Problem Solving with Data Structures:
A Multimedia Approach

Mark Guzdial
College of Computing/GVU
Georgia Institute of Technology

ALPHA VERSION OF TEXT

January 9, 2006
Copyright held by Mark Guzdial, 2006.
Dedicated to TBD.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>iii</td>
</tr>
<tr>
<td>List of Program Examples</td>
<td>v</td>
</tr>
<tr>
<td>List of Figures</td>
<td>vi</td>
</tr>
</tbody>
</table>

## I Introducing Modelling

1 Constructing the World

1.1 Making Representations of the World . . . . . . . . . . . . 8
1.2 Why Java? . . . . . . . . . . . . . . . . . . . . . . . . . . 14

2 Introduction to Java

2.1 What’s Java about? . . . . . . . . . . . . . . . . . . . . . . 19
2.2 Basic (Syntax) Rules of Java . . . . . . . . . . . . . . . . . 20
2.3 Manipulating Pictures in Java . . . . . . . . . . . . . . . . 23
2.4 Exploring Sound in Java . . . . . . . . . . . . . . . . . . . 28
2.5 Exploring Music in Java . . . . . . . . . . . . . . . . . . . . 29

3 Manipulating Pictures

3.1 Reviewing Java Basics . . . . . . . . . . . . . . . . . . . . . . 33
3.2 Methods for compositing images . . . . . . . . . . . . . . . . 36

4 Manipulating Turtles

4.1 Drawing with Turtles . . . . . . . . . . . . . . . . . . . . . . . 43

5 Manipulating Sounds

5.1 Sampled Sounds . . . . . . . . . . . . . . . . . . . . . . . . . . 47

## II Structuring Media

6 Structuring Music

6.1 JMusic and Imports . . . . . . . . . . . . . . . . . . . . . . . . 53
6.2 Starting out with JMusic . . . . . . . . . . . . . . . . . . . . . 58
6.3 Making a Simple Song Object .................................. 58
6.4 Simple structuring of notes with an array ................. 60
6.5 Making the Song Something to Explore .................. 61
6.6 Making Any Song Something to Explore ................. 68
6.7 Structuring Music .............................................. 81

7 Structuring Images .............................................. 95
  7.1 Simple arrays of pictures .................................. 95
  7.2 Listing the Pictures, Left-to-Right ....................... 95
  7.3 Listing the Pictures, layering ......................... 101
  7.4 Representing scenes with trees ......................... 107
  7.5 Basic FrameSequence .................................. 108

8 Structuring Sounds .......................................... 111

9 Generalizing Lists and Trees ................................ 113

10 Lists That Loop .............................................. 115

11 User Interface Structures ................................... 117

III Simulations ................................................... 119

12 Continuous Simulation ..................................... 121

13 Making the Villagers ....................................... 123

14 Discrete Event Simulation ................................ 125
  14.1 Distributions and Events ................................. 125

A MIDI Instrument names in JMusic ........................ 127
B Utility Classes ............................................... 131
Bibliography .................................................. 135
Index ........................................................... 136
List of Program Examples

Program Example #1: An Example Program .......................... 2
Program Example #2: Method to increase red in Picture .......... 26
Program Example #3: Method to flip an image .................... 27
Program Example #4: Method for Picture to scale by a factor ... 36
Program Example #5: Method to compose this picture into a target 37
Program Example #6: Methods for general chromakey and blue-screen ................................................................. 40
Program Example #7: Sound methods ............................... 47
Program Example #8: Amazing Grace as a Song Object ....... 58
Program Example #9: Amazing Grace with Multiple Voices .... 61
Program Example #10: Amazing Grace as Song Elements ...... 64
Program Example #11: Amazing Grace as Song Elements, Take 2 69
Program Example #12: General Song Elements and Song Phrases 72
Program Example #13: More phrases to play with ............... 75
Program Example #14: Computed Phrases ......................... 79
Program Example #15: SongNode class ........................... 83
Program Example #16: Repeating and weaving methods ....... 86
Program Example #17: SongPart class ............................. 89
Program Example #18: Song class–root of a tree-like music structure 90
Program Example #19: MySong class with a main method ....... 91
Program Example #20: Elements of a scene in position order ... 97
Program Example #21: Methods to remove and insert elements in a list ................................................................. 100
Program Example #22: LayeredSceneElements .................... 102
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 ......................................................... 24
2.2 Doubling the amount of red in a picture ......................... 25
2.3 Doubling the amount of red using our `increaseRed` method 27
2.4 Flipping our guy character–original (left) and flipped (right) 29
2.5 Just two notes .............................................................. 31

3.1 Using the `explore` method to see the sizes of the guy and the jungle 38
3.2 Composing the guy into the jungle ................................... 39
3.3 Chromakeying the monster into the jungle using different levels of bluescreening ............................................. 40

4.1 A drawing with a turtle ................................................ 43
4.2 Dropping the monster character .................................... 44
4.3 Dropping the monster character after a rotation ............. 45
4.4 An iterated turtle drop of a monster ............................... 45

