null)
65      { // Even if deer close, only eat it if you're hungry.
66          if (satisfied <= 0)
67          {Deer thisDeer = (Deer) closeDeer.getAgent();
68              this.moveTo(thisDeer.getXPos(),
69                          thisDeer.getYPos());
70              thisDeer.die();
71              satisfied = MAX_SATISFIED;
72
73      }
74
75      else
76      {
77          // if the random number is > prob of NOT turning then turn
78          if (randNumGen.nextFloat() > PROB_OF_STAY)
79          {
80              this.turn(randNumGen.nextInt(360));
81          }
82
83          // go forward some random amount
84          forward(randNumGen.nextInt(maxSpeed));
85
86      }
87  }
88
89 }

```


Bibliography

- [Goldberg and Robson, 1989] Goldberg, A. and Robson, D. (1989). *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley.
- [Resnick, 1997] Resnick, M. (1997). *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds*. MIT Press, Cambridge, MA.

Index

*/, 28
++, 37, 84
-, 37, 84
/*, 28

abstract superclass, 141
abstraction, 10
act(), 190
addFrame(aPicture), 71
addNote(), 33
AgentNode, 190
agents, 61, 188
aggregation, 191, 192
AmazingGraceSong, 96
AmazingGraceSongElement, 101
and, 21, 38
and (logical), 38
animations
 how they work, 71
API, 47
Application Program Interface, 47
array, 77
ArrayIndexOutOfBoundsException,
 23
association relationship, 192

BBN Labs, 61
behavior, 9
big-Oh, 86
binary trees, 219
block, 21, 38
bluescreen(), 56
Bobrow, Danny, 61
Boolean, 65
break, 22

BufferedReader, 210
buttons, 181
bytes, 78

Call and response, 98
calls, 43
cascade, 54
cast, 45, 84
casting, 198
chaining, 210
Chromakey, 55
chromakey(), 56
class, 20, 24, 40
class diagram, 192
class method, 24
CLASSPATH, 171
classpath, 15
collaboration diagram, 192
comment, 32
compiled, 20
compression, 78
connections, 11
constant, 199
constructor, 37, 78, 84, 90
continuous simulations, 187
coordinated resource, 188
curly braces, 21

data structure
 properties, 11
data structures, 8, 9
declare, 20
declaring, 20
delegate, 74
delegation, 74

die, 203
 die(), 190
 discourse rules, 36
 discrete event simulations, 187,
 219
 dot notation, 24
 double, 21, 46
 drawWith, 160
 DrJava, 15
 drop, 68
 edges, 11
 Element, 172
 else, 38
 event loop, 190
 event queue, 187
 Exception, 210
 exceptions, 210
 expert musicians, 100
 explore, 54
 extends, 47
 false, 65
 Feurzeig, Wally, 61
 field, 39, 45
 fields, 36, 40, 193
 FIFO list, 188
 file paths, 24
 FileChooser, 24
 getMediaPath, 24
 pickAFile, 24
 setMediaPath, 24
 FileChooser.getMediaPath, 54
 FileChooser.getMediapath, 24
 FileChooser.setMediaPath, 24, 54
 FileNotFoundException, 210
 FileReader, 210
 final, 199
 finally, 211
 for, 22, 39
 forward, 62, 63
 frames, 71, 156
 FrameSequence, 71
 methods, 71
 functions, 20
 gen-spec, 195
 generalization, 192
 generalization-specialization, 191
 getDistance(x,y), 67
 getMediaPath, 24, 54
 getMessage(), 211
 getSamples(), 79
 getSampleValueAt, 80
 getValue(), 79
 Goldberg, Adele, 186
 graph, 11
 graphical user interface (GUI), 181
 graphs, 179
 GUI, 181
 has-a, 192
 head-rest, 197
 head-tail, 197
 hierarchy, 10
 Hunchback of Notre Dame, The,
 7
 HungryWolf, 206
 IDE, 15, 36
 implementation inheritance, 195
 import, 32, 89
 increaseRed, 27
 index variable, 42
 information hiding, 48
 inherited, 160
 inherits, 47
 insertAfter, 131
 instance variable, 39, 45
 instance variables, 40
 instances, 24, 31, 112
 instrument, 90, 92
 Integrated Development Environment, 15
 Interface, 173
 interface, 186, 200
 invokes, 43
 jar file, 15
 Java, 8
 java, 60, 171
 Java compiler, 171
 Java Development Kit, 15

Java programming style, 36
 javac, 171
 JavaDoc, 47
 param, 153
 Javadoc, 92, 128
 Javanese, 65
 JDK, 15
 JMC, 90
 JMC.C4, 90
 JMC.FLUTE, 92
 JMC.QN, 90
 JMusic, 15, 32
 Javadoc, 92
 Kay, Alan, 19, 62, 187
 kind-of relationships, 191
 Knight Bus, 159
 LayeredSceneElement, 151
 layering, 141, 148
 layout managers, 181
 length, 45
 linked list, 10, 108, 143
 images, 143
 music, 101
 traversal, 108, 154
