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 15, 2006
Copyright held by Mark Guzdial and Barbara Ericson, 2006.
Dedicated to TBD.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Contents</td>
<td>iii</td>
</tr>
<tr>
<td></td>
<td>List of Program Examples</td>
<td>vi</td>
</tr>
<tr>
<td></td>
<td>List of Figures</td>
<td>viii</td>
</tr>
<tr>
<td>I</td>
<td>Introduction to Java: Object-Oriented Programming for Modeling the World</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>Objects for Modeling the World</td>
<td>7</td>
</tr>
<tr>
<td>1.1</td>
<td>Making Representations of the World</td>
<td>8</td>
</tr>
<tr>
<td>1.2</td>
<td>Why Java?</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>Introduction to Java</td>
<td>19</td>
</tr>
<tr>
<td>2.1</td>
<td>What’s Java about?</td>
<td>19</td>
</tr>
<tr>
<td>2.2</td>
<td>Basic (Syntax) Rules of Java</td>
<td>20</td>
</tr>
<tr>
<td>2.3</td>
<td>Manipulating Pictures in Java</td>
<td>23</td>
</tr>
<tr>
<td>2.4</td>
<td>Exploring Sound in Java</td>
<td>29</td>
</tr>
<tr>
<td>2.5</td>
<td>Exploring Music in Java</td>
<td>30</td>
</tr>
<tr>
<td>3</td>
<td>Methods in Java: Manipulating Pictures</td>
<td>33</td>
</tr>
<tr>
<td>3.1</td>
<td>Reviewing Java Basics</td>
<td>33</td>
</tr>
<tr>
<td>3.2</td>
<td>Java is about Classes and Methods</td>
<td>37</td>
</tr>
<tr>
<td>3.3</td>
<td>Methods that return something: Compositing images</td>
<td>45</td>
</tr>
<tr>
<td>3.4</td>
<td>Creating classes that do something</td>
<td>54</td>
</tr>
<tr>
<td>4</td>
<td>Objects as Agents: Manipulating Turtles</td>
<td>57</td>
</tr>
<tr>
<td>4.1</td>
<td>Turtles: An Early Computational Object</td>
<td>57</td>
</tr>
<tr>
<td>4.2</td>
<td>Drawing with Turtles</td>
<td>58</td>
</tr>
<tr>
<td>4.3</td>
<td>Creating animations with turtles and frames</td>
<td>66</td>
</tr>
<tr>
<td>5</td>
<td>Arrays: A Static Data Structure for Sounds</td>
<td>73</td>
</tr>
<tr>
<td>5.1</td>
<td>Manipulating Sampled Sounds</td>
<td>73</td>
</tr>
<tr>
<td>5.2</td>
<td>Inserting and Deleting in an Array</td>
<td>79</td>
</tr>
</tbody>
</table>
CONTENTS

II Linked Lists: Linear Lists of Media

6 Structuring Music
   6.1 JMusic and Imports ........................................... 85
   6.2 Starting out with JMusic ...................................... 88
   6.3 Making a Simple Song Object .................................. 90
   6.4 Simple structuring of notes with an array ..................... 92
   6.5 Making the Song Something to Explore ......................... 94
   6.6 Making Any Song Something to Explore ......................... 101
   6.7 Structuring Music ............................................. 119

7 Linked Lists of Images
   7.1 Simple arrays of pictures .................................... 135
   7.2 Listing the Pictures, Left-to-Right .......................... 135
   7.3 Listing the Pictures, layering .................................. 141
   7.4 Representing scenes with trees ............................... 147
   7.5 Basic FrameSequence ......................................... 148

8 Abstract Data Types: Separating the Use from the Implementation ............................ 151

III Trees: Hierarchical Structures for Media

9 Trees of Images ...................................................... 155

10 Lists and Trees for Structuring Sounds ................................ 157

11 Generalizing Lists and Trees ...................................... 159

12 Graphs and Circular Linked Lists: Lists and Trees That Loop ................................. 161

13 User Interface Structures ......................................... 163

IV Simulations: Problem Solving with Data Structures

14 Introducing UML and Continuous Simulations .............................. 165
   14.1 Introducing Simulations ...................................... 167
   14.2 Our First Model and Simulation: Wolves and Deer ................ 170
   14.3 Modelling in Objects ......................................... 172
   14.4 Implementing the Simulation Class ............................ 176
   14.5 Implementing a Wolf .......................................... 179
   14.6 Implementing Deer ........................................... 184
   14.7 Implementing AgentNode ...................................... 185
   14.8 Extending the Simulation ...................................... 186
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 Abstracting Simulations: Creating a Simulation Package</td>
<td>195</td>
</tr>
<tr>
<td>16 Discrete Event Simulation</td>
<td>197</td>
</tr>
<tr>
<td>16.1 Distributions and Events</td>
<td>197</td>
</tr>
<tr>
<td>A MIDI Instrument names in JMusic</td>
<td>199</td>
</tr>
<tr>
<td>B Whole Class Listings</td>
<td>203</td>
</tr>
<tr>
<td>Bibliography</td>
<td>221</td>
</tr>
<tr>
<td>Index</td>
<td>223</td>
</tr>
</tbody>
</table>
## List of Program Examples

<table>
<thead>
<tr>
<th>Program Example #1: An Example Program</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program Example #2: Method to increase red in Picture</td>
<td>26</td>
</tr>
<tr>
<td>Program Example #3: Method to flip an image</td>
<td>28</td>
</tr>
<tr>
<td>Program Example #4: decreaseRed in a Picture</td>
<td>40</td>
</tr>
<tr>
<td>Program Example #5: decreaseRed with an input</td>
<td>43</td>
</tr>
<tr>
<td>Program Example #6: Method to compose this picture into a target</td>
<td>45</td>
</tr>
<tr>
<td>Program Example #7: Method for Picture to scale by a factor</td>
<td>48</td>
</tr>
<tr>
<td>Program Example #8: Methods for general chromakey and blue-screen</td>
<td>52</td>
</tr>
<tr>
<td>Program Example #9: A public static void main in a class</td>
<td>55</td>
</tr>
<tr>
<td>Program Example #10: Creating a hundred turtles</td>
<td>61</td>
</tr>
<tr>
<td>Program Example #11: Making a picture with dropped pictures</td>
<td>65</td>
</tr>
<tr>
<td>Program Example #12: An animation generated by a Turtle</td>
<td>67</td>
</tr>
<tr>
<td>Program Example #13: Increase the volume of a sound by a factor</td>
<td>74</td>
</tr>
<tr>
<td>Program Example #14: Reversing a sound</td>
<td>75</td>
</tr>
<tr>
<td>Program Example #15: Create an audio collage</td>
<td>76</td>
</tr>
<tr>
<td>Program Example #16: Append one sound with another</td>
<td>77</td>
</tr>
<tr>
<td>Program Example #17: Mix in part of one sound with another</td>
<td>78</td>
</tr>
<tr>
<td>Program Example #18: Scale a sound up or down in frequency</td>
<td>78</td>
</tr>
<tr>
<td>Program Example #19: Inserting into the middle of sounds</td>
<td>80</td>
</tr>
<tr>
<td>Program Example #20: Amazing Grace as a Song Object</td>
<td>91</td>
</tr>
<tr>
<td>Program Example #21: Amazing Grace with Multiple Voices</td>
<td>94</td>
</tr>
<tr>
<td>Program Example #22: Amazing Grace as Song Elements</td>
<td>96</td>
</tr>
<tr>
<td>Program Example #23: Amazing Grace as Song Elements, Take 2</td>
<td>101</td>
</tr>
<tr>
<td>Program Example #24: General Song Elements and Song Phrases</td>
<td>109</td>
</tr>
<tr>
<td>Program Example #25: More phrases to play with</td>
<td>112</td>
</tr>
<tr>
<td>Program Example #26: Computed Phrases</td>
<td>116</td>
</tr>
<tr>
<td>Program Example #27: 10 random notes SongPhrase</td>
<td>118</td>
</tr>
<tr>
<td>Program Example #28: 10 slightly less random notes</td>
<td>119</td>
</tr>
<tr>
<td>Program Example #29: SongNode class</td>
<td>120</td>
</tr>
<tr>
<td>Program Example #30: Repeating and weaving methods</td>
<td>124</td>
</tr>
<tr>
<td>Program Example #31: RepeatNextInserting</td>
<td>127</td>
</tr>
<tr>
<td>Program Example #32: SongPart class</td>
<td>129</td>
</tr>
<tr>
<td>Program Example #33: Song class—root of a tree-like music structure</td>
<td>130</td>
</tr>
<tr>
<td>Program Example #34: MySong class with a main method</td>
<td>132</td>
</tr>
</tbody>
</table>
List of Program Examples

Program Example #35: Elements of a scene in position order . . . . 137
Program Example #36: Methods to remove and insert elements in a
  list . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 140
Program Example #37: LayeredSceneElements . . . . . . . . . . . . 142
Program Example #38: WolfDeerSimulation.java . . . . . . . . . . 206
Program Example #39: Wolf.java . . . . . . . . . . . . . . . . . . . . 209
Program Example #40: Deer.java . . . . . . . . . . . . . . . . . . . . 212
Program Example #41: AgentNode . . . . . . . . . . . . . . . . . . . . 215
Program Example #42: HungryWolf . . . . . . . . . . . . . . . . . . . . 217
## List of Figures

1.1 Wildebeests in *The Lion King* .......................... 7  
1.2 Parisian villagers in *The Hunchback of Notre Dame* .......... 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 `java-source` to DrJava ............................... 17  
1.9 Parts of DrJava window .......................................... 17  

2.1 Showing a picture ................................................. 25  
2.2 Doubling the amount of red in a picture ....................... 26  
2.3 Doubling the amount of red using our `increaseRed` method ... 28  
2.4 Flipping our guy character–original (left) and flipped (right) .. 29  
2.5 Just two notes ....................................................... 31  

3.1 Structure of the Picture class defined in Picture.java ........ 38  
3.2 Part of the JavaDoc page for the Pixel class .................. 44  
3.3 Composing the guy into the jungle ............................. 46  
3.4 Mini-collage created with `scale` and `compose` ............... 51  
3.5 Using the `explore` method to see the sizes of the guy and the jungle 52  
3.6 Chromakeying the monster into the jungle using different levels of bluescreening ........................................ 53  
3.7 Run the main method from DrJava ............................... 56  

4.1 Starting a Turtle in a new World .............................. 59  
4.2 A drawing with a turtle ............................................. 60  
4.3 What you get with a hundred turtles starting from the same point, pointing in random directions, then moving forward the same amount ......................................................... 63  
4.4 Dropping the monster character .................................. 64  
4.5 Dropping the monster character after a rotation ................ 64  
4.6 An iterated turtle drop of a monster ............................ 65  
4.7 Making a more complex pattern of dropped pictures .......... 66
List of Figures

6.1 Playing all the notes in a score ................................. 86
6.2 Frequencies, keys, and MIDI notes ........................... 87
6.3 Viewing a multipart score ...................................... 90
6.4 JMusic documentation for the class Phrase ................... 90
6.5 Playing all the notes in a score ............................... 91
6.6 Trying the Amazing Grace song object ....................... 93
6.7 A hundred random notes ...................................... 93
6.8 Multi-voice Amazing Grace notation .......................... 96
6.9 AmazingGraceSongElements with 3 pieces ................... 100
6.10 AmazingGraceSongElements with 3 pieces ................. 100
6.11 Playing some different riffs in patterns .................... 115
6.12 Sax line in the top part, rhythm in the bottom .......... 118
6.13 We now have layers of software, where we deal with only one at a time .................................................. 118
6.14 First score generated from ordered linked list ........... 123
6.15 Javadoc for the class SongNode .............................. 124
6.16 Repeating a node several times .............................. 126
6.17 Weaving a new node among the old .......................... 126
6.18 Multi-part song using our classes ............................ 133
7.1 Array of pictures composed into a background ............. 136
7.2 Elements to be used in our scenes ............................ 136
7.3 Our first scene ................................................. 139
7.4 Our second scene .............................................. 139
7.5 Removing the doggy from the scene ........................... 141
7.6 Inserting the doggy into the scene ............................ 142
7.7 First rendering of the layered scene ......................... 145
7.8 Second rendering of the layered scene ...................... 146
7.9 Three frames from the simple FrameSequence example .... 148
9.1 Reserving more memory for the Interactions Pane in DrJava's Preferences pane ........................................... 156
14.1 An execution of our wolves and deer simulation ........ 170
14.2 The class relationships in the Wolves and Deer simulation . 171
14.3 A UML class diagram for the wolves and deer simulation .. 174
14.4 One UML class ................................................ 174
14.5 A Reference Relationship .................................... 175
14.6 A Gen-Spec (Generalization-Specialization) relationship ... 176
14.7 The structure of the wolves linked list ....................... 178
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:

```java
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:

```java
public class Greeter {
    public static void main(String[] argv)
    {
        // show the string "Hello World" on the console
        System.out.println("Hello World");
    }
}
```

**Program Example #0**

Example Java Code: **An Example Program**

A program creates a model of interest to us.