6.1 Playing all the notes in a score ..................................... 53
6.2 Frequencies, keys, and MIDI notes ................................. 54
6.3 Viewing a multipart score ............................................. 57
6.4 JMusic documentation for the class `Phrase` .................... 57
6.5 Playing all the notes in a score ..................................... 58
6.6 Trying the Amazing Grace song object .......................... 60
6.7 A hundred random notes .............................................. 61
6.8 Multi-voice *Amazing Grace* notation ............................ 63
6.9 AmazingGraceSongElements with 3 pieces ................. 67
List of Figures

6.10 AmazingGraceSongElements with 3 pieces ........................................... 68
6.11 Playing some different riffs in patterns .............................................. 79
6.12 Sax line in the top part, rhythm in the bottom ...................................... 81
6.13 We now have layers of software, where we deal with only one at a time .......................................................... 82
6.14 First score generated from ordered linked list .................................... 85
6.15 Javadoc for the class SongNode ......................................................... 86
6.16 Repeating a node several times ............................................................ 88
6.17 Weaving a new node among the old ..................................................... 88
6.18 Multi-part song using our classes ......................................................... 92

7.1 Array of pictures composed into a background ..................................... 96
7.2 Elements to be used in our scenes ......................................................... 96
7.3 Our first scene ....................................................................................... 99
7.4 Our second scene .................................................................................. 99
7.5 Removing the doggy from the scene ..................................................... 101
7.6 Inserting the doggy into the scene ......................................................... 102
7.7 First rendering of the layered scene ..................................................... 105
7.8 Second rendering of the layered sene ................................................. 106
7.9 Three frames from the simple FrameSequence example ...................... 108
The current version of this text doesn't quite work as a book yet, though we're working on it. It's really meant to be the examples (with minimal explanatory text) and tables that can be a resource for CS1316 students in the first offerings of the course.

The focus in this book is on teaching data structures as a way to solve problems in modelling 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 modelling 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.

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 #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

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
Key computer science concepts appear like this.

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

Introducing Modelling
1 Constructing 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*

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

---

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](image)

• 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.

<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 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 next part of Katie’s bedroom note, and the location and next parts of the (new) Matthew’s bedroom note. No copying of data was necessary.

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

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

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

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

### 1.2 Why Java?

Why is this class taught in Java?

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


- You’ll need to tell DrJava about JM Music in order to access it. You use the Preferences in DrJava (see Figure 1.6) to add in the JM Music 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 JM Music 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?

Virtually 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.

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

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

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

    1
    2
    3
    4
    5
    6
    7
    8
    9
    10

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

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

    > for (int num = 1 ; num <= 10 ; num++)
        System.out.println(num);
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(chooser.getMediaPath("jungle.jpg"));
> myarray[0]=new Picture(chooser.getMediaPath("katie.jpg"));
> myarray[2]=new Picture(chooser.getMediaPath("barbara.jpg"));
> myarray[3]=new Picture(chooser.getMediaPath("flower1.jpg"));
> myarray[4]=new Picture(chooser.getMediaPath("flower2.jpg"));
> myarray[5]=new Picture(chooser.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"

New pictures don’t have any value – they’re null.

> Picture p;
> p
null
```

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)
\texttt{new Picture()}. Then we’ll have the picture show itself by telling it (using dot notation) to show() (Figure 2.1).

\begin{verbatim}
> p = new Picture("/Users/guzdial/cs1316/MediaSources/beach-smaller.jpg");
> p
\end{verbatim}

\begin{verbatim}
Picture, filename /Users/guzdial/cs1316/MediaSources/beach-smaller.jpg height 360 width 640
> p.show()
\end{verbatim}

![Figure 2.1: Showing a picture](image)

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

\textbf{Common Bug: One declaration per scope}
Within a given \textit{scope} (e.g., a single method, or the Interactions Pane in DrJava between compilations or reset), a variable can be declared once and only once.

\textbf{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](image)

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**

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)
    {
        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 COMPILE ALL button. If there are no errors, we can test our new method. When you compile your code, the objects and variables you had in the Interactions Pane disappear. You’ll have to recreate the objects you want.

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

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

You can see how this works in Figure 2.3.

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

![Figure 2.3: Doubling the amount of red using our increaseRed method](image)

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).

**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, trgy = 0; srcy < getHeight(); srcy++, trgy++)
        {
            // get the current pixel
            currPixel = this.getPixel(srcx, srcy);

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

    return target;
}

> 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();

**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(Chooser.pickAFile());
> s.play();
2.5. EXPLORING MUSIC IN JAVA

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

**[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?

```java
> 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.

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

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>Notes in the phrase.</td>
</tr>
<tr>
<td>Phrase</td>
<td>addNote(aNote)</td>
</tr>
</tbody>
</table>

Figure 2.5: Just two notes
3 Manipulating Pictures

Chapter Learning Objectives
The computer science goals for this chapter are:

• To be able to use Java with more 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 use JavaDoc.

