Course Notes for Representing Structure and Behavior: Multimedia Data Structures in Java

Mark Guzdial

College of Computing/GVU
Georgia Institute of Technology

Prentice Hall

PRENTICE HALL, *Upper Saddle River, New Jersey 07458*

Dedicated to TBD.

Contents

Contents **iv**

I Introducing Modelling **1**

1 Constructing the World **2**
1.1 Things to do to get started 4

2 Introduction to Java **7**
2.1 Basic (Syntax) Rules of Java 7
2.1.1 Declarations and Types 7
2.1.2 Assignment 8
2.1.3 Conditionals 9
2.1.4 Iteration 9
2.1.5 Arrays 10
2.2 Manipulating Pictures in Java 10
2.3 Drawing with Turtles 21
2.4 Sampled Sounds 22
2.5 JMusic and Imports 26

II Structuring Media **35**

3 Structuring Music **36**
3.1 Starting out with JMusic 36
3.2 Making a Simple Song Object 37
3.3 Simple structuring of notes with an array 38
3.4 Making the Song Something to Explore 40
3.5 Making Any Song Something to Explore 46
3.5.1 Adding More Phrases 53
3.5.2 Computing phrases 56
3.6 Structuring Music 60
3.6.1 Now Let's Play! 63
3.6.2 Creating a Music Tree 66

4 Structuring Images **72**
4.1 Simple arrays of pictures 72
4.2 Listing the Pictures, Left-to-Right 72
4.2.1 Generalizing moving the element 76
4.3 Listing the Pictures, layering 78
4.3.1 Reversing a List 83
4.4 Representing scenes with trees 84

5 Structuring Sounds **85**

6 Generalizing Lists and Trees **86**

7 User Interface Structures **87**

8 Objects in Graphics: Animation **88**
8.1 Basic FrameSequence 88

III Simulations 89

9 Continuous Simulation **90**

10 Discrete Event Simulation **91**
10.1 Distributions and Events 91

APPENDICES

A Utility Classes **92**

List of Figures

1.1 Katie's list of treasure hunt clues 3
1.2 An organization chart 3
1.3 A map of a town 4
1.4 Opening the DrJava Preferences 5
1.5 Adding the JMusic libraries to DrJava in Preferences 5
1.6 Adding book classes to DrJava 6
2.1 Parts of DrJava window 7
2.2 Showing a picture 11
2.3 Doubling the amount of red in a picture 12
2.4 Doubling the amount of red using our `increaseRed` method 14
2.5 Flipping our guy character–original (left) and flipped (right) 15
2.6 Using the `explore` method to see the sizes of the guy and the jungle 17
2.7 Composing the guy into the jungle 18
2.8 Chromakeying the monster into the jungle using different levels of bluescreening 19
2.9 A drawing with a turtle 21
2.10 Dropping the monster character 22
2.11 Dropping the monster character after a rotation 23
2.12 An iterated turtle drop of a monster 23
2.13 Playing all the notes in a score 26
2.14 Frequencies, keys, and MIDI notes 27
2.15 Viewing a multipart score 30
2.16 JMusic documention for the class `Phrase` 34
3.1 Playing all the notes in a score 36
3.2 Trying the Amazing Grace song object 39
3.3 A hundred random notes 39
3.4 Multi-voice *Amazing Grace* notation 42
3.5 AmazingGraceSongElements with 3 pieces 45
3.6 AmazingGraceSongElements with 3 pieces 46
3.7 Playing some different riffs in patterns 57
3.8 Sax line in the top part, rhythm in the bottom 59
3.9 We now have layers of software, where we deal with only one at a time 59
3.10 First score generated from ordered linked list 63
3.11 Javadoc for the class `SongNode` 64
3.12 Repeating a node several times 66
3.13 Weaving a new node among the old 66
3.14 Multi-part song using our classes 70
4.1 Array of pictures composed into a background 73
4.2 Elements to be used in our scenes 73
4.3 Our first scene 76

4.4 Our second scene 76
4.5 Removing the doggy from the scene 78
4.6 Inserting the doggy into the scene 79
4.7 First rendering of the layered sene 82
4.8 Second rendering of the layered sene 83
8.1 Three frames from the simple `FrameSequence` example 88

PART ONE

INTRODUCING MODELLING

Chapter 1 Constructing the World

Chapter 2 Introduction to Java

CHAPTER 1

Constructing the World

1.1 THINGS TO DO TO GET STARTED

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?

The really amazing 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 paintings? That's hard to say–"I don't know what art is, but I know what I like."

But are there better and worse *representations*? That's easier. 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 staff gets a new person, the second representation makes it easier to write the new person's name in.

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 something new to computers. There are lots of examples of data structuring and the use of representations in your daily life.

- My daughter, Katie, likes to create treasure hunts for the family, where she hides notes in various rooms (Figure 1.1). Each note references the next note in the list. This is an example of a *linked list*. Think about some of the advantages of this structure: the pieces work as a structure 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).