```java
import jm.music.data.*;
import jm.JMC;
import jm.util.*;
import jm.music.tools.*;

public class Dot03 {
    public static void main(String[] args) {
        Note n = new Note(JMC.C4, JMC.QUARTER_NOTE);
        Phrase phr = new Phrase(0.0);
        phr.addNote(n);
        Mod.repeat(phr, 15);

        Phrase phr2 = new Phrase(0.0);
        Note r = new Note(JMC.REST, JMC.EIGHTH_NOTE);
        phr2.addNote(r);
        Note n2 = new Note(JMC.E4, JMC.EIGHTH_NOTE);
        phr2.addNote(n2);
        Note r2 = new Note(JMC.REST, JMC.QUARTER_NOTE);
        phr2.addNote(r2);
        Mod.repeat(phr2, 7);

        Part p = new Part();
        p.addPhrase(phr);
        p.addPhrase(phr2);

        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 1.1). 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 1.2). 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*](image1)

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

---

CHAPTER 1. OBJECTS FOR MODELING THE WORLD

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
1.1. MAKING REPRESENTATIONS OF THE WORLD

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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arnold</td>
<td>12</td>
<td>220</td>
</tr>
<tr>
<td>Kermit</td>
<td>47</td>
<td>3</td>
</tr>
<tr>
<td>Ms. Piggy</td>
<td>42</td>
<td>54</td>
</tr>
</tbody>
</table>

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.)

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arnold</td>
<td>12</td>
<td>220</td>
</tr>
<tr>
<td>Fozzie</td>
<td>38</td>
<td>125</td>
</tr>
<tr>
<td>Kermit</td>
<td>47</td>
<td>3</td>
</tr>
<tr>
<td>Ms. Piggy</td>
<td>42</td>
<td>54</td>
</tr>
</tbody>
</table>

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.
1.1. MAKING REPRESENTATIONS OF THE WORLD

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:

```
bedroom  →  kitchen
```

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:

```
<table>
<thead>
<tr>
<th>Current location:</th>
<th>Where to go next:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bedroom</td>
<td>Kitchen</td>
</tr>
</tbody>
</table>
```

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 \textit{next} part of Katie’s bedroom note, and the location and \textit{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 \textit{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 \textit{list} (like a linked list).
- The images to be used in making the villagers wave or wildebeests run are usually stored in a list.

\section*{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.
1.2. **WHY JAVA?**

- 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](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](http://www.drjava.org).


- 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).
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
1.2. WHY JAVA?

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.

```java
> 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.

```java
> float f;
> f = 13.2;
```
2.2. BASIC (SYNTAX) RULES OF JAVA

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.

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

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

```java
> 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.

```java
> for (int num = 1 ; num <= 10 ; num++)
  System.out.println(num);
```
2.3. MANIPULATING PICTURES IN JAVA

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.

```java
> 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)
```

2.3 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().

```java
> 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 COMPIL<e> 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 can't find Picture, you can't 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()

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).

```java
> 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()

Figure 2.2: Doubling the amount of red in a picture

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.)

Program Example #1

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

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

public void increaseRed()
{
  Pixel px;
  int index = 0;
  Pixel [] mypixels = this.getPixels();
  while (index < mypixels.length)
```

```
2.3. MANIPULATING PICTURES IN JAVA

{  
px = mypixels[index];  
px.setRed(px.getRed() * 2);  
index = index + 1;  
}

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 COMPILATE 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()

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).
CHAPTER 2. INTRODUCTION TO JAVA

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

Program Example #2

Example Java Code: Method to flip an image

```java
/**
 * Method to flip an image left-to-right
 */
public Picture flip() {

    Pixel currPixel;
    Picture target = new Picture(this.getWidth(), this.getHeight());

    for (int srcx = 0, trgx = getWidth() - 1; srcx < getWidth(); srcx++, trgx --) {
        for (int srcy = 0, trgry = 0; trgry < getHeight(); srcy++, trgry++) {
            // get the current pixel
            currPixel = this.getPixel(srcx, srcy);

            /* copy the color of currPixel into target */
            target.getPixel(trgx, trgry).setColor(currPixel.getColor());
        }
    }
    return target;
}
```

2.4 Exploring Sound in Java

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

```java
    Picture p = new Picture(FileChooser.pickAFile());
    Picture flipp = p.flip();
    flipp.show();
```

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

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

**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.

2.4 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.

### 2.5 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.
2.5. EXPLORING MUSIC IN JAVA

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

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.

<table>
<thead>
<tr>
<th>What instances of this class know</th>
<th>What instances of this class do</th>
</tr>
</thead>
<tbody>
<tr>
<td>Note</td>
<td>A musical pitch and its duration.</td>
</tr>
<tr>
<td></td>
<td>(Nothing we’ve seen yet.)</td>
</tr>
<tr>
<td>Phrase</td>
<td>Notes in the phrase.</td>
</tr>
<tr>
<td></td>
<td>addNote(aNote)</td>
</tr>
</tbody>
</table>
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 declara-
tion, 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 in-
dices 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();
3.2. JAVA IS ABOUT CLASSES AND METHODS

> 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 var-
iables 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.

```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 an 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 #3

Example Java Code: decreaseRed in a Picture

```java
/**
 * Method to decrease the red by half in the current picture
 */
public void decreaseRed()
{
    Pixel pixel = null; // the current pixel
    int redValue = 0; // the amount of red

    // get the array of pixels for this picture object
    Pixel[] pixels = this.getPixels();

    // start the index at 0
    int index = 0;

    // loop while the index is less than the length of the pixels array
    while (index < pixels.length)
    {
        // get the current pixel at this index
        pixel = pixels[index];
        // get the red value at the pixel
        redValue = pixel.getRed();
        // set the red value to half what it was
        redValue = (int) (redValue * 0.5);
        // set the red for this pixel to the new value
        pixel.setRed(redValue);
        // increment the index
        index++;
    }
}
```

We would use this method like this:

```java
> 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
3.2. **JAVA IS ABOUT CLASSES AND METHODS**

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 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

```java
/**
 * 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:

```java
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: COMPOSING IMAGES

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

```
/**
 * 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() && trgry < target.getHeight())
            {
                newPixel = target.getPixel(trgx, trgry);
                newPixel.setColor(currPixel.getColor());
            }
        }
    }
}
```
Using this method, we can then compose the guy into the jungle like this (Figure 3.3).

```java
> 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](image)

**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
3.3. METHODS THAT RETURN SOMETHING: COMPOSITING IMAGES

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 trg 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() && trg < 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 maximum height of the target. We stay well inside the boundary of the picture that way.

So far, we’ve only 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).

```java
// 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 #6

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

```java
/**
 * Method to scale the picture by a factor, and return the result
 * @param scaleFactor to scale by (1.0 stays the same,
 * 0.5 decreases each side by 0.5, 2.0 doubles each side)
 * @return the scaled picture
 */
public Picture scale(double factor)
{
    Pixel sourcePixel, targetPixel;
    Picture canvas = new Picture(
        (int) (factor * this.getWidth()) + 1,
        (int) (factor * this.getHeight()) + 1);
    // loop through the columns
    for (double sourceX = 0, targetX = 0;
        sourceX < this.getWidth();
        sourceX += (1 / factor), targetX++)
    {
        // loop through the rows
        for (double sourceY = 0, targetY = 0;
            sourceY < this.getHeight();
            sourceY += (1 / factor), targetY++)
        {
            sourcePixel = this.getPixel((int) sourceX, (int) sourceY);
            targetPixel = canvas.getPixel((int) targetX, (int) targetY);
            targetPixel.setColor(sourcePixel.getColor());
        }
    }
    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.
3.3. METHODS THAT RETURN SOMETHING: COMPOSITING IMAGES

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.

```java
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 media. 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
3.3. METHODS THAT RETURN SOMETHING: COMPOSING IMAGES

Figure 3.4: Mini-collage created with scale and compose

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

```java
> 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.*; //to get to colors
> 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 #7**

**Example Java Code: Methods for general chromakey and bluescreen**

```java
/**
 * 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
 */
```
### 3.3. METHODS THAT RETURN SOMETHING: COMPOSITING IMAGES

![Figure 3.6: Chromakeying the monster into the jungle using different levels of bluescreening](image)

```java
public void chromakey(Picture target, Color bgColor, int threshold, int targetx, int targety)
{
    Pixel currPixel = null;
    Pixel newPixel = null;

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

            /* if the color at the current pixel is within threshold of
             * the input color, then don't copy the pixel */
            if (currPixel.colorDistance(bgColor)>threshold)
            {
                target.getPixel(trgx, trgy).setColor(currPixel.getColor());
            }
        }
    }
}
```
CHAPTER 3. METHODS IN JAVA: MANIPULATING PICTURES

/**
 * Method to do chromakey assuming blue background for background
 * 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 bluescreen(Picture target,
                      int targetx, int targety)
{
    Pixel currPixel = null;
    Pixel newPixel = null;

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

            /* if the color at the current pixel mostly blue (blue value is
             * greater than red and green combined), then don't copy pixel
             */
            if (currPixel.getRed() + currPixel.getGreen() > currPixel.getBlue())
            {
                target.getPixel(trgx, trgy).setColor(currPixel.getColor());
            }
        }
    }
}

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
3.4. CREATING CLASSES THAT DO SOMETHING

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

```java
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

```java
public class MyPicture {
    public static void main(String args[]){
        Picture canvas = new Picture(600,600);
        Picture swan = new Picture("C:/cs1316/MediaSources/swan.jpg");
        Picture rose = new Picture("C:/cs1316/MediaSources/rose.jpg");
        Picture turtle = new Picture("C:/cs1316/MediaSources/turtle.jpg");

        swan.scale(0.5).compose(canvas,10,10);
        swan.scale(0.5).compose(canvas,350,350);
        swan.flip().scale(0.5).compose(canvas,10,350);
        swan.flip().scale(0.5).compose(canvas,350,10);
        rose.scale(0.25).compose(canvas,200,200);
        turtle.scale(2.0).compose(canvas,10,200);
        canvas.show();
    }
}
```

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](image)

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 it’s 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
4.2. DRAWING WITH TURTLES

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).

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

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?

```java
> 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

```java
public class LotsOfTurtles { 

    public static void main(String[] args) {
        // Create a world
        World myWorld = new World();
        // A flotilla of turtles
        Turtle[] myTurtles = new Turtle[100];
    }
}
```

**Program Example #9**
CHAPTER 4. OBJECTS AS AGENTS: MANIPULATING TURTLES

```java
// Make a hundred turtles
for (int i = 0; i < 100; i++) {
    myTurtles[i] = new Turtle(myWorld);
}

// Tell them all what to do
for (int i = 0; i < 100; i++) {
    // Turn a random amount between 0 and 360
    myTurtles[i].turn((int) (360 * Math.random()));
    // Go 100 pixels
    myTurtles[i].forward(100);
}
```

**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

```java
> World tatooine = new World();
> Turtle r2d2 = new Turtle(tatooine);
> r2d2.turn((int) (360 * Math.random()));
> r2d2.forward((int) (Math.random() * 100));
> Turtle c3po = new Turtle(tatooine);
> 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).

```java
> 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).
CHAPTER 4. OBJECTS AS AGENTS: MANIPULATING TURTLES

Figure 4.4: Dropping the monster character

```java
> 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?

```java
> 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).
4.2. DRAWING WITH TURTLES

Figure 4.6: An iterated turtle drop of a monster

* * *

Example Java Code: Making a picture with dropped pictures

```java
public class MyTurtlePicture {

    public static void main(String[] args) {
        Picture canvas = new Picture(600,600);
        Turtle jenny = new Turtle(canvas);
        Picture lilTurtle = new Picture(
            FileChooser.getMediaPath("Turtle.jpg"));

        for (int i=0; i <=40; i++)
            if (i < 20)
                jenny.turn(20);
            else
                jenny.turn(-20);
            jenny.forward(40);
            jenny.drop(lilTurtle.scale(0.5));
        canvas.show();
    }
}
```

* * *
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 as 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
4.3. CREATING ANIMATIONS WITH TURTLES AND FRAMES

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).