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

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.

* * *
CHAPTER 3. MANIPULATING PICTURES

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. 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(inputs?). You may have already noted that sometimes you create a new class with inputs, and sometimes you don’t. That 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 those in as input and they were assembled into the new object.

Java has a couple 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 \( x++ \). There’s a general form, too. The phrase \( x = x + y \) can be shortened to \( x += y \).

Arrays

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

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

### Conditionals

You’ve seen already that conditionals look like this:

```java
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: `<`, `>`, `<=`, `>=` (where `==` is the test for equivalence, not the assignment operator `=`). Depending on the language that you took previously, you may have used the words and or 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:

```java
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.

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

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

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

A while loop looks like an if:

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

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

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

```
> p Picture, filename D:/cs1316/MediaSources/Swan.jpg height 360 width 480
> Pixel [] mypixels = p.getPixels();
> int index = 0;
> while (index < mypixels.length)
    {mypixels[index].setRed(0);
     index++;}
```

**How it works:** Notice the reference to mypixels.length above. This is the standard way of getting an array’s length. The expression .length isn’t referring to a method. Instead it’s referring to an instance variable or field. Every array knows an instance variable that provides its length.

### 3.2 Methods for compositing images

Here’s how we use the methods in class Picture to scale a picture larger (or smaller).

```
> Picture doll = new Picture(FileChooser.pickAFile());
> Picture bigdoll = doll.scale(2.0);
> bigdoll.show();
> bigdoll.write("bigdoll.jpg");
```

**Program Example #3**

**Method for Picture to scale by a factor**

```java
/**
 * Method to scale the picture by a factor, and return the result
 * @param scale factor to scale by (1.0 stays the same, 0.5 decreases each side by
 * @return the scaled picture
 */
```
3.2. METHODS FOR COMPOSITING IMAGES

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;
}

Let’s place our “guy” in the jungle. First, we’ll explore the pictures to figure out their sizes and where we want to compose them (Figure 3.1). We’ll use setMediaPath and getMediaPath to make it easier to get the jungle by name.

> FileChooser.setMediaPath("D:\cs1316\Mediasources\";
> Picture bg = new Picture(FileChooser.getMediaPath("jungle.jpg"));
> bg.explore();
> p.explore();

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 < getWidth();
    srcx++, trgx++)
{

    // loop through the rows
    for (int srcy=0, trgy = targety; srcy < getHeight();
        srcy++, trgy++)
    {

        // get the current pixel
        currPixel = this.getPixel(srcx, srcy);

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

Figure 3.1: Using the explore method to see the sizes of the guy and the jungle
3.2. METHODS FOR COMPOSING IMAGES

* * *

We can then compose the guy into the jungle like this (Figure 3.2).

```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 3.2: Composing the guy into the jungle](image)

**Common Bug: Don’t try to change the input variables**

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

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 3.3. If there’s a lot of blue in the character, it’s hard to get a threshold to work right

```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");

Figure 3.3: Chromakeying the monster into the jungle using different levels of bluescreening

Program Example #5

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
 */
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++)
    {
        // loop through the rows
```
3.2. METHODS FOR COMPOSING IMAGES

```java
for (int srcy = 0, trg y = targety; srcy < getHeight() && trg y < target.getHeight(); srcy++, trg y++)
{
    // 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(bg color) > threshold)
    {
        target.getPixel(trgx, trg y).setColor(currPixel.getColor());
    }
}

/**
   * 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, trg x = targetx;
        srcx < getWidth() && trg x < target.getWidth();
        srcx++, trg x++)
    {

        // loop through the rows
        for (int srcy = 0, trg y = targety;
            srcy < getHeight() && trg y < target.getHeight();
            srcy++, trg y++)
        {

            // 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());

4 Manipulating Turtles

4.1 Drawing with Turtles

We’re going to use turtles to draw on our pictures and to simplify animation. (See the Appendix for what the Turtle class looks like.) Here’s how we’ll use this class (Figure 4.1). 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.

```java
> 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")
```

Figure 4.1: A drawing with a turtle

How it works:
• Picture objects can be created as blank, with just a horizontal and vertical number of pixels.

• Positive turns are clockwise, and negative are counter-clockwise.

We can use turtles with pictures, through 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 4.2).

```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();
```

![Figure 4.2: Dropping the monster character](image)

We’ll rotate the turtle and drop again (Figure 4.3).