 linked lists, 89, 141
 Lion King, The, 7
 list, 14, 125, 143
 lists, 181
 literal, 21
 Logo, 62
 machine language, 20
 main, 59, 137
 Math.random(), 67, 97, 123, 200
 matrix, 10, 42
 memory, 12
 memory model, 12
 method, 27
 method signature, 47
 methods, 20, 35, 40
 MIDI, 31
 drum kit, 139
 MIDI channels, 92
 MIDI Drum Kit, 139
 MIDI music, 77
 MIDI note, 90
 MIDI program, 90
 MIT, 61
 MMList, 177
 model, 19, 101, 188
 ordering, 101
 modeled, 7
 modelling, 188, 191
 movies
 how they work, 71
 MTTree, 177
 Mufasa, 7
 Musical Information Data Inter-change, 31
 mySim, 193
 MySong, 137
 MyTurtleAnimation, 73
 new, 21
 new Picture(), 25
 nextFloat(), 203
 node, 125
 nodes, 11
 normal distribution, 200
 notate(), 33
 Note, 90
 null, 24, 44
 O(), 86
 Object, 205
 object, 19
 object model, 191
 object modelling, 191
 object-oriented analysis, 191
 object-oriented programming, 19
 openFile(), 212
 or, 21, 38
 or (logical), 38
 ordered list, 219
 ordering, 101
 OutOfBoundsException, 45
 package, 42
 panes, 181
 Papert, Seymour, 61

Part, 100, 124, 125, 138
 PATH, 171
 pausing, 198
 penDown(), 65
 penUp(), 65
 persistence of vision, 71
 Person, 192
 Phrase, 33, 90, 92, 101, 138
 pickAFile(), 24
 Picture, 20
 creating blank, 53, 54
 picture element, 42
 pig Latin, 62
 pile, 156
 Pixel, 26
 pixel, 42
 Pixels [], 26
 play(), 78
 PositionedSceneElement, 145, 151
 Potter, Harry, 159
 PowerPoint, 154
 predator and prey, 189
 print, 22
 println, 100
 printStackTrace(), 211
 private, 28
 properties, 11
 property, 156
 protected, 41, 200
 example of use, 200
 public, 28, 41
 public static void main, 59
 queue, 188
 Random, 200
 random, 219
 random number, 67
 random values, 200
 random(), 123
 rarefaction, 78
 recursion, 175
 refactoring, 125
 reference, 74
 reference relationship, 192
 references, 15
 render, 157
 rendering, 154
 repeatNext, 131
 repeatNextInserting, 132
 replay(delay), 71
 representation problem, 8
 resource, 187
 coordinated, 188
 responsibility-driven design, 19
 return, 54
 reverse(), 80
 Robson, David, 186
 Rotation, 173
 rotations, 172
 rsource
 fixed, 188
 run(), 190, 196
 Sample, 20
 sampled sounds, 77
 samples, 77
 satisfied, 206
 scale, 52
 scene graph, 172
 SceneElement, 160
 SceneElementLayered, 160
 SceneElementPositioned, 160
 scenes, 141
 scope, 26
 Score, 92, 100, 138
 self, 27
 sequence, 101
 sequence diagram, 192
 setMediaPath, 24, 54
 setPenDown(false), 65
 setPenUp(true), 65
 setSampleValueAt, 80
 show, 43
 FrameSequence, 71
 Picture, 20
 show(), 20, 25, 71
 showFromMeOn, 108
 SimplePicture, 47
 Simula, 19, 186
 simulation, 7
 event queue, 187

resources, 187
 simulations, 186
 continuous, 187
 defined, 186
 discrete event, 187
 sleeping, 198
 Smalltalk, 19, 62, 186
 Song, 139
 SongElement, 113, 125
 SongNode, 125
 SongPart, 134
 SongPhrase, 113, 134
 sort, 219
 Sound, 20, 78, 84
 getSamples, 79
 sound
 increasing volume, 78
 playing, 78
 SoundSample, 79
 getting and setting, 80
 getValue, 79
 specialization, 192
 square brackets, 23
 Stack, 156
 stack, 156
 StarLogo, 62
 state diagram, 192
 static, 24, 112
 stream, 100, 209
 string, 23
 compared to arrays, 23
 substring, 23
 string concatenation, 100
 structure, 9
 Student, 191
 subclass, 48, 160
 subclass-superclass relationship,
 191
 substring, 23
 substrings, 23
 superclass, 160
 abstract, 141
 System.err, 209
 System.in, 209
 System.out, 209
 System.out.println, 22
 System.out.println(), 100
 table, 10
 The Hunchback of Notre Dame, 7
 The Lion King, 7
 this, 27, 44
 Thread.sleep(), 198
 threshold, 55
 time loop, 190
 time order, 219
 time step, 189
 timestep, 189
 Toy Story, 7
 Translation, 173
 translations, 172
 traverses, 154
 tree, 10, 125, 134
 true, 65
 try-catch, 210
 turn, 62, 64
 turnToFace(anotherTurtle), 67
 Turtle, 63, 172, 188
 turtle, 61, 62
 turtle steps, 63
 types, 20
 UML, 192
 Unified Modelling Language, 192
 uniform distribution, 200
 View, 33
 Visio, 154
 void, 28, 43
 WAV files, 77
 while, 22, 38
 wildebeests, 7
 WolfDeerSimulation, 189, 190
 World, 63, 196
 write(), 209
 x++, 37