```java
> 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

```java
public class MyTurtleAnimation {

private Picture canvas;
private Turtle jenny;
private FrameSequence f;

public MyTurtleAnimation() {
```

Program Example #11
canvas = new Picture(600,600);
jenny = new Turtle(canvas);
f = new FrameSequence("C:/Temp/");
}

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);
}

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

public void show(){
    f.show();
}

public void replay(int delay){
    f.show();
    f.replay(delay);
}

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 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:

```java
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.

```java
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
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 FrameSequence 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.
4.3. CREATING ANIMATIONS WITH TURTLES AND FRAMES
Chapter 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
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.

```java
> 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 #12**

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

```java
/**
 * Increase the volume of a sound
 **/  
public void increaseVolume(double factor){
    SoundSample [] samples = this.getSamples();
    SoundSample current = null;
    for (int i=0; i < samples.length; i++) {
        current = samples[i];
        current.setValue((int) (factor * current.getValue()));
    }
}
```

**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:

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

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

```java
/**
 * 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
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 #14**

Example Java Code: Create an audio collage

```java
public class MySoundCollage {
    public static void main(String [] args){
        Sound snap = new Sound(
            FileChooser.getMediaPath("snap-tenth.wav"));
    }
}
```
5.1. MANIPULATING SAMPLED SOUNDS

Sound drum = new Sound(
    FileChooser.getMediaPath("drumroll-1.wav"));
Sound clink = new Sound(
    FileChooser.getMediaPath("clink-tenth.wav"));
Sound clap = new Sound(
    FileChooser.getMediaPath("clap-q.wav"));

Sound drumRev = drum.reverse().scale(0.5);
Sound soundA = snap.append(clink).
    append(clink).append(clap).append(drumRev);
Sound soundB = clink.append(clap).
    append(clap).append(drum).append(snap).append(snap);

Sound collage = soundA.append(soundB).
    append(soundB).append(soundA).
    append(soundA).append(soundB);
collage.play();
}

Here is how some of these additional methods are coded.

Program Example #15

Example Java Code: **Append one sound with another**

```java
/**
 * Return this sound appended with the input sound
 * @param appendSound sound to append to this
 **/
public Sound append(Sound appendSound) {
    Sound target = new Sound(getLength()+appendSound.getLength());
    int sampleValue;

    // Copy this sound in
    for (int srcIndex=0, trgIndex=0;
        srcIndex < getLength();
        srcIndex++, trgIndex++)
    {
        sampleValue = this.getSampleValueAt(srcIndex);
        target.setSampleValueAt(trgIndex, sampleValue);
    }

    // Copy appendSound in to target
    for (int srcIndex=0, trgIndex=getLength();
        srcIndex < appendSound.getLength();
        srcIndex++, trgIndex++)
```
Program Example #16

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

```java
/**
 * Mix the input sound with this sound, with percent ratio of input.
 * Use mixIn sound up to length of this sound.
 * Return mixed sound.
 * @param mixIn sound to mix in
 * @param ratio how much of input mixIn to mix in
 **/
 public Sound mix(Sound mixIn, double ratio){
     Sound target = new Sound(getLength());
     int sampleValue, mixValue, newValue;

     // Copy this sound in
     for (int srcIndex=0, trgIndex=0;
         srcIndex < getLength() && srcIndex < mixIn.getLength();
         srcIndex++, trgIndex++)
     {
         sampleValue = this.getSampleValueAt(srcIndex);
         mixValue = mixIn.getSampleValueAt(srcIndex);
         newValue = (int)(ratio*mixValue) + (int)((1.0-ratio)*sampleValue);
         target.setSampleValueAt(trgIndex, newValue);
     }

     return target;
}
```

Program Example #17

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

```java
/**
 * Scale up or down a sound by the given factor
 * (1.0 returns the same, 2.0 doubles the length, and 0.5 halves the length)
 */
```
5.2. INSERTING AND DELETING IN AN ARRAY

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 isclink a test.” It might look like this:

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

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.
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:

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

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

Program Example #18

Example Java Code: **Inserting into the middle of sounds**
5.2. INSERTING AND DELETING IN AN ARRAY

```java
/**
 * insert the input Sound after the nth sample (input integer).
 * Modifies the given sound
 * @param insound Sound to insert
 * @param startIndex where to start inserting the new sound
 **/
public void insertAfter(Sound inSound, int startIndex) {
    SoundSample current = null;
    // Find how long insound is
    int amtToCopy = inSound.getLength();
    int endOfThis = this.getLength() - 1;
    if (startIndex + amtToCopy > endOfThis) {
        // If too long, copy only as much as will fit
        amtToCopy = endOfThis - startIndex - 1;
    } else {
        // If short enough, need to clear out room.
        // Copy from endOfThis-amtToCopy, moving backwards
        // (toward front of list) to start,
        // moving UP (toward back) to endOfThis
        // KEY INSIGHT: How much gets lost off the end of the
        // array? Same size as what we're inserting — amtToCopy
        for (int source = endOfThis - amtToCopy; source >= startIndex; source--) {
            // current is the TARGET — where we're copying to
            current = this.getSample(source + amtToCopy);
            current.setValue(this.getSampleValueAt(source));
        }
    }
    // NOW, copy in inSound up to amtToCopy
    for (int target = startIndex, source = 0;
         source < amtToCopy;
         target++, source++) {
        current = this.getSample(target);
        current.setValue(inSound.getSampleValueAt(source));
    }
}
```

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 grown of the data.
Part II

Linked Lists: Linear Lists of Media
6 Structuring Music

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;
> import jm.util.*;
> Note n = new Note(60,101);
Figure 6.1: Playing all the notes in a score

> 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 notes\(^1\).

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.

\(^1\)Taken from [http://www.phys.unsw.edu.au/~jw/notes.html](http://www.phys.unsw.edu.au/~jw/notes.html)
### 6.1. JMUSIC AND IMPORTS

#### Figure 6.2: Frequencies, keys, and MIDI notes

<table>
<thead>
<tr>
<th>Frequency</th>
<th>Keyboard</th>
<th>Name</th>
<th>MIDI number</th>
</tr>
</thead>
<tbody>
<tr>
<td>4186.0</td>
<td></td>
<td>C8</td>
<td>108</td>
</tr>
<tr>
<td>3951.1</td>
<td></td>
<td>B7</td>
<td>107</td>
</tr>
<tr>
<td>3920.0</td>
<td></td>
<td>A7</td>
<td>106</td>
</tr>
<tr>
<td>3840.0</td>
<td></td>
<td>G7</td>
<td>104</td>
</tr>
<tr>
<td>3799.0</td>
<td></td>
<td>F7</td>
<td>102</td>
</tr>
<tr>
<td>3750.0</td>
<td></td>
<td>E7</td>
<td>101</td>
</tr>
<tr>
<td>3300.0</td>
<td></td>
<td>D7</td>
<td>99</td>
</tr>
<tr>
<td>3277.0</td>
<td></td>
<td>C7</td>
<td>97</td>
</tr>
<tr>
<td>1760.0</td>
<td></td>
<td>B6</td>
<td>95</td>
</tr>
<tr>
<td>1600.0</td>
<td></td>
<td>A6</td>
<td>94</td>
</tr>
<tr>
<td>1200.0</td>
<td></td>
<td>G6</td>
<td>92</td>
</tr>
<tr>
<td>1108.1</td>
<td></td>
<td>F6</td>
<td>90</td>
</tr>
<tr>
<td>1046.5</td>
<td></td>
<td>E6</td>
<td>87</td>
</tr>
<tr>
<td>997.77</td>
<td></td>
<td>D6</td>
<td>86</td>
</tr>
<tr>
<td>932.33</td>
<td></td>
<td>C6</td>
<td>85</td>
</tr>
<tr>
<td>880.00</td>
<td></td>
<td>B5</td>
<td>83</td>
</tr>
<tr>
<td>824.03</td>
<td></td>
<td>A5</td>
<td>82</td>
</tr>
<tr>
<td>783.99</td>
<td></td>
<td>G5</td>
<td>80</td>
</tr>
<tr>
<td>739.99</td>
<td></td>
<td>F5</td>
<td>78</td>
</tr>
<tr>
<td>698.46</td>
<td></td>
<td>E5</td>
<td>76</td>
</tr>
<tr>
<td>659.26</td>
<td></td>
<td>D5</td>
<td>75</td>
</tr>
<tr>
<td>622.25</td>
<td></td>
<td>C5</td>
<td>73</td>
</tr>
<tr>
<td>597.53</td>
<td></td>
<td>B4</td>
<td>71</td>
</tr>
<tr>
<td>554.37</td>
<td></td>
<td>A4</td>
<td>69</td>
</tr>
<tr>
<td>523.25</td>
<td></td>
<td>G4</td>
<td>66</td>
</tr>
<tr>
<td>493.88</td>
<td></td>
<td>F4</td>
<td>63</td>
</tr>
<tr>
<td>466.16</td>
<td></td>
<td>E4</td>
<td>62</td>
</tr>
<tr>
<td>440.00</td>
<td></td>
<td>D4</td>
<td>60</td>
</tr>
<tr>
<td>415.31</td>
<td></td>
<td>C4</td>
<td>57</td>
</tr>
<tr>
<td>392.00</td>
<td></td>
<td>B3</td>
<td>55</td>
</tr>
<tr>
<td>369.99</td>
<td></td>
<td>A3</td>
<td>53</td>
</tr>
<tr>
<td>349.23</td>
<td></td>
<td>G3</td>
<td>51</td>
</tr>
<tr>
<td>329.63</td>
<td></td>
<td>F3</td>
<td>49</td>
</tr>
<tr>
<td>311.13</td>
<td></td>
<td>E3</td>
<td>47</td>
</tr>
<tr>
<td>297.18</td>
<td></td>
<td>D3</td>
<td>45</td>
</tr>
<tr>
<td>283.06</td>
<td></td>
<td>C3</td>
<td>43</td>
</tr>
<tr>
<td>266.26</td>
<td></td>
<td>B2</td>
<td>41</td>
</tr>
<tr>
<td>250.00</td>
<td></td>
<td>A2</td>
<td>39</td>
</tr>
<tr>
<td>234.92</td>
<td></td>
<td>G2</td>
<td>37</td>
</tr>
<tr>
<td>220.00</td>
<td></td>
<td>F2</td>
<td>35</td>
</tr>
<tr>
<td>207.65</td>
<td></td>
<td>E2</td>
<td>33</td>
</tr>
<tr>
<td>196.00</td>
<td></td>
<td>D2</td>
<td>31</td>
</tr>
<tr>
<td>185.00</td>
<td></td>
<td>C2</td>
<td>29</td>
</tr>
<tr>
<td>175.51</td>
<td></td>
<td>B1</td>
<td>27</td>
</tr>
<tr>
<td>166.84</td>
<td></td>
<td>A1</td>
<td>25</td>
</tr>
<tr>
<td>159.56</td>
<td></td>
<td>G1</td>
<td>23</td>
</tr>
<tr>
<td>152.59</td>
<td></td>
<td>F1</td>
<td>21</td>
</tr>
<tr>
<td>146.83</td>
<td></td>
<td>E1</td>
<td>19</td>
</tr>
<tr>
<td>139.59</td>
<td></td>
<td>D1</td>
<td>17</td>
</tr>
<tr>
<td>133.64</td>
<td></td>
<td>C1</td>
<td>15</td>
</tr>
<tr>
<td>128.22</td>
<td></td>
<td>B0</td>
<td>13</td>
</tr>
<tr>
<td>123.65</td>
<td></td>
<td>A0</td>
<td>11</td>
</tr>
</tbody>
</table>

Figure 6.2: Frequencies, keys, and MIDI notes
• We put the notes into the Phrase instance, and then notate and view the whole phrase.