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

We can drop using loops and patterns, too (Figure 4.4). Why don’t we see 12 monsters here? I’m not sure – there may be limits to how much we can rotate.

```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);}
```
4.1. DRAWING WITH TURTLES

Figure 4.3: Dropping the monster character after a rotation

Figure 4.4: An iterated turtle drop of a monster
5 Manipulating Sounds

5.1 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 (see the next section) which encodes music (notes, durations, instrument selections) but not the sounds themselves.

```java
Sound s = new Sound(chooser.getMediaPath("gonga-2.wav"));
Sound s2 = new Sound(chooser.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
s.mix(s2, 0.25).play(); // Mix in the second sound
s.mix(s2.scale(0.5), 0.25).play(); // Mix in the second sound sped up
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();
```

Sound methods

```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);
```

Program Example #6
CHAPTER 5. MANIPULATING SOUNDS

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

```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
/**
 * 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++)
```
5.1. SAMPLED SOUNDS

```java
54
{ sampleValue = this.getSampleValueAt(srcIndex);
  mixValue = mixIn.getSampleValueAt(srcIndex);
  newValue = (int)(ratio*mixValue) + (int)((1.0-ratio)*sampleValue);
  target.setSampleValueAt(trgIndex,newValue);
}
return target;
```

```java
72
/**
 * 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)
 * @param factor ratio to increase or decrease
 **/
public Sound scale(double factor){
  Sound target = new Sound((int)(factor * (1+getLength())));
  int sampleValue;

  // Copy this sound in
  for(double srcIndex=0.0,trgIndex=0; srcIndex < getLength();
    srcIndex+=(1/factor),trgIndex++)
  {
    sampleValue = this.getSampleValueAt((int)srcIndex);
    target.setSampleValueAt((int)trgIndex,sampleValue);
  }
  return target;
}
```

**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.`
Part II

Structuring Media
6 Structuring 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);
> 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.

\[^1\text{Taken from http://www.phys.unsw.edu.au/~jw/notes.html}\]

Figure 6.1: Playing all the notes in a score
Figure 6.2: Frequencies, keys, and MIDI notes
6.1. JMusic and Imports

```java
> 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.

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

Table 6.1: MIDI Program numbers
6.1. JMUSIC AND IMPORTS

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

Figure 6.3: Viewing a multipart score

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

Figure 6.4: JMusic documentation for the class Phrase

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


6.2 Starting out with JMusic

Here’s what it looks like when you run:

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

![Figure 6.5: Playing all the notes in a score](image)

6.3 Making a Simple Song Object

Program Example #7

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);
    }
}
```
6.3. MAKING A SIMPLE SONG OBJECT

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};

double[] phrase2data =
{JMC.G5, JMC.HN, JMC.E5, JMC.EN, JMC.G5, JMC.EN,
 JMC.G5, JMC.HN, JMC.E5, JMC.EN, JMC.C5, JMC.EN,
 JMC.E5, JMC.HN, JMC.D5, JMC.QN,
 JMC.C5, JMC.HN, JMC.A4, JMC.QN,
 JMC.G4, JMC.HN, JMC.G4, JMC.EN, JMC.A4, JMC.EN,
 JMC.C5, JMC.HN, JMC.E5, JMC.EN, JMC.C5, JMC.EN,
 JMC.E5, JMC.HN, JMC.D5, JMC.QN,
 JMC.C5, JMC.DHN
};
Phrase myPhrase = new Phrase();
myPhrase.addNoteList(phrase1data);
myPhrase.addNoteList(phrase2data);
// Mod. repeat(aPhrase, repeats);
// create a new part and add the phrase to it
Part aPart = new Part("Parts",
 JMC.FLUTE, 1);
aPart.addPhrase(myPhrase);
// add the part to the score
myScore.addPart(aPart);

public void showMe(){
    View.notate(myScore);
}

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).
6.5. 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.

**Amazing Grace with Multiple Voices**

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

---

Figure 6.7: A hundred random notes

```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);
```
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.DHIN};
        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.DHIN};

        Phrase trumpetPhrase = new Phrase();
        trumpetPhrase.addNoteList(phrase1data); // 22.0 beats long
        double endphrase1 = trumpetPhrase.getEndTime();
        System.out.println("End of phrase1:"+endphrase1);
        trumpetPhrase.addNoteList(phrase2data);
        // create a new part and add the phrase to it
        Part part1 = new Part("TRUMPET PART",
            JMC.TRUMPET, 1);
        part1.addPhrase(trumpetPhrase);
        // add the part to the score
        myScore.addPart(part1);

        Phrase flutePhrase = new Phrase(endphrase1);
        flutePhrase.addNoteList(phrase1data); // 22.0 beats long
        flutePhrase.addNoteList(phrase2data); // optionally, remove this
        // create a new part and add the phrase to it
6.5. MAKING THE SONG SOMETHING TO EXPLORE

```java
Part part2 = new Part("FLUTE PART",
                      JMC.FLUTE, 2);
part2.addPhrase(flutePhrase);
// add the part to the score
myScore.addPart(part2);
}

public void showMe(){
  View.notate(myScore);
};
}
```

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

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

![Figure 6.8: Multi-voice Amazing Grace notation](image)