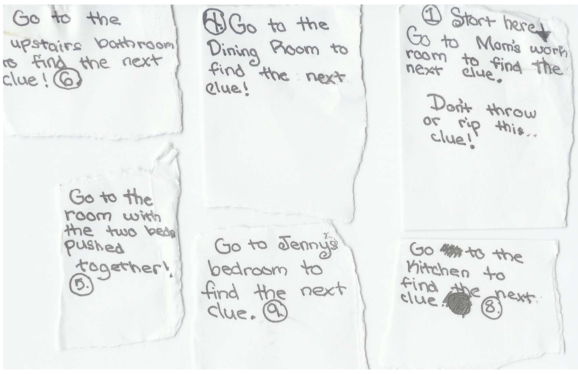

FIGURE 1.1: Katie's list of treasure hunt clues

- An organization chart (Figure 1.2) 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*.

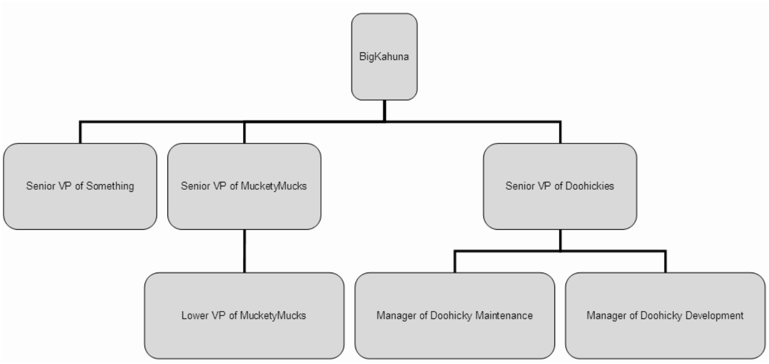

FIGURE 1.2: An organization chart

- A map (Figure 1.3) 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 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*.

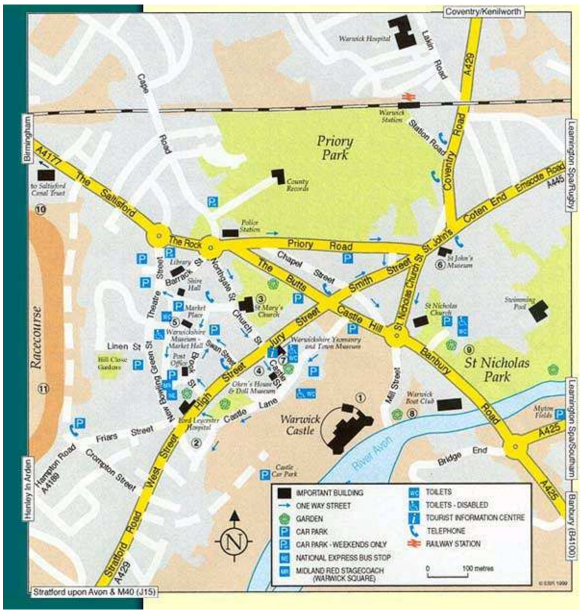

FIGURE 1.3: A map of a town

1.1 THINGS TO DO TO GET STARTED

- Download and install *DrJava* from `http://www.drjava.org`.
- Download and install *JMusic* from `http://jmusic.ci.qut.edu.au/`.
- You'll need to tell DrJava about JMusic in order to access it. You use the Preferences in DrJava (see Figure 1.4) to add in the JMusic *jar file* and the instruments (Figure 1.5).
- Make sure that you grab the `MediaSources` and `bookClasses` from the CD or the website.

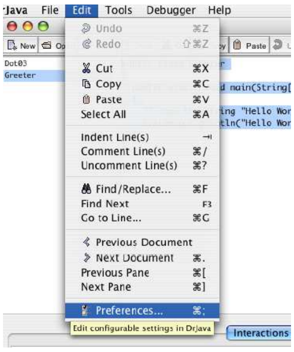

FIGURE 1.4: Opening the DrJava Preferences

FIGURE 1.5: Adding the JMusic libraries to DrJava in Preferences

- Just as you added JMusic to your DrJava preferences, add the `bookClasses` folder to your preferences, too. That way, you'll be able to access the classes there immediately. As you create additional classes, store them in the same folder, so that you'll have easy access to your new classes, too. (Figure 1.6).

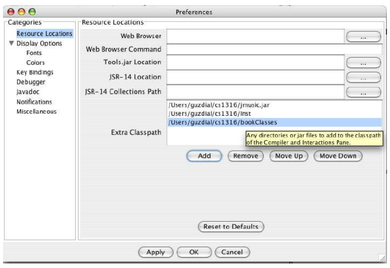

FIGURE 1.6: Adding book classes to DrJava

CHAPTER 2

Introduction to Java

2.1 BASIC (SYNTAX) RULES OF JAVA
2.2 MANIPULATING PICTURES IN JAVA
2.3 DRAWING WITH TURTLES
2.4 SAMPLED SOUNDS
2.5 JMUSIC AND IMPORTS

Once you start DrJava, you'll have a screen that looks like Figure 2.1.