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.

```java
> 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)
> phr
-------- 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)
```

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.

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:

```bash
Welcome to DrJava.
> import jm.music.data.*;
```
### Table 6.1: MIDI Program Numbers

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

6.3 Making a Simple Song Object

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

```java
import jm.music.data.*;
import jm.JMC;
import jm.util.*;
import jm.music.tools.*;

public class AmazingGraceSong {
    private Score myScore = new Score("Amazing Grace");

    public void fillMeUp() {
        myScore.setTimeSignature(3, 4);
        double[] phrase1data = {
            JMC.G4, JMC.QN,
            JMC.C5, JMC.HN, JMC.E5, JMC.EN, JMC.C5, JMC.EN,
            JMC.E5, JMC.HN, JMC.D5, JMC.QN,
            JMC.C5, JMC.HN, JMC.A4, JMC.QN,
            JMC.G4, JMC.HN, JMC.G4, JMC.EN, JMC.A4, JMC.EN,
            JMC.C5, JMC.HN, JMC.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);
        myPhrase.addNoteList(phrase2data);
        // create a new part and add the phrase to it
        Part aPart = new Part("Parts",
```
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 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):

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

### 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).
6.4. SIMPLE STRUCTURING OF NOTES WITH AN ARRAY

```java
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);
```

Figure 6.6: Trying the Amazing Grace song object

Figure 6.7: A hundred random notes
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.

**Program Example #20**

Example Java Code: Amazing Grace with Multiple Voices

```java
import jm.music.data.*;
import jm.JMC;
import jm.util.*;
import jm.music.tools.*;

public class MVAmazingGraceSong {
    private Score myScore = new Score("Amazing Grace");

    public Score getScore() {
        return myScore;
    }

    public void fillMeUp() {
        myScore.setTimeSignature(3,4);

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

        double[] phrase2data = 
            {JMC.G5, JMC.HN, JMC.E5, JMC.EN, JMC.G5, JMC.EN,
             JMC.G5, JMC.HN, JMC.E5, JMC.EN, JMC.C5, JMC.EN,
```
We can use this program like this (Figure 6.8:

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

**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 #21

Example Java Code: Amazing Grace as Song Elements

```java
import jm.music.data.*;
import jm.JMC;
import jm.util.*;
import jm.music.tools.*;
```
public class AmazingGraceSongElement {
    // Every element knows its next element and its part (of the score)
    private AmazingGraceSongElement next;
    private Part myPart;

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

    // addPhrase1 puts the first part of AmazingGrace into our part of the song
    // at the desired start time with the given instrument
    public void addPhrase1(double startTime, int instrument) {
        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.C5, JMC.DHN
        };

        Phrase myPhrase = new Phrase(startTime);
        myPhrase.addNoteList(phrase1data);
        this.myPart.addPhrase(myPhrase);
        // In MVAmingGraceSong, we did this when we initialized
        // the part. But we CAN do it later
        this.myPart.setInstrument(instrument);
    }

    public void addPhrase2(double startTime, int instrument) {
        double[] phrase2data = {
            JMC.G5, JMC.HN, JMC.E5, JMC.EN, JMC.G5, JMC.EN,
            JMC.G5, JMC.HN, JMC.E5, JMC.EN, JMC.C5, JMC.EN,
            JMC.E5, JMC.HN, JMC.D5, JMC.QN,
            JMC.C5, JMC.HN, JMC.A4, JMC.QN,
            JMC.G4, JMC.HN, JMC.G4, JMC.EN, JMC.A4, JMC.EN,
            JMC.C5, JMC.HN, JMC.E5, JMC.EN, JMC.C5, JMC.EN,
            JMC.E5, JMC.HN, JMC.D5, JMC.QN,
            JMC.C5, JMC.DHN
        };

        Phrase myPhrase = new Phrase(startTime);
        myPhrase.addNoteList(phrase2data);
        this.myPart.addPhrase(myPhrase);
        this.myPart.setInstrument(instrument);
// Here are the two methods needed to make a linked list of elements
public void setNext(AmazingGraceSongElement nextOne) {
    this.next = nextOne;
}

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

// We could just access myPart directly
// but we can CONTROL access by using a method
// (called an accessor)
// We'll use it in showFromMeOn
// (So maybe it doesn't need to be Public?)
public Part part() {
    return this.myPart;
}

// Why do we need this?
// If we want one piece to start after another, we need
// to know when the last one ends.
// Notice: It's the phrase that knows the end time.
// We have to ask the part for its phrase (assuming only one)
// to get the end time.
public double getEndTime() {
    return this.myPart.getPhrase(0).getEndTime();
}

// We need setChannel because each part has to be in its
// own channel if it has different start times.
// So, we'll set the channel when we assemble the score.
// (But if we only need it for showFromMeOn, we could
// make it PRIVATE...)
public void setChannel(int channel) {
    myPart.setChannel(channel);
}

public void showFromMeOn() {
    // Make the score that we'll assemble the elements into
    Score myScore = new Score("Amazing Grace");
    myScore.setTimeSignature(3, 4);

    // Each element will be in its own channel
    int channelCount = 1;

    // Start from this element (this)
    AmazingGraceSongElement current = this;
    // While we're not through...
while (current != null)
{
    // Set the channel, increment the channel, then add it in.
    current.setChannel(channelCount);
    channelCount = channelCount + 1;
    myScore.addPart(current.part());

    // Now, move on to the next element
    // which we already know isn’t null
    current = current.next();
}

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

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();

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 ??). “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.

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!
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

```java
import jm.music.data.*;
import jm.JMC;
import jm.util.*;
import jm.music.tools.*;

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

    // When we make a new element, the next part is empty, and ours is a blank new part
    public AmazingGraceSongElement2() {
        this.next = null;
        this.myPart = new Part();
    }

    // setPhrase takes a phrase and makes it the one for this element
    // at the desired start time with the given instrument
    public void setPhrase(Phrase myPhrase, double startTime, int instrument) {

        // Phrases get returned from phrase1() and phrase2() with default (0.0) startTime
        // We can set it here with whatever setPhrase gets as input
        myPhrase.setStartTime(startTime);
        this.myPart.addPhrase(myPhrase);
    }
}
```
// In MVAmazingGraceSong, we did this when we initialized
// the part. But we CAN do it later
this.myPart.setInstrument(instrument);
}

// First phrase of Amazing Grace
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.DN};
    Phrase myPhrase = new Phrase();
    myPhrase.addNoteList(phrase1data);
    return myPhrase;
}

public Phrase phrase2() {
    double[] phrase2data =
    {JMC.G5,JMC.HN,JMC.E5,JMC.EN,JMC.G5,JMC.EN,
     JMC.G5,JMC.HN,JMC.E5,JMC.EN,JMC.C5,JMC.EN,
     JMC.E5,JMC.HN,JMC.D5,JMC.QN,
     JMC.C5,JMC.HN,JMC.A4,JMC.QN,
     JMC.G4,JMC.HN,JMC.G4,JMC.EN,JMC.A4,JMC.EN,
     JMC.C5,JMC.HN,JMC.E5,JMC.EN,JMC.C5,JMC.EN,
     JMC.E5,JMC.HN,JMC.D5,JMC.QN,
     JMC.C5,JMC.DN};
    Phrase myPhrase = new Phrase();
    myPhrase.addNoteList(phrase2data);
    return myPhrase;
}

// Here are the two methods needed to make a linked list of elements
public void setNext(AmazingGraceSongElement2 nextOne){
    this.next = nextOne;
}

public AmazingGraceSongElement2 next(){
    return this.next;
}

// We could just access myPart directly
// but we can CONTROL access by using a method
6.6. MAKING ANY SONG SOMETHING TO EXPLORE

// (called an accessor)
private Part part()
{
    return this.myPart;
}

// Why do we need this?
// If we want one piece to start after another, we need
// to know when the last one ends.
// Notice: It's the phrase that knows the end time.
// We have to ask the part for its phrase (assuming only one)
// to get the end time.
public double getEndTime()
{
    return this.myPart.getPhrase(0).getEndTime();
}

// We need setChannel because each part has to be in its
// own channel if it has different start times.
// So, we'll set the channel when we assemble the score.
private void setChannel(int channel)
{
    myPart.setChannel(channel);
}

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

    // Each element will be in its own channel
    int channelCount = 1;

    // Start from this element (this)
    AmazingGraceSongElement2 current = this;
    // While we're not through...
    while (current != null)
    {
        // Set the channel, increment the channel, then add it in.
        current.setChannel(channelCount);
        channelCount = channelCount + 1;
        myScore.addPart(current.part());

        // Now, move on to the next element
        // which we already know isn't null
        current = current.next();
    }

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

```java
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.

```java
import jm.JMC;
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.
• Since current is not null (our right hand is holding something), we go ahead and process it.

• 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.
• This one isn’t null either, so we process it.

• 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.
6.6. MAKING ANY SONG SOMETHING TO EXPLORE

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

- 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!
• We check at the top of the loop—yup, we’re done.

Now let’s make a few observations about this code. Notice the `part2.phrase2()` expression. What would have happened if we did `part1.phrase2()` 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.DHIN};

    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.

Example Java Code: **General Song Elements and Song Phrases**

```
import jm.music.data.*;
import jm.JMC;
import jm.util.*;
import jm.music.tools.*;

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

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

    // setPhrase takes a phrase and makes it the one for this element
    // at the desired start time with the given instrument
    public void setPhrase(Phrase myPhrase, double startTime, int instrument){
        myPhrase.setStartTime(startTime);
        this.myPart.addPhrase(myPhrase);
        this.myPart.setInstrument(instrument);
    }

    // Here are the two methods needed to make a linked list of elements
    public void setNext(SongElement nextOne){
        this.next = nextOne;
    }

    public SongElement next(){
        return this.next;
    }
}
```

*Program Example #23*
CHAPTER 6. STRUCTURING MUSIC

// We could just access myPart directly
// but we can control access by using a method
// (called an accessor)
private Part part() {
    return this.myPart;
}

// Why do we need this?
// If we want one piece to start after another, we need
// to know when the last one ends.
// Notice: It's the phrase that knows the end time.
// We have to ask the part for its phrase (assuming only one)
// to get the end time.
public double getEndTime() {
    return this.myPart.getPhrase(0).getEndTime();
}

// We need setChannel because each part has to be in its
// own channel if it has different start times.
// So, we'll set the channel when we assemble the score.
private void setChannel(int channel) {
    myPart.setChannel(channel);
}

public void showFromMeOn() {
    // Make the score that we'll assemble the elements into
    // We'll set it up with a default time signature and tempo we like
    // (Should probably make it possible to change these — maybe with inputs?)
    Score myScore = new Score("My Song");
    myScore.setTimeSignature(3, 4);
    myScore.setTempo(120.0);

    // Each element will be in its own channel
    int channelCount = 1;

    // Start from this element (this)
    SongElement current = this;
    // While we're not through...
    while (current != null) {
        // Set the channel, increment the channel, then add it in.
        current.setChannel(channelCount);
        channelCount = channelCount + 1;
        myScore.addPart(current.part());

        // Now, move on to the next element
        // which we already know isn't null
        current = current.next();
    }
6.6. MAKING ANY SONG SOMETHING TO EXPLORE

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

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

public class SongPhrase {

// First phrase of Amazing Grace
static public Phrase AG1() {
    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.DEN};
    Phrase myPhrase = new Phrase();
    myPhrase.addNoteList(phrase1data);
    return myPhrase;
}

// Second phrase of Amazing Grace
static public Phrase AG2() {
    double[] phrase2data =
    {JMC.G5, JMC.HN, JMC.E5, JMC.EN, JMC.G5, JMC.EN,
     JMC.G5, JMC.HN, JMC.E5, JMC.EN, JMC.C5, JMC.EN,
     JMC.E5, JMC.HN, JMC.D5, JMC.QN,
     JMC.C5, JMC.HN, JMC.A4, JMC.QN,
     JMC.G4, JMC.HN, JMC.G4, JMC.EN, JMC.A4, JMC.EN,
     JMC.C5, JMC.HN, JMC.E5, JMC.EN, JMC.C5, JMC.EN,
     JMC.E5, JMC.HN, JMC.D5, JMC.QN,
     JMC.C5, JMC.DEN};
    Phrase myPhrase = new Phrase();
    myPhrase.addNoteList(phrase2data);
    return myPhrase;
}
}
We can use this like this:

```java
> 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 explo-

**Adding More Phrases**

*Program Example #24*

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