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 #9

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.G5,JMC.DHN};
```

Phrase myPhrase = new Phrase(startTime);
myPhrase.addNoteList(phrase1data);
this.myPart.addPhrase(myPhrase);
// In MVAmazingGraceSong, 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)
CHAPTER 6. STRUCTURING MUSIC

// 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.notaate(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;
6.5. MAKING THE SONG SOMETHING TO EXPLORE

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

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

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

![Figure 6.9: AmazingGraceSongElements with 3 pieces](image)

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 different instruments playing the same thing, but one measure (three beats) behind the previous? Try them out!

**Computer Science Idea**

**Computer Science Idea: Layering software makes it easier to change**

Notice that Phrase and Part has disappeared here. All that we’re manipulating are song elements. A good layer allows you to ignore the layers below.

### 6.6 Making Any Song Something to Explore

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

First, let’s create a second version (cunningly called AmazingGraceSongElement2) where there is only one addPhrase method, but you decide which phrase you want as an input. We’ll also clean up some of our protections here, while we’re revising.
Amazing Grace as Song Elements, Take 2

```
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.C4, JMC.GN,
         JMC.C5, JMC.HN, JMC.E5,JMC.EN, JMC.C5,JMC.EN,
         JMC.E5,JMC.HN,JMC.D5,JMC.QN,
         JMC.C5,JMC.HN,JMC.A4,JMC.QN,
         JMC.G4,JMC.HN,JMC.G4,JMC.EN,JMC.A4,JMC.EN,
         JMC.C5,JMC.HN,JMC.E5,JMC.EN,JMC.C5,JMC.EN,
         JMC.E5,JMC.HN,JMC.D5,JMC.EN,JMC.E5,JMC.EN,
         JMC.G5,JMC.DHN};

        Phrase myPhrase = new Phrase();
        myPhrase.addNoteList(phrase1data);
        return myPhrase;
    }
```
public Phrase phrase2() {
    // Define the phrase data
    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.EN,
        JMC.G5, JMC.HN, JMC.G4, 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
    };

    // Create the phrase
    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
// (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);
}
6.6. MAKING ANY SONG SOMETHING TO EXPLORE

```java
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();
```

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

That doesn't really make a lot of sense, does it, in terms of what each object should know? Does every song element object need to know how
CHAPTER 6. STRUCTURING MUSIC

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:

```java
// 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.DH4N};

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

We’d actually use this method like this:

```java
> 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?

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

**Program Example #11**

**General Song Elements and Song Phrases**

```java
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;
```
6.6. MAKING ANY SONG SOMETHING TO EXPLORE

// 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
public void setPhrase(Phrase myPhrase, double startTime, int instrument)
{
    myPhrase.setStart_time(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;
}

// 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 is 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();
}

// 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.DN};
6.6. MAKING ANY SONG SOMETHING TO EXPLORE

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

We can use this like this:
> import jm.JMC;
> SongElement part1 = new SongElement();
> part1.setPhrase(SongPhrase.AG1(),0.0,JMC.FLUTE);
> SongElement part2 = new SongElement();
> part2.setPhrase(SongPhrase.AG2(),22.0,JMC.PIANO);
> part1.setNext(part2);
> part1.showFromMeOn();

We now have a structure to do more songs and more general explo-

Adding More Phrases

Program Example #12

More phrases to play with

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.DHN
        };

        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.DHN
        };

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

    // House of the rising sun
    static public Phrase house() {
        double[] phrasedata = {
            JMC.E4, JMC.EN, JMC.A3, JMC.HN, JMC.B3, JMC.EN, JMC.A3, JMC.EN,
            JMC.C4, JMC.HN, JMC.D4, JMC.EN, JMC.D4, JMC.EN,
            JMC.E4, JMC.HN, JMC.C4, JMC.EN, JMC.B3, JMC.EN,
            JMC.A3, JMC.HN, JMC.E4, JMC.QN,
            JMC.A4, JMC.HN, JMC.E4, JMC.QN,
            JMC.G4, JMC.HN, JMC.E4, JMC.EN, JMC.D4, JMC.EN, JMC.E4, JMC.DHN,
            JMC.E4, JMC.HN, JMC.GS4, JMC.EN, JMC.G4, JMC.EN,
            JMC.A4, JMC.HN, JMC.A3, JMC.QN,
            JMC.C4, JMC.EN, JMC.E4, JMC.DQN, JMC.E4, JMC.QN,
            JMC.E4, JMC.EN, JMC.E4, JMC.EN, JMC.E4, JMC.QN, JMC.C4, JMC.EN, JMC.B3, JMC.EN,
            JMC.A3, JMC.HN, JMC.E4, JMC.QN,
        };
    }
JMC.E4, JMC.HN, JMC.E4, JMC.EN,
JMC.E4, JMC.EN, JMC.G3, JMC.QN, JMC.C4, JMC.EN, JMC.B3, JMC.EN,
JMC.A3, 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 =
    {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.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();
> riff1.setNext(riff2);
> riff2.getEndTime() 2.0
> riff2.setNext(riff4);
> riff4.setNext(riff5);
> riff5.setNext(riff6);
> riff6.setNext(riff7);
> riff7.setNext(riff1);
>
> 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();
steel.setPhrase(SongPhrase.riff1(), 0.0, JMC.STEEL_DRUM);
steel.showFromMeOn();
```

**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);
    }
```

**Figure 6.11: Playing some different riffs in patterns**
return myPhrase;
}

// Larger Riff2
static public Phrase pattern2() {
    double[] riff1data = {JMC.G3, JMC.EN, JMC.B3, JMC.EN, JMC.C4, JMC.EN, JMC.D4, JMC.EN};
    double[] riff2data = {JMC.D4, JMC.EN, JMC.C4, JMC.EN, JMC.E4, JMC.EN, JMC.E4, 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);

    return myPhrase;
}

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

    int counter1;
    int counter2;

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

    return myPhrase;
}

> import jm.JMC;
> SongElement sax1 = new SongElement();
> sax1.setPhrase(SongPhrase.pattern1(), 0.0, JMC.TENOR_SAX);
6.7. STRUCTURING MUSIC

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

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

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

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

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

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

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 el-
ement 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.

**SongNode class**

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

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

   // 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
   * @param thisPhrase the phrase for this node
   */
   public void setPhrase(Phrase thisPhrase)
   { 
      this.myPhrase = thisPhrase;
   }

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

   /* Provides public access to the next node.
   */
} Program Example #14
```
CHAPTER 6. STRUCTURING MUSIC

* @return a SongNode instance (or null)
  */
public SongNode next()
{
  return this.next;
}

/*
 * Accessor for the node's Phrase
 * @return internal 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();
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!
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](image)

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 doc-
CHAPTER 6. STRUCTURING MUSIC

Documentation from that structure. The commenting structure is: (XXX-TO-
DO See DrJava docs for now.) (Figure 6.15

Figure 6.15: Javadoc for the class SongNode

Now Let's Play!

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

Program Example #15

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  
    }  
}  
/**  
* Insert the input SongNode AFTER this node,  
* and make whatever node comes NEXT become the next of the input node.  
* @param nextOne SongNode to insert after this one  
*/  
public void insertAfter(SongNode nextOne) {  
    SongNode oldNext = this.next(); // Save its next  
    this.setNext(nextOne); // Insert the copy  
    nextOne.setNext(oldNext); // Make the copy point on to the rest  
}  
/**  
* Weave the input phrase count times every skipAmount nodes  
* @param nextOne node to be copied into the list  
* @param count how many times to copy  
* @param skipAmount how many nodes to skip per weave  
*/  
public void weave(SongNode nextOne, int count, int skipAmount) {  
    SongNode current = this; // Start from here  
    SongNode copy; // Where we keep the one to be weaved in  
    SongNode oldNext; // Need this to insert properly  
    int skipped; // Number skipped currently  
    for (int i=1; i <= count; i++) {  
        copy = nextOne.copyNode(); // Make a copy  
        // Skip skipAmount nodes  
        skipped = 1;  
        while ((current.next() != null) && (skipped < skipAmount)) {  
            current = current.next();  
            skipped++;  
        }  
    }  
}
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.
> SongNode another = new SongNode();
> another.setPhrase(SongPhrase.rhythm1());
> first.weave(another,10,2);
> first.showFromMeOn(JMC.STEEL_DRUMS);

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.

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
     */
    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
    }
}
```

Program Example #16
CHAPTER 6. STRUCTURING MUSIC

/
* 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 #17

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);
    }
}
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).

**MySong class with a main method**

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

Program Example #18
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();
    }
}

Figure 6.18: Multi-part song using our classes

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

**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 *SongPart* s. 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>Acoustic Bass Drum</th>
<th>51</th>
<th>Ride Cymbal 1</th>
</tr>
</thead>
<tbody>
<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>Vibraphone</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 Structuring Images

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

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

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

7.1 Simple arrays of pictures

The simplest thing to do is to simply list all the pictures we want in an 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.
Figure 7.1: Array of pictures composed into a background

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

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

Figure 7.2: Elements to be used in our scenes

* * *
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
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
7.2. LISTING THE PICTURES, LEFT-TO-RIGHT

tree3 to point at the house (what the doggy used to point at). Then redraw 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 7.3: Our first scene

Figure 7.4: Our second scene
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 #20**

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.  
    */
```
7.3. LISTING THE PICTURES, LAYERING

```java
/**
 * 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:

```java
> 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):

```java
> 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 #21

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
     */
```
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 its 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
 **/
### CHAPTER 7. STRUCTURING IMAGES

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

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

    LayeredSceneElement current = this;
    // While there are more nodes to consider
    while (current.getNext() != null) {
        if (current.getNext() == node) {
            // Simply make node’s next be this next
            current.setNext(node.getNext());
            // Make this node point to nothing
            node.setNext(null);
            return;
        }
        current = current.getNext();
    }
}