```java
import jm.music.data.*;
import jm.JMC;
import jm.util.*;
import jm.music.tools.*;

public class SongPhrase {

    // First phrase of Amazing Grace
    static public Phrase AG1() {
        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.EN};

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

    // Second phrase of Amazing Grace
    static public Phrase AG2() {
        double[] phrase2data =
        {JMC.G5, JMC.HN, JMC.E5, JMC.EN, JMC.G5, JMC.EN,
```

```
6.6. MAKING ANY SONG SOMETHING TO EXPLORE

```java
Phrase myPhrase = new Phrase();
myPhrase.addNoteList(phrase2data);
return myPhrase;
}

// House of the rising sun
static public Phrase house() {
    double[] phrasedata = {
        JMC.G5, JMC.HN, JMC.E5, JMC.EN, JMC.C5, JMC.EN,
        JMC.E5, JMC.HN, JMC.D5, JMC.QN,
        JMC.C5, JMC.HN, JMC.A4, JMC.QN,
        JMC.G4, JMC.HN, JMC.G4, JMC.EN, JMC.A4, JMC.EN,
        JMC.C5, JMC.HN, JMC.E5, JMC.EN, JMC.C5, JMC.EN,
        JMC.E5, JMC.HN, JMC.D5, JMC.QN,
        JMC.C5, JMC.DHN
    };
    Phrase myPhrase = new Phrase();
    myPhrase.addNoteList(phrasedata);
    return myPhrase;
}

// Little Riff1
static public Phrase riff1() {
    double[] phrasedata = {
        JMC.G3, JMC.EN, JMC.B3, JMC.EN, JMC.C4, JMC.EN, JMC.D4, JMC.EN
    };
    Phrase myPhrase = new Phrase();
    myPhrase.addNoteList(phrasedata);
    return myPhrase;
}

// Little Riff2
static public Phrase riff2() {
    double[] phrasedata =
```
CHAPTER 6. STRUCTURING MUSIC

{JMC.D4,JMC.EN,JMC.C4,JMC.EN,JMC.E4,JMC.EN,JMC.G4,JMC.EN};

Phrase myPhrase = new Phrase();
myPhrase.addNoteList(phrasedata);
return myPhrase;

// Little Riff3
static public Phrase riff3() {
  double[] phrasedata =
  {JMC.C4,JMC.QN,JMC.E4,JMC.EN,JMC.G4,JMC.EN,JMC.E4,JMC.EN,
   JMC.G4,JMC.EN,JMC.E4,JMC.EN,JMC.G4,JMC.EN,JMC.C4,JMC.QN};

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

// Little Riff4
static public Phrase riff4() {
  double[] phrasedata =
  {JMC.C4,JMC.QN,JMC.E4,JMC.QN,JMC.G4,JMC.QN,JMC.C4,JMC.QN};

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

> 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 playback
> 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 playback
> 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();

Figure 6.11: Playing some different riffs in patterns

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.

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

Program Example #25

Example Java Code: **Computed Phrases**

```java
// 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);
}

// 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);
  myPhrase.addNoteList(riff2data);
}

return myPhrase;
```
6.6. MAKING ANY SONG SOMETHING TO EXPLORE

} //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 playback
> 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).

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?

Program Example #26

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

```java
/*
   * 10 random notes
   **/
static public Phrase random() {
    Phrase ranPhrase = new Phrase();
    Note n = null;

    for (int i=0; i < 10; i++) {
      n = new Note((int) (128*Math.random()), 0.1);
```
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.

Example Java Code: 10 slightly less random notes

```java
/*
 * 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 Structuring 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 describe) 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 #28**

Example Java Code: **SongNode class**

```java
import jm.music.data.*;
import jm.JMC;
import jm.util.*;
```
6.7. STRUCTURING MUSIC

```java
import jm.music.tools.*;

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

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

    /**
     * setPhrase takes a Phrase and makes it the one for this node
     */
    public void setPhrase(Phrase thisPhrase){
        this.myPhrase = thisPhrase;
    }

    /**
     * Creates a link between the current node and the input node
     */
    public void setNext(SongNode nextOne){
        this.next = nextOne;
    }

    /**
     * Provides public access to the next node.
     */
    public SongNode next(){
        return this.next;
    }

    /**
     * Accessor for the node's Phrase
     */
    private Phrase getPhrase();
```
return this.myPhrase;
}

/**
 * Accessor for the notes inside the node's phrase
 * @return array of notes and durations inside the phrase
 */
private Note[] getNotes()
{
    return this.myPhrase.getNoteArray();
}

/**
 * Collect all the notes from this node on
 * in an part (then a score) and open it up for viewing.
 * @param instrument MIDI instrument (program) to be used in playing this list
 */
public void showFromMeOn(int instrument)
{
    // Make the Score that we'll assemble the elements into
    // We'll set it up with a default time signature and tempo we like
    // (Should probably make it possible to change these — maybe with inputs?)
    Score myScore = new Score("My Song");
    myScore.setTimeSignature(3,4);
    myScore.setTempo(120.0);

    // Make the Part that we'll assemble things into
    Part myPart = new Part(instrument);

    // Make a new Phrase that will contain the notes from all the phrases
    Phrase collector = new Phrase();

    // Start from this element (this)
    SongNode current = this;
    // While we're not through...
    while (current != null)
    {
        collector.addNoteList(current.getNotes());

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

    // Now, construct the part and the score.
    myPart.addPhrase(collector);
    myScore.addPart(myPart);

    // At the end, let's see it!
    View.notate(myScore);
}
We can use this new class to do some of the things that we did before (Figure 6.14).

```java
> 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!
> first.setNext(second); // OOPS!
Error: No 'next' method in 'SongNode' with arguments: (SongNode)
> first.setNext(second);
> first.showFromMeOn(JMC.PIANO);
```

Figure 6.14: First score generated from ordered linked list

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)

**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**

```java
/*
 * copyNode returns a copy of this node
 * @return another song node with the same notes
 */
public SongNode copyNode()
{
    SongNode returnMe = new SongNode();
    returnMe.setPhrase(this.getPhrase());
    return returnMe;
}

/**
 * Repeat the input phrase for the number of times specified.
 * It always appends to the current node, NOT insert.
 * @param nextOne node to be copied in to list
 * @param count number of times to copy it in.
 */
public void repeatNext(SongNode nextOne, int count) {
    SongNode current = this; // Start from here
    SongNode copy; // Where we keep the current copy
    for (int i=1; i <= count; i++)
    {
        copy = nextOne.copyNode(); // Make a copy
        current.setNext(copy); // Set as next
        current = copy; // Now append to copy
    }
}
```
public void insertAfter(SongNode nextOne)
{
    SongNode oldNext = this.next();  // Save its next
    this.setNext(nextOne);           // Insert the copy
    nextOne.setNext(oldNext);        // Make the copy point on the rest
}

public void weave(SongNode nextOne, int count, int skipAmount)
{
    SongNode current = this;          // Start from here
    SongNode copy;                    // Where we keep the one to be weaved in
    SongNode oldNext;                 // Need this to insert properly
    int skipped;                      // Number skipped currently

    for (int i=1; i <= count; i++)
    {
        copy = nextOne.copyNode();    // Make a copy

        //Skip skipAmount nodes
        skipped = 1;
        while ((current.next() != null) && (skipped < skipAmount))
        {
            current = current.next();
            skipped++;
        }

        if (current.next() == null)    // Did we actually get to the end early?
            break;                     // Leave the loop

        oldNext = current.next();      // Save its next
        current.insertAfter(copy);    // Insert the copy after this one
        current = oldNext;            // Continue on with the rest
    }
}
CHAPTER 6. STRUCTURING MUSIC

* * *

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

```java
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 ??).

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

Figure 6.17: Weaving a new node among the old

And we can keep weaving in more.

```java
SongNode another = new SongNode();
another.setPhrase(SongPhrase.rhythm1());
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.
Example Java Code: RepeatNextInserting

```java
/**
 * 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:

```java
// 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)!

Linked Lists versus Arrays
• 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.

```java
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); }
```
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**

```java
import jm.music.data.*; import jm.JMC; import jm.util.*; import jm.music.tools.*;

public class SongPart {

    /* SongPart has a Part */
    public Part myPart;

    /* SongPart has a SongNode that is the beginning of its
```
CHAPTER 6. STRUCTURING MUSIC

*/
public SongNode myList;

/**
 * Construct a SongPart
 * @param instrument MIDI instrument (program)
 * @param startNode where the song list starts from
 */
public SongPart(int instrument, SongNode startNode)
{
    myPart = new Part(instrument);
    myList = startNode;
}

/**
 * Collect parts of this SongPart
 */
public Phrase collect()
{
    return this.myList.collect(); // delegate to SongNode’s collect
}

/**
 * Collect all notes in this SongPart and open it up for viewing.
 */
public void show()
{
    // Make the Score that we’ll assemble the part into
    // We’ll set it up with a default time signature and tempo we like
    // (Should probably make it possible to change these — maybe with inputs?)
    Score myScore = new Score("My Song");
    myScore.setTimeSignature(3,4);
    myScore.setTempo(120.0);

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

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

Program Example #32

Example Java Code: Song class—root of a tree-like music structure
import jm.music.data.*; import jm.JMC; import jm.util.*; import jm.music.tools.*;

public class Song {
    /**
     * first Channel
     */
    public SongPart first;

    /**
     * second Channel
     */
    public SongPart second;

    /**
     * Take in a SongPart to make the first channel in the song
     */
    public void setFirst(SongPart channel1){
        first = channel1;
        first.myPart.setChannel(1);
    }

    /**
     * Take in a SongPart to make the second channel in the song
     */
    public void setSecond(SongPart channel2){
        second = channel2;
        first.myPart.setChannel(2);
    }

    public void show(){
        // Make the Score that we’ll assemble the parts into
        // We’ll set it up with a default time signature and tempo we like
        // (Should probably make it possible to change these — maybe with inputs?)
        Score myScore = new Score("My Song");
        myScore.setTimeSignature(3,4);
        myScore.setTempo(120.0);

        // Now, construct the part and the score.
        first.myPart.addPhrase(first.collect());
        second.myPart.addPhrase(second.collect());
        myScore.addPart(first.myPart);
        myScore.addPart(second.myPart);

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

* * *

Program Example #33

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

```java
import jm.music.data.*;
import jm.JMC;
import jm.util.*;
import jm.JMC;

public class MyFirstSong {
    public static void main(String[] args) {
        Song songroot = new Song();
        SongNode node1 = new SongNode();
        SongNode riff3 = new SongNode();
        riff3.setPhrase(SongPhrase.riff3());
        node1.repeatNext(riff3, 16);
        SongNode riff1 = new SongNode();
        riff1.setPhrase(SongPhrase.riff1());
        node1.weave(riff1, 7, 1);
        SongPart part1 = new SongPart(JMC.PIANO, node1);
        songroot.setFirst(part1);

        SongNode node2 = new SongNode();
        SongNode riff4 = new SongNode();
        riff4.setPhrase(SongPhrase.riff4());
        node2.repeatNext(riff4, 20);
        node2.weave(riff1, 4, 5);
        SongPart part2 = new SongPart(JMC.STEELDRUMS, node2);
        songroot.setSecond(part2);
        songroot.show();
    }
}
```

* * *
6.7. STRUCTURING MUSIC

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

Table 6.2: MIDI Drum Kit Notes
7 Linked Lists of Images

We know a lot about manipulating individual images. We know how to manipulate the pixels of an image to create various effect. 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).

```java
> Picture [] myarray = new Picture[5];
> myarray[0]=new Picture(FileChooser.getMediaPath("katie.jpg"));
> myarray[1]=new Picture(FileChooser.getMediaPath("barbara.jpg"));
> myarray[2]=new Picture(FileChooser.getMediaPath("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();
```

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).
Example Java Code: Elements of a scene in position order

```java
public class PositionedSceneElement {

    /**
     * the picture that this element holds
     */
    private Picture myPic;

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

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

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

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

    /**
     * Returns the picture in the node.
     * @return the picture in the node
     */
    public Picture getPicture() {
        return this.myPic;
    }

    /**
     * Method to draw from this node on in the list, using bluescreen.
     * Each new element has it’s lower–left corner at the lower–right
     * of the previous node. Starts drawing from left–bottom
     */
```
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).