/**
 * Insert the input node after this node.
 * @param node element to insert after this.
 **/
public void insertAfter(LayeredSceneElement node) {
    // Save what "this" currently points at
    LayeredSceneElement oldNext = this.getNext();
    this.setNext(node);
    node.setNext(oldNext);
}
```

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.
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.

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);
> tree1.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](image)

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.
8 Structuring Sounds

The same structures that we used for images can also be used for sounds.

- We can create lists of sounds that, by rendering (traversing), we can generate music pieces. Changing the music pieces is pretty easy within the list. We can use the weaving and repeating methods that we developed for music here. We might even use lists to make wholesale changes in music, e.g., replace all snaps with pops.

- At this point, you might be wondering, “Do we have to go to all that trouble? Do we have to use lists? How about just using arrays like we used to?” Let’s recreate our list of sound elements using arrays instead. We’ll find that it’s do-able but not easy. Linked lists offer us more flexibility.

- Finally, let’s construct a tree of sound elements, like our tree of picture elements. Again, different traversals lead to different renderings, where a rendering here means a different sounding piece.
9 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.
10 Lists That Loop

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.
11 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 III

Simulations
Simulations are representations of the world (models) that are executed (made to behave like things in the world). Continuous simulations represent every moment of the simulated world.

We’ll explore a few different kinds of continuous simulations here. We’ll use our Turtle class to represent individuals in our simulated worlds.

- A common form of continuous simulation is *predator and prey* simulations, like the way wolves and deer interact.

- We can create more sophisticated simulations, too. We might simulate the spread of disease (or ideas, or political influence).

- One of the critical factors in any simulation is access to *resources*. We need to be able to represent how agents in the simulation *queue* to take turns at a resource.
13 Making the Villagers

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.
14 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.

14.1 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>Description1</th>
<th>Description2</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAH</td>
<td>BREATHTNOISE</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>ELPIANO</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>EPiano2</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>FRETS</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>BASSESSOON</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>EBASS</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>MIDI Program Change 1</th>
<th>MIDI Program Change 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>HARMONICA</td>
<td>PANFLUTE</td>
<td>SLAP</td>
</tr>
<tr>
<td>HARMONICS</td>
<td>PBASS</td>
<td>SLAP_BASS</td>
</tr>
<tr>
<td>HARP</td>
<td>PHONE</td>
<td>SLOW_STRINGS</td>
</tr>
<tr>
<td>HARPSICHORD</td>
<td>PIANO</td>
<td>SOLO_VOX</td>
</tr>
<tr>
<td>HELICOPTER</td>
<td>PIANO_ACCORDION</td>
<td>SOP</td>
</tr>
<tr>
<td>HONKYTEONK</td>
<td>PIC</td>
<td>SOPRANO</td>
</tr>
<tr>
<td>HONKYTEONK_PIANO</td>
<td>PICC</td>
<td>SOPRANO_SAX</td>
</tr>
<tr>
<td>HORN</td>
<td>PICCOLO</td>
<td>SOPRANO_SAX</td>
</tr>
<tr>
<td>ICE_RAIN</td>
<td>PICKED_BASS</td>
<td>SOUNDEFFECTS</td>
</tr>
<tr>
<td>ICERAIN</td>
<td>PIPE_ORGAN</td>
<td>SOUNDFX</td>
</tr>
<tr>
<td>JAZZ_GUITAR</td>
<td>PIPES</td>
<td>SPACE_VOICE</td>
</tr>
<tr>
<td>JAZZ_ORGAN</td>
<td>PIZZ</td>
<td>SPACE_VOICE</td>
</tr>
<tr>
<td>JGUITAR</td>
<td>PIZZ</td>
<td>SQUARE</td>
</tr>
<tr>
<td>KALIMBA</td>
<td>PIZZICATO_STRINGS</td>
<td>STAR_THEME</td>
</tr>
<tr>
<td>KOTO</td>
<td>POLY_SYNTH</td>
<td>STEEL_DRUM</td>
</tr>
<tr>
<td>MARIMBA</td>
<td>POLYSYNTH</td>
<td>STEEL_DRUMS</td>
</tr>
<tr>
<td>METAL_PAD</td>
<td>PSTRINGS</td>
<td>STEEL_GUITAR</td>
</tr>
<tr>
<td>MGUITAR</td>
<td>RAIN</td>
<td>STEELDRUM</td>
</tr>
<tr>
<td>MUSIC_BOX</td>
<td>RECORDER</td>
<td>STEELDRUMS</td>
</tr>
<tr>
<td>MUTED_GUITAR</td>
<td>REED_ORGAN</td>
<td>STR</td>
</tr>
<tr>
<td>MUTED_TRUMPET</td>
<td>REVERSE_CYMBAL</td>
<td>STREAM</td>
</tr>
<tr>
<td>NGUITAR</td>
<td>RHODES</td>
<td>STRINGS</td>
</tr>
<tr>
<td>NYLON_GUITAR</td>
<td>SAW</td>
<td>SWEEP</td>
</tr>
<tr>
<td>OBOE</td>
<td>SAWTOOTH</td>
<td>SWEEP_PAD</td>
</tr>
<tr>
<td>OCARINA</td>
<td>SAX</td>
<td>SYN_CALLIOPE</td>
</tr>
<tr>
<td>OGUITAR</td>
<td>SAXOPHONE</td>
<td>SYN_STRINGS</td>
</tr>
<tr>
<td>OOH</td>
<td>SBASS</td>
<td>SYNTH_BASS</td>
</tr>
<tr>
<td>OOHS</td>
<td>SEA</td>
<td>SYNTH_BRASS</td>
</tr>
<tr>
<td>ORCHESTRA_HIT</td>
<td>SEASHORE</td>
<td>SYNTH_CALLIOPE</td>
</tr>
<tr>
<td>ORGAN</td>
<td>SFX</td>
<td>SYNTH_DRUM</td>
</tr>
<tr>
<td>ORGAN2</td>
<td>SGUITAR</td>
<td>SYNTH_DRUMS</td>
</tr>
<tr>
<td>ORGAN3</td>
<td>SHAKUHACHI</td>
<td>SYNTH_STRINGS</td>
</tr>
<tr>
<td>OVERDRIVE_GUITAR</td>
<td>SHAMISEN</td>
<td>SYNVOX</td>
</tr>
<tr>
<td>PAD</td>
<td>SHANNAI</td>
<td>TAIKO</td>
</tr>
<tr>
<td>PAN_FLUTE</td>
<td>SITAR</td>
<td>TELEPHONE</td>
</tr>
</tbody>
</table>

Table A.2: JMusic constants in JMC for MIDI program changes, Part 2
| TENOR | TENOR_SAX | TENOR_SAXOPHONE | THUMB_PIANO | THUNDER | TIMP | TIMPANI | TINKLE_BELL | TOM | TOM_TOM | TOM_TOMS | TOMS | TOMS | TREMOLO | TREMOLO_STRINGS | TROMBONE | TRUMPET | TUBA | TUBULAR_BELL | TUBULAR_BELLS | VIBES | VIBRAPHONE | VIOLA | VIOLIN | VIOLIN_CELLO | VOICE | VOX | WARM_PAD | WHISTLE | WIND | WOODBLOCK | WOODBLOCKS | XYLOPHONE |

Table A.3: JMusic constants in JMC for MIDI program changes, Part 3
B Utility Classes

Utility #2: Turtle

UTILITY PROGRAM

/*******************************************************************************/
 * Creates a Turtle on an input
 *
/*******************************************************************************/

import java.awt. *;
import java.awt.event. *;
import java.awt.geom. *;
import javax.swing. *;
import java.awt.image. *;

public class Turtle {

private Picture myPicture; // the picture that we’re drawing on
private Graphics2D myGraphics;
JFrame myWindow;

private double x = 0.0, y = 0.0; // turtle is at coordinate (x, y)
private int height, width;
private double heading = 180.0; // facing this many degrees counterclockwise
private Color foreground = Color.black; // foreground color
private boolean penDown = true;

// turtles are created on pictures
public Turtle(Picture newPicture) {
    myPicture = newPicture;
    myGraphics = (Graphics2D) myPicture.getBufferedImage().createGraphics();
    myGraphics.setColor(foreground);
}
APPENDIX B. UTILITY CLASSES