```java
> PositionedSceneElement tree1 = new PositionedSceneElement(new Picture(FileChooser.getMediaPath("tree.jpg"));
> PositionedSceneElement tree2 = new PositionedSceneElement(new Picture(FileChooser.getMediaPath("tree2.jpg"));
> PositionedSceneElement tree3 = new PositionedSceneElement(new Picture(FileChooser.getMediaPath("tree3.jpg"));
> PositionedSceneElement doggy = new PositionedSceneElement(new Picture(FileChooser.getMediaPath("dog.jpg"));
> PositionedSceneElement house = new PositionedSceneElement(new Picture(FileChooser.getMediaPath("house.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");
```

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
7.2. LISTING THE PICTURES, LEFT-TO-RIGHT

Figure 7.3: Our first scene

the scene on a new background (Figure 7.4).

```java
> 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
CHAPTER 7. LINKED LISTS OF IMAGES

Generalizing moving the element
Let’s consider what happened in this line:

```java
> 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 #35**

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

```java
/** Method to remove node from list, fixing links appropriately.
 * @param node element to remove from list.
 */
public void remove(PositionedSceneElement node) {
    if (node == this) {
        System.out.println("I can’t remove the first node from the list.");
        return;
    }

    PositionedSceneElement current = this;
    // While 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(PositionedSceneElement node) {
      // Save what "this" currently points at
      PositionedSceneElement oldNext = this.getNext();
      this.setNext(node);
      node.setNext(oldNext);
  }

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).

![Figure 7.5: Removing the doggy from the scene](image)

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);

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*.

*Program Example #36*

Example Java Code: **LayeredSceneElements**

```java
public class LayeredSceneElement {

    /**
     * the picture that this element holds
     */
    private Picture myPic;

    /**
     * the next element in the list
     */
    private LayeredSceneElement next;

    /**
     * The coordinates for this element
     */
    private int x, y;

    /**
     * Make a new element with a picture as input, and
     * next as null, to be drawn at given x,y
     * @param heldPic Picture for element to hold
     */
}
```

Figure 7.6: Inserting the doggy into the scene
7.3. LISTING THE PICTURES, LAYERING

```java
/**
 * @param xpos x position desired for element
 * @param ypos y position desired for element
 **/
public LayeredSceneElement(Picture heldPic, int xpos, int ypos){
    myPic = heldPic;
    next = null;
    x = xpos;
    y = ypos;
}

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

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

/**
 * Returns the picture in the node.
 * @return the picture in the node
 **/
public Picture getPicture(){
    return this.myPic;
}

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

/**
 * Method to draw from this picture, using bluescreen.
 * @param bg Picture to draw drawing on
 **/
```
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.
7.3. LISTING THE PICTURES, LAYERING

```java
> Picture bg = new Picture(400,400);
> LayeredSceneElement tree1 = new LayeredSceneElement(
new Picture(FileChooser.getMediaPath("tree-blue.jpg")),10,10);
> LayeredSceneElement tree2 = new LayeredSceneElement(
new Picture(FileChooser.getMediaPath("tree-blue.jpg")),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.

![Figure 7.7: First rendering of the layered scene](image)

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 7.8) has completely different layering. The trees in the upper left have swapped, and the tree and dog are now in front of the house.

```java
> house.setNext(doggy); doggy.setNext(tree3); tree3.setNext(tree2); tree2.setNext(tree1);
> treel.setNext(null);
> bg = new Picture(400,400);
> house.drawFromMeOn(bg);
> bg.show();
> bg.write("D:/cs1316/second-layered-scene.jpg");
```

Figure 7.8: 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?

**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?

First, we need to create a seriously large scene. Let’s not do it in the Interactions Pane—it would take too long to recreate when we needed to. Let’s create a class just for our specific scene and put our messages there for creating 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.

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.

**7.4 Representing scenes with trees**

A list can only really represent a single dimension—either a linear placement on the screen, or a linear layering. A full scene has multiple dimensions. 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 typically represent more than just things that are to be drawn. They also represent operations on the scene, such as translations (moving the starting position for drawing the next list of elements) and rotations (changing the direction in which we’re drawing). Let’s use a Turtle to handle translations and rotations.

Here’s how we’ll do it:

- We need a new kind of Element class to represent things we’ll draw.
- We’ll also need Translation and Rotation elements.
• But then we have a Java problem. If we have three different kinds of elements, how do we put them all in a tree? How do we declare the variables representing the elements in the tree? Java gives us an out here—we’ll have all of the elements have the same kind of method for drawing, and we’ll define an Interface which represents that standardized method.

Trees have a property that they can be traversed in more than one way. While a list is traversed linearly, a tree can be traversed in several different ways. When the tree represents a scene, different traversals lead to different renderings—the scene looks different.

How would you create an animation in Java? One good answer is, "Modify your structure describing your picture, then render it again!" We’ll also be using linked lists and even graphs to create structures representing the flow of images representing a single character in motion.

7.5 Basic FrameSequence

We’ll use the utility class FrameSequence to do the basics of animation. We use FrameSequence by giving it a directory to write frames to. Each time we addFrame, we add a picture to the frame sequence. If you show the FrameSequence, you see the animation as it gets written out to frames frame0001.jpg, frame0002.jpg, and so on.

Here’s an example using some simple turtle graphics to create frames (Figure 7.9).

```java
> Picture p = new Picture(400,400);
> Turtle t = new Turtle(p);
> t
Unknown at 200, 200 heading 0
> t.forward(100);
> p.show();
> FrameSequence f = new FrameSequence("D:/movie");
> for (int i = 0; i < 100; i++)
   {t.forward(10);t.turn(36);f.addFrame(p);}
```
Exercises

1. Set up a scene with PositionedSceneElement, then change the layering of just a single element using remove and insertAfter.
Abstract Data Types: Separating the Use from the Implementation
Part III

Trees: Hierarchical Structures for Media
9 Trees of Images

Common Bug: When you run out of memory
Rendering large movies can easily eat up all available 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 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 I added to WolfAttackMovie to make it work like this:

```java
public static void main(String[] args){
    WolfAttackMovie wam = new WolfAttackMovie();
    wam.setUp();
    wam.renderAnimation();
    wam.replay();
}
```

Common Bug: When you run out of memory, within DrJava
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 9.1).

Figure 9.1: Reserving more memory for the Interactions Pane in DrJava’s Preferences pane
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 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.
11 Generalizing Lists and Trees

There's a lot of code in common between our different list and tree implementations. It's a good idea to pull out the common code into more abstract MMList (MultiMedia List) and MTree classes. There are a couple of reasons for creating such abstractions:

- It's wasteful to have the same code in different places. More importantly, it's hard to maintain. What if we found a better way to write some of that common code? To make the improvement everywhere involves updating several different classes. If the common code were in one and only one class, then we would have only one place to fix it.

- Once we have the abstract classes defined, it becomes easier to create new lists and trees in the future.

- Finally, computer scientists have studied the properties of abstract lists and trees. What they've learned can help us to use lists and trees to make our code more efficient.
Lists can loop—a later node can have as its next an earlier node. We use this sometimes to create circular lists, as when you are representing the cells in an animation (see *Mario Brothers*).

Trees can also loop. We call those kinds of trees graphs.
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 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.

14.1 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-
CHAPTER 14. INTRODUCING UML AND CONTINUOUS
SIMULATIONS

tion is to provide a framework in which to understand the simu-
lated 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 col-
llect 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 exe-
cuted (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 Regis-
trar 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 eas-
ier, but not just to build simulations. Alan Kay, who was one of the key
thinkers behind Smalltalk and object-oriented programming, had the in-
sight that simulations were a great way to think about all kinds of pro-
grams. 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.
14.1. INTRODUCING SIMULATIONS

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.

14.2 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 14.1). Wolves and deer is an instance of a common form of continuous simulation called predator and prey simulations.

Figure 14.1: 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 WolfDeerSimulation() wds.run() Timestep: 0 Wolves left: 5 Deer left: 20 ¡SIGH! A deer died... Timestep: 1 Wolves left: 5 Deer left: 19 ¡SIGH! A deer died... Timestep: 2 Wolves left: 5 Deer left: 18 ¡SIGH! A deer died... Timestep: 3 Wolves left: 5 Deer left: 19 ¡SIGH! A deer died...
```
14.2. OUR FIRST MODEL AND SIMULATION: WOLVES AND DEER

What we see is Figure 14.1. 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 14.2 describes the relationships in the Wolves and Deer simulation.

![Figure 14.2: The class relationships in the Wolves and Deer simulation](image)

- 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 14.2 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.

### 14.3 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 modeling 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 ?? 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-
CHAPTER 14. INTRODUCING UML AND CONTINUOUS SIMULATIONS

Figure 14.3: 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 14.4). 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 14.4: One UML class

Some fields may not appear in the class box. Instead, they might appear as a name on a reference relationship. Figure 14.5 pulls out just the
14.3. MODELLING IN OBJECTS

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 14.5: 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 ?? 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 14.6). The lines that have closed arrows at their ends are depicting generalization-specialization (gen-spec) relationships. The arrow points toward the generalization (superclass). Figure 14.6 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.
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.

### 14.4 Implementing the Simulation Class

The whole WolfDeerSimulation can be found at Program Program Example #38 (page 206). Let's walk through the key parts here.

```java
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.

```java
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 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 14.7 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 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.
Here's the next part of the run() method, where we invite our wolves and deer act.

```java
// 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.

```java
// 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.

### 14.5 Implementing a Wolf

The complete Wolf class can be found at Program Program Example #39 (page 209). Here’s how it starts.

```java
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;
```
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 subclasses.²

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.

²In 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.

/**
 * 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.

/**
 * Method to initialize the new wolf object
 */
public void init(WolfDeerSimulation thisSim) {
    // set the color of this wolf
    setColor(grey);
    // turn some random direction
CHAPTER 14. INTRODUCING UML AND CONTINUOUS SIMULATIONS

```
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?

```java
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`. 

```java
// turn (randNumGen.nextInt(360))
// set my simulation
mySim = thisSim;
```
14.5. IMPLEMENTING A WOLF

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().

```java
/**
 * 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 closestDeer = getClosest(30,
                                       (AgentNode) mySim.getDeer().getNext());

    if (closestDeer != null)
    {
        Deer thisDeer = (Deer) closestDeer.getAgent();
        this.moveTo(thisDeer.getXPos(),
                     thisDeer.getYPos());
        thisDeer.die();
    } // if we can’t eat, then move
    else // if the random number is > prob of NOT turning then turn
    {
        // 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 keep-
ing 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.

14.6 Implementing Deer

The Deer class is at Program Program Example #40 (page 212). 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 in-
stances 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);
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.

14.7 Implementing AgentNode

The full class AgentNode is at Program Program Example #41 (page 215). 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.

```java
/**
 * 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.next;

    while (current != null) {
        if (current.agent == myTurtle) {
            // If found the turtle, remove that node
            head.remove(current);
        }

        current = (AgentNode) current.next;
    }
}
```

In this method, we have a linked list traversal where we’re looking for the node whose agent is the input Turtle–current.agent == 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.
14.8 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 #42 (page 217).

```java
/**
 * 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.

```
public HungryWolf (ModelDisplay modelDisplay, WolfDeerSimulation thisSim) {
    super(modelDisplay, 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 modelDisplay the thing that displays the model
 * @param mySim my simulation
 */
public HungryWolf (int x, int y, ModelDisplay modelDisplay, WolfDeerSimulation thisSim) {
    // let the parent constructor handle it
    super(x, y, modelDisplay, 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 = MAX_SATISFIED;
}

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)
CHAPTER 14. INTRODUCING UML AND CONTINUOUS SIMULATIONS

```java
{ // 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 random 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).
CHAPTER 14. INTRODUCING UML AND CONTINUOUS SIMULATIONS

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:

```java
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:

```java
try {
    \ code that can throw the exception
} catch (FileNotFoundException ex) {
    System.err.println( "Exception: " + ex.getMessage());
    System.err.println( "StackTrace is: ");
    ex.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.

```java
try {
    \ code that can cause the exception
} catch (FileNotFoundException ex) {
    \ code to handle when the file isn’t found
```
14.8. EXTENDING THE SIMULATION

Putting it all together, here’s how you would read from a file in Java.

```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();
}

// handle exception
```

Now let’s add a file output capability to our simulation. We want to be able to do this:

```java
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.