```java
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, (int) w, (int) h, 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);
}
Bibliography
Index

, 35
*/, 26
++, 34, 49
–, 49
*/, 26

abstraction, 10
addNote(), 31
AmazingGraceSong, 59
AmazingGraceSongElement, 64
and, 21, 35
and (logical), 35

behavior, 9
block, 21, 35
bluescreen, 39
break, 22

Call and response, 61
cast, 49
chromakey, 39
class, 20, 23
class method, 23
classpath, 15
comment, 30
compiled, 20
connections, 11
constructor, 34, 49, 53
curly braces, 21

data structure
   properties, 11
data structures, 8, 9
declare, 20
declaring, 20
discourse rules, 34
dot notation, 23
double, 21
DrJava, 15
drop, 44

drop, 44
edges, 11
Element, 107
ever, 35
expert musicians, 64
explore, 37
field, 36
fields, 34
file paths, 23
FileChooser, 23
for, 22
functions, 20

getMediaPath, 37
graph, 11

hierarchy, 10
Hunchback of Notre Dame, The,
   7

IDE, 15, 34
import, 29, 53
increaseRed, 25
instance variable, 36
instances, 23, 29, 72
instrument, 53, 55
Integrated Development Environment,
   15

jar file, 15
Java, 8
show(), 24
Simula, 19
simulation, 7
Smalltalk, 19
Song, 93
SongElement, 72, 82
SongNode, 82
SongPart, 89
SongPhrase, 72, 89
Sound, 20, 49
square brackets, 23
Stack, 107
stack, 107
static, 23, 72
stream, 63
string concatenation, 63
structure, 9
System.out.println, 22
System.out.println(), 63

table, 10
The Hunchback of Notre Dame, 7
The Lion King, 7
this, 26
Toy Story, 7
Translation, 107
translations, 107
traverses, 106
tree, 10, 83, 89
Turtle, 107
types, 20

View, 31
Visio, 106
void, 26

WAV files, 47
while, 22, 36
wildebeests, 7

x++, 34