```java
/* A BufferedWriter for writing to */
public BufferedWriter output;

/**
 * Constructor to set output to null
 **/
public WolfDeerSimulation() {
    output = null;
```
CHAPTER 14. INTRODUCING UML AND CONTINUOUS SIMULATIONS

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.

```java
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.

```java
// 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)
      {
         // Something went wrong closing the file
      }
      finally {
         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.
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.
16 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.

16.1 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
<table>
<thead>
<tr>
<th>Instrument</th>
<th>Description</th>
<th>MIDI Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAH</td>
<td>Breathnoise</td>
<td>EL_BASS</td>
</tr>
<tr>
<td>ABASS</td>
<td>Bright_Acoustic</td>
<td>EL_GUITAR</td>
</tr>
<tr>
<td>AC_GUITAR</td>
<td>Brightness</td>
<td>ELECTRIC_BASS</td>
</tr>
<tr>
<td>ACCORDION</td>
<td>Callope</td>
<td>ELECTRIC_GRAND</td>
</tr>
<tr>
<td>ACOUSTIC_BASS</td>
<td>Celesta</td>
<td>ELECTRIC_GUITAR</td>
</tr>
<tr>
<td>ACOUSTIC_GRAND</td>
<td>Celeste</td>
<td>ELECTRIC_ORGAN</td>
</tr>
<tr>
<td>ACOUSTIC_GUITAR</td>
<td>Cello</td>
<td>ELECTRIC_PIANO</td>
</tr>
<tr>
<td>AGOGO</td>
<td>CGuitar</td>
<td>EL_PIANO</td>
</tr>
<tr>
<td>AHHS</td>
<td>Charang</td>
<td>ENGLISH_HORN</td>
</tr>
<tr>
<td>ALTO</td>
<td>Chiffer</td>
<td>EPIANO</td>
</tr>
<tr>
<td>ALTO_SAX</td>
<td>Chiffer_Lead</td>
<td>EPIANO02</td>
</tr>
<tr>
<td>ALTO_SAXOPHONE</td>
<td>Choir</td>
<td>FANTASIA</td>
</tr>
<tr>
<td>APPLAUSE</td>
<td>Church_Organ</td>
<td>FBASS</td>
</tr>
<tr>
<td>ATMOSPHERE</td>
<td>Clar</td>
<td>FIDDLE</td>
</tr>
<tr>
<td>BAGPIPES</td>
<td>Clarinet</td>
<td>FINGERED_BASS</td>
</tr>
<tr>
<td>BAGPIPE</td>
<td>Clav</td>
<td>FLUTE</td>
</tr>
<tr>
<td>BAGPIPES</td>
<td>Clavinet</td>
<td>FRENCH_HORN</td>
</tr>
<tr>
<td>BANDNEON</td>
<td>Clean_Guitar</td>
<td>FRET</td>
</tr>
<tr>
<td>BANJO</td>
<td>Concertina</td>
<td>FRET_NOISE</td>
</tr>
<tr>
<td>BARI</td>
<td>Contra_Bass</td>
<td>FRETLESS</td>
</tr>
<tr>
<td>BARI_SAX</td>
<td>Contrabass</td>
<td>FRETLESS_BASS</td>
</tr>
<tr>
<td>BARITONE</td>
<td>Crystal</td>
<td>FRETNOISE</td>
</tr>
<tr>
<td>BARITONE_SAX</td>
<td>Cymbal</td>
<td>FRESTS</td>
</tr>
<tr>
<td>BARITONE_SAXOPHONE</td>
<td>DGuitar</td>
<td>GLOCK</td>
</tr>
<tr>
<td>BASS</td>
<td>Dist_Guitar</td>
<td>GLOCKENSPIEL</td>
</tr>
<tr>
<td>BASSOON</td>
<td>Distorted_Guitar</td>
<td>GMSAW_WAVE</td>
</tr>
<tr>
<td>BELL</td>
<td>Double_Bass</td>
<td>GMSQUARE_WAVE</td>
</tr>
<tr>
<td>BELLS</td>
<td>Drops</td>
<td>GOBLIN</td>
</tr>
<tr>
<td>BIRD</td>
<td>Drum</td>
<td>GT_HARMONICS</td>
</tr>
<tr>
<td>BOTTLE</td>
<td>DX_EPIANO</td>
<td>GUITAR</td>
</tr>
<tr>
<td>BOTTLE_BLOW</td>
<td>E Bass</td>
<td>GUITAR_HARMONICS</td>
</tr>
<tr>
<td>BOWED_GLASS</td>
<td>Echo</td>
<td>HALO</td>
</tr>
<tr>
<td>BRASS</td>
<td>Echo_Drop</td>
<td>HALO_PAD</td>
</tr>
<tr>
<td>BREATH</td>
<td>Echo_Drops</td>
<td>HAMMOND_ORGAN</td>
</tr>
</tbody>
</table>

Table A.1: JMusic constants in JMC for MIDI program changes, Part 1
<table>
<thead>
<tr>
<th>Instrument</th>
<th>JMusic Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HARMONICA</td>
<td>PANFLUTE SLAP</td>
<td>Harp</td>
</tr>
<tr>
<td>HARMONICS</td>
<td>P BASS SLAP BASS</td>
<td>Harp</td>
</tr>
<tr>
<td>HARP</td>
<td>PHONE SLOW STRINGS</td>
<td>Harp</td>
</tr>
<tr>
<td>HARPSICHORD</td>
<td>PIANO SOLO VOX</td>
<td>Harp</td>
</tr>
<tr>
<td>HELICOPTER</td>
<td>PIANO_ACCORDION SOP</td>
<td>Harp</td>
</tr>
<tr>
<td>HONKYTONK</td>
<td>PIC SOPRANO</td>
<td>Harp</td>
</tr>
<tr>
<td>HONKYTONK_PIANO</td>
<td>PICC SOPRANO_SAX</td>
<td>Harp</td>
</tr>
<tr>
<td>HORN</td>
<td>PICCOLO SOPRANO_SAXOPHONE</td>
<td>Harp</td>
</tr>
<tr>
<td>ICE_RAIN</td>
<td>PICKED_BASS SOUNDEFFECTS</td>
<td>Harp</td>
</tr>
<tr>
<td>ICERAIN</td>
<td>PIPE_ORGAN SOUNDFX</td>
<td>Harp</td>
</tr>
<tr>
<td>JAZZ_GUITAR</td>
<td>PIPES SPACE_VOICE</td>
<td>Harp</td>
</tr>
<tr>
<td>JAZZ_ORGAN</td>
<td>PITZ SPACE_VOICE</td>
<td>Harp</td>
</tr>
<tr>
<td>JGUITAR</td>
<td>PIZZ SQUARE</td>
<td>Harp</td>
</tr>
<tr>
<td>KALIMBA</td>
<td>PIZZICATO_STRINGS STAR_THEME</td>
<td>Harp</td>
</tr>
<tr>
<td>KOTO</td>
<td>POLY_SYNTH STEEL_DRUM</td>
<td>Harp</td>
</tr>
<tr>
<td>MARIMBA</td>
<td>POLYSYNTH STEEL_DRUMS</td>
<td>Harp</td>
</tr>
<tr>
<td>METAL_PAD</td>
<td>PSTRINGS STEEL_GUITAR</td>
<td>Harp</td>
</tr>
<tr>
<td>MGUITAR</td>
<td>RAIN STEELDRUM</td>
<td>Harp</td>
</tr>
<tr>
<td>MUSIC_BOX</td>
<td>RECORDER STEELDRUMS</td>
<td>Harp</td>
</tr>
<tr>
<td>MUTED_GUITAR</td>
<td>REED_ORGAN STR</td>
<td>Harp</td>
</tr>
<tr>
<td>MUTED_TRUMPET</td>
<td>REVERSE_CYMBAL STREAM</td>
<td>Harp</td>
</tr>
<tr>
<td>NGUITAR</td>
<td>RHODES STRINGS</td>
<td>Harp</td>
</tr>
<tr>
<td>NYLON_GUITAR</td>
<td>SAW SWEEP</td>
<td>Harp</td>
</tr>
<tr>
<td>OBOE</td>
<td>SAWTOOTH SWEEP_PAD</td>
<td>Harp</td>
</tr>
<tr>
<td>OCARINA</td>
<td>SAX SYN_CALLIOPE</td>
<td>Harp</td>
</tr>
<tr>
<td>O GUITAR</td>
<td>SAXOPHONE SYN_STRINGS</td>
<td>Harp</td>
</tr>
<tr>
<td>OOH</td>
<td>SBASS SYNTH_BASS</td>
<td>Harp</td>
</tr>
<tr>
<td>OOHS</td>
<td>SEA SYNTH_BRASS</td>
<td>Harp</td>
</tr>
<tr>
<td>ORCHESTRA_HIT</td>
<td>SEASHORE SYNTH_CALLIOPE</td>
<td>Harp</td>
</tr>
<tr>
<td>ORGAN</td>
<td>SFX SYNTH_DRUM</td>
<td>Harp</td>
</tr>
<tr>
<td>ORGAN2</td>
<td>SGUITAR SYNTH_DRUMS</td>
<td>Harp</td>
</tr>
<tr>
<td>ORGAN3</td>
<td>SHAKUHACHI SYNTH_STRINGS</td>
<td>Harp</td>
</tr>
<tr>
<td>OVERDRIVE_GUITAR</td>
<td>SHAMISEN SYNVOX</td>
<td>Harp</td>
</tr>
<tr>
<td>PAD</td>
<td>SHANNAI TAIKO</td>
<td>Harp</td>
</tr>
<tr>
<td>PAN_FLUTE</td>
<td>SITAR TELEPHONE</td>
<td>Harp</td>
</tr>
</tbody>
</table>

Table A.2: JMusic constants in JMC for MIDI program changes, Part 2
<table>
<thead>
<tr>
<th>Instrument</th>
</tr>
</thead>
<tbody>
<tr>
<td>TENOR</td>
</tr>
<tr>
<td>TENOR_SAX</td>
</tr>
<tr>
<td>TENOR_SAXOPHONE</td>
</tr>
<tr>
<td>THUMB_PIANO</td>
</tr>
<tr>
<td>THUNDER</td>
</tr>
<tr>
<td>TIMP</td>
</tr>
<tr>
<td>TIMPANI</td>
</tr>
<tr>
<td>TINKLE_BELL</td>
</tr>
<tr>
<td>TOM</td>
</tr>
<tr>
<td>TOM_TOM</td>
</tr>
<tr>
<td>TOM_TOMS</td>
</tr>
<tr>
<td>TOMS</td>
</tr>
<tr>
<td>TOMS</td>
</tr>
<tr>
<td>TUBULAR_BELL</td>
</tr>
<tr>
<td>TUBULAR_BELLS</td>
</tr>
<tr>
<td>VIBES</td>
</tr>
<tr>
<td>VIBRAPHONE</td>
</tr>
<tr>
<td>VIOLA</td>
</tr>
<tr>
<td>VIOLIN</td>
</tr>
<tr>
<td>VIOLIN CELLO</td>
</tr>
<tr>
<td>VOICE</td>
</tr>
<tr>
<td>VOX</td>
</tr>
<tr>
<td>WARM_PAD</td>
</tr>
<tr>
<td>WHISTLE</td>
</tr>
<tr>
<td>WIND</td>
</tr>
<tr>
<td>WOODBLOCK</td>
</tr>
<tr>
<td>WOODBLOCKS</td>
</tr>
<tr>
<td>XYLOPHONE</td>
</tr>
</tbody>
</table>
Utility Program

Utility #2: Turtle

```java
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);
}

}
Constructor to set output to null

```
public WolfDeerSimulation() {
    output = 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();
    }

    // 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());

    // 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;
        }
    }

    // Wait for one second
    //Thread.sleep(1000);
// 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;
    }
}
/** random number generator */
protected static Random randNumGen = new Random();

Constructors

/**
 * Constructor that takes the model display (the original
 * position will be randomly assigned)
 * @param modelDisplay thing that displays the model
 * @param thisSim my simulation
 */
public Wolf (ModelDisplay modelDisplay, WolfDeerSimulation thisSim)
{
 super(randNumGen.nextInt(modelDisplay.getWidth()),
 randNumGen.nextInt(modelDisplay.getHeight()),
 modelDisplay);
 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 modelDisplay the thing that displays the model
 * @param thisSim my simulation
 */
public Wolf (int x, int y, ModelDisplay modelDisplay, 
WolfDeerSimulation thisSim)
{
 super(x,y,modelDisplay);
 init(thisSim);
}

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
/**
 * Method to get the closest deer within the passed distance
 * to this wolf. We'll search the input list of the kind
 * of objects to compare to.
 * @param distance the distance to look within
 * @param list the list of agents to look at
 * @return the closest agent in the given distance or null
 */
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 = 999;
    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;
}

/**
 * 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());
APPENDIX B. WHOLE CLASS LISTINGS

Program Example #39

Example Java Code: Deer.java

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

/**
 * Class that represents a deer. The deer class
 * tracks all living deer with a linked list.
 * @author Barb Ericson ericson@cc.gatech.edu
 */
public class Deer extends Turtle {

        // fields

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

        /** class constant for probability of NOT turning */
        private static final double PROB_OF_STAY = 1/5;
```

```java
        if (closeDeer != null)
        {
            Deer thisDeer = (Deer) closeDeer.getAgent();
            this.moveTo(thisDeer.getXPos(),
                        thisDeer.getYPos());
            thisDeer.die();
        }

        else
        {
            // 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));
        }
        }
    }
```
/** class constant for how far deer can smell */
private static final double SMELL_RANGE = 50;

/** class constant for top speed (max num steps can move in a timestep) */
private static final int maxSpeed = 30;

/** random number generator */
private static Random randNumGen = new Random();

/** the simulation I’m in */
private WolfDeerSimulation mySim;

///////////////////////// Constructors /////////////

/**
 * Constructor that takes the model display (the original
 * position will be randomly assigned
 * @param modelDisplayer thing which will display the model
 */
public Deer (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
 */
public Deer (int x, int y, ModelDisplay modelDisplayer,
              WolfDeerSimulation thisSim)
{
    // let the parent constructor handle it
    super(x,y,modelDisplayer);
    init(thisSim);
}

////////////////// methods //////////////////////////

/**
 * Method to initialize the new deer object
 */
public void init(WolfDeerSimulation thisSim)
{
    // set the color of this deer
    setColor(brown);
// turn some random direction
this.turn(randNumGen.nextInt(360));

// know my simulation
mySim = thisSim;

/**
 * Method to get the closest wolf within the passed distance
 * to this deer. We'll search the input list of the kind
 * of objects to compare to.
 * @param distance the distance to look within
 * @param list the list of agents to look at
 * @return the closest agent in the given distance or null
 */
public AgentNode getClosest(double distance, AgentNode list)
{
    // get the head of the deer linked list
    AgentNode head = list;
    AgentNode curr = head;
    AgentNode closest = null;
    Wolf thisWolf;
    double closestDistance = 999;
    double currDistance = 0;

    // loop through the linked list looking for the closest deer
    while (curr != null)
    {
        thisWolf = (Wolf) curr.getAgent();
        currDistance = thisWolf.getDistance(this.getXPos(), this.getYPos());
        if (currDistance < distance)
        {
            if (closest == null || currDistance < closestDistance)
            {
                closest = curr;
                closestDistance = currDistance;
            }
        }
        curr = (AgentNode) curr.getNext();
    }
    return closest;
}

/**
 * Method to act during a time step
 * pick a random direction and move some random amount up to top speed
 */
public void act() {
    if (randNumGen.nextFloat() > PROB_OF_STAY) {
        this.turn(randNumGen.nextInt(360));
    }
    // go forward some random amount
    forward(randNumGen.nextInt(maxSpeed));
}

/**
 * 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"
    // getModelDisplay().remove(this);
    System.out.println("<SIGH!> A deer died...");
}

Example Java Code: AgentNode

/**
 * Class to implement a linked list of Turtle–like characters.
 * (Maybe "agents"?)
 ***/
public class AgentNode extends LLNode {
    /**
     * The Turtle being held
     ***/
    private Turtle myTurtle;
/** Two constructors: One for creating the head of the list 
 * , with no agent 
 **/ 
     public AgentNode() {super();}

/** 
 * One constructor for creating a node with an agent 
 **/ 
     public AgentNode(Turtle agent){
             super();
             this.setAgent(agent);
         }

/** 
 * Make a printable form 
 **/ 
     public String toString() {
             return "AgentNode with agent ("+myTurtle+"), and next: "+
             (AgentNode) getNext();
         }

/** 
 * Setter for the turtle 
 **/ 
     public void setAgent(Turtle agent){
             myTurtle = agent;
         }

/** 
 * Getter for the turtle 
 **/ 
     public Turtle getAgent(){return myTurtle;}

/** 
 * 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();
             }
Example Java Code: **HungryWolf**

```java
public class HungryWolf extends Wolf {

    private int satisfied;

    private static final int MAX_SATISFIED = 3;

    public HungryWolf (ModelDisplay modelDisplayer, WolfDeerSimulation thisSim) {
        super(modelDisplayer, thisSim);
    }

    public HungryWolf (int x, int y, ModelDisplay modelDisplayer, WolfDeerSimulation thisSim) {
        super(x, y, modelDisplayer, thisSim);
    }
}
```

Program Example #41
/**
 * Method to initialize the hungry wolf object
 */

public void init(WolfDeerSimulation thisSim) {
    super.init(thisSim);
    satisfied = MAX_SATISFIED;
}

/**
 * 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 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));
    }
}
Bibliography


Index

*/, 27
+++, 35, 79
--, 35, 79
/**/, 27

abstraction, 10
act(), 171
addFrame(aPicture), 67
addNote(), 31
AgentNode, 171
agents, 57, 169
aggregation, 172, 173
AmazingGraceSong, 92
AmazingGraceSongElement, 96
and, 21, 35
and (logical), 35
animations
  how they work, 66
API, 44
Application Program Interface, 44
array, 73
association relationship, 173

BBN Labs, 57
behavior, 9
big-Oh, 81
binary trees, 197
block, 21, 36
bluescreen(), 51
Bobrow, Danny, 57
Boolean, 60
break, 22
BufferedReader, 189
buttons, 163
bytes, 73

Call and response, 94
calls, 41
cascade, 50
cast, 42, 79
casting, 178
chaining, 189
Chromakey, 51
chromakey(), 51
class, 20, 23, 37
class diagram, 173
class method, 23
CLASSPATH, 155
classpath, 15
collaboration diagram, 173
comment, 30
compiled, 20
compression, 73
connections, 11
constant, 180
constructor, 34, 74, 79, 86
continuous simulations, 169
coordinated resource, 169
curly braces, 21

data structure
  properties, 11
data structures, 8, 9
declare, 20
declaring, 20
delegate, 69
deblegation, 69
die, 183
die(), 171
discourse rules, 34
discrete event simulations, 169, 197
dot notation, 23
double, 21, 43
DrJava, 15
drop, 63
edges, 11
Element, 147
else, 36
event loop, 172
event queue, 169
Exception, 190
exceptions, 189
expert musicians, 96
explore, 50
extends, 44
false, 60
Feurzeig, Wally, 57
field, 37, 42
fields, 34, 37, 174
FIFO list, 170
file paths, 23
FileChooser, 23
    getMediaPath, 23
    pickAFile, 23
    setMediaPath, 23
FileChooser.getMediaPath, 50
FileChooser.getMediapath, 23
FileChooser.setMediaPath, 23, 50
FileNotFoundException, 190
FileChooser.getMediaPath, 23, 50
getMediaPath, 23, 50
getSampleValueAt, 76
getValue(), 75
Goldberg, Adele, 168
ingraph, 11
graphical user interface (GUI), 163
graphs, 148, 161
GUI, 163
has-a, 173
head-rest, 177
head-tail, 177
hierarchy, 10
Hunchback of Notre Dame, The, 7
HungryWolf, 186
IDE, 15, 34
implementation inheritance, 175
import, 30, 85
increaseRed, 26
index variable, 39
information hiding, 44
inherits, 44
insertAfter, 126
instance variable, 37, 42
instance variables, 37
instances, 23, 30, 108
instrument, 86
Integrated Development Environment, 15
Interface, 148
interface, 168, 181
invokes, 41
jar file, 15
Java, 8
java, 55, 155
Java compiler, 155
Java Development Kit, 15
Java programming style, 34
javac, 155
JavaDoc, 44
Javadoc, 88, 123
Javanese, 61
JDK, 15
JMC, 86
JMC.C4, 86
JMC. FLUTE, 86
JMC.QN, 86
JMusic, 15, 30
Javadoc, 88
Kay, Alan, 19, 58, 168
kind-of relationships, 173
LayeredSceneElement, 144
layering, 142
layout managers, 163
length, 42
linked list, 10, 104, 135, 136
images, 136
music, 96
traversal, 104, 146
linked lists, 85
Lion King, The, 7
list, 14, 120, 136
lists, 163
literal, 21
Logo, 58
machine language, 20
main, 55, 132
Math.random(), 62, 92, 119, 180
matrix, 10, 39
memory, 12
memory model, 12
method, 26
method signature, 43
methods, 20, 33, 37
MIDI, 30
drum kit, 133
MIDI channels, 88
MIDI Drum Kit, 133
MIDI music, 73
MIDI note, 86
MIDI program, 86
MIT, 57
MMList, 159
model, 19, 96, 170
ordering, 96
modeled, 7
modelling, 170, 172
movies
how they work, 66
MTree, 159
Mufasa, 7
Musical Information Data Interchange, 30
mySim, 175
MySong, 132
MyTurtleAnimation, 68
new, 21
new Picture(), 24
nextFloat(), 184
node, 120
nodes, 11
normal distribution, 180
notate(), 31, 32
Note, 86
null, 23, 41
O(), 81
Object, 185
object, 19
object model, 172
object modelling, 172
object-oriented analysis, 172
object-oriented programming, 19
openFile(), 192
or, 21, 35
or (logical), 35
ordered list, 197
ordering, 96
OutOfBoundsException, 42
package, 39
panes, 163
Papert, Seymour, 57
Part, 96, 120, 133
PATH, 155
pausing, 179
penDown(), 60
penUp(), 60
INDEX

persistence of vision, 66
Person, 173
Phrase, 31, 86, 96, 133
pickAFile(), 23
Picture, 20
  creating blank, 49, 50
picture element, 39
pig Latin, 58
pile, 147
Pixel, 25
pixel, 39
Pixels [], 25
play(), 74
PositionedSceneElement, 138, 144
PowerPoint, 146
predator and prey, 170
print, 22
println, 96
printStackTrace(), 190
private, 27
properties, 11
property, 147
protected, 39, 180
  example of use, 180
public, 27, 39
public static void main, 55
queue, 169
Random, 180
random, 197
random number, 62
random values, 180
random(), 119
rarefaction, 73
recursion, 157
refactoring, 120
reference, 70
reference relationship, 173
references, 15
render, 148
rendering, 146
repeatNext, 126
repeatNextInserting, 126
replay(delay), 67
representation problem, 8
resource, 169
  coordinated, 169
responsibility-driven design, 19
return, 50
reverse(), 75
Robson, David, 168
Rotation, 147
rotations, 147
rsource
  fixed, 169
run(), 172, 177
Sample, 20
sampled sounds, 73
samples, 73
satisfied, 186
scale, 49
scene graph, 147
scenes, 135
scope, 24
Score, 88, 96, 133
self, 26
sequence, 96
sequence diagram, 173
setMediaPath, 23, 50
setPenDown(false), 60
setPenUp(true), 60
setSampleValueAt, 76
show, 40
  FrameSequence, 67
  Picture, 20
  show(), 20, 24, 67
  showFromMeOn, 104
SimplePicture, 44
Simula, 19, 168
simulation, 7
  event queue, 169
  resources, 169
simulations, 168
  continuous, 169
  defined, 168
  discrete event, 169
sleeping, 179
Smalltalk, 19, 58, 168
Song, 133
SongElement, 109, 120