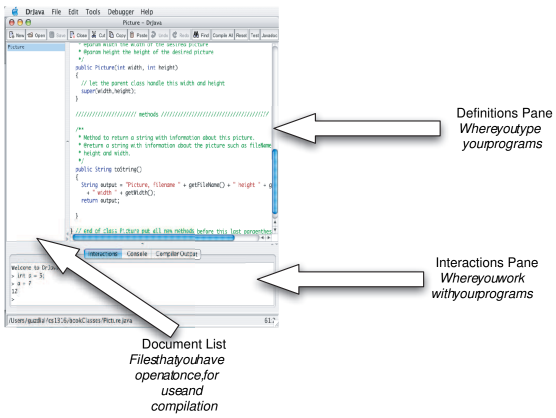

FIGURE 2.1: Parts of DrJava window

2.1 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.

2.1.1 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.

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

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

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

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

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

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

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

There are strings, too.

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

2.1.2 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.

2.1.3 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
```

2.1.4 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);
```

2.1.5 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.

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

2.2 MANIPULATING PICTURES IN JAVA

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

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

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

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

To make a new picture, we use the code (you'll never guess this one) `new Picture()`. Then we'll have the picture show itself by telling it (using dot notation) to `show()` (Figure 2.2).

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

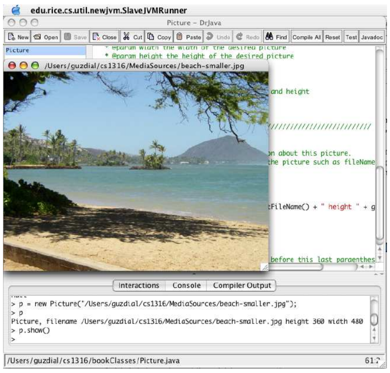

FIGURE 2.2: Showing a picture

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 2.3).

```
> 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()
```

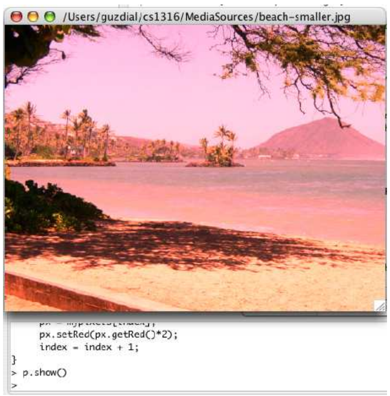

FIGURE 2.3: Doubling the amount of red in a picture

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

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

Program 1: Method to increase red in Picture

```
/**
 * 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.

You can see how this works in Figure 2.4.

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

Later on, we're going to want to have characters moving to the left or to the right. We'll probably only want to create one of these (left or right), then flip it for the other side. Let's create the method for doing that. Notice that this method

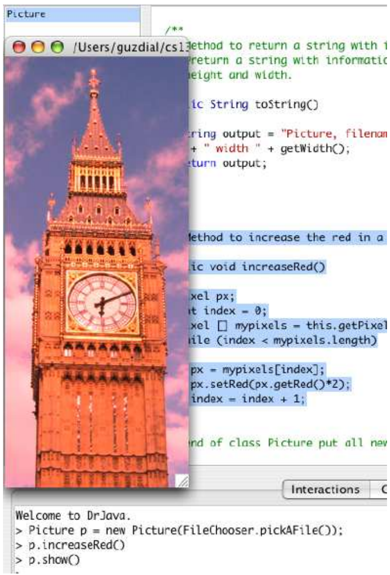

FIGURE 2.4: Doubling the amount of red using our `increaseRed` method

returns a *new* picture, not modifying the original one. We'll see later that that's pretty useful (Figure 2.5).

Program 2: Method to flip an image

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

FIGURE 2.5: Flipping our guy character–original (left) and flipped (right)

Common Bug: Width is the size, not the coordinate
Why did we subtract one from `getWidth()` (which defaults to `this.getWidth()` to set the target X coordinate (`trgx`)? `getWidth()` returns the *number of pixels* across the picture. But the last *coordinate* in the row is one less than that, because Java starts all arrays at *zero*. Normal everyday counting starts with one, and that's what `getWidth()` reports.

Scaling a picture larger.

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

Program 3: Method for Picture to scale by a factor

```
/**
 * 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 0.5, 2
 * @return the scaled picture
 */
public Picture scale(double factor)
{
  Pixel sourcePixel, targetPixel;
  Picture canvas = new Picture((int) (factor*this.getWidth())+1,
                               (int) (factor*this.getHeight())+1);
  // loop through the columns
  for (double sourceX = 0, targetX=0;
       sourceX < this.getWidth();
       sourceX+=(1/factor), targetX++)
  {
    // loop through the rows
    for (double sourceY=0, targetY=0;
         sourceY < this.getHeight();
         sourceY+=(1/factor), targetY++)
    {
      sourcePixel = this.getPixel((int) sourceX,(int) sourceY);
      targetPixel = canvas.getPixel((int) targetX, (int) targetY);
      targetPixel.setColor(sourcePixel.getColor());
    }
  }
  return canvas;
}
```

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 2.6). 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();
```

Program 4: Method to compose this picture into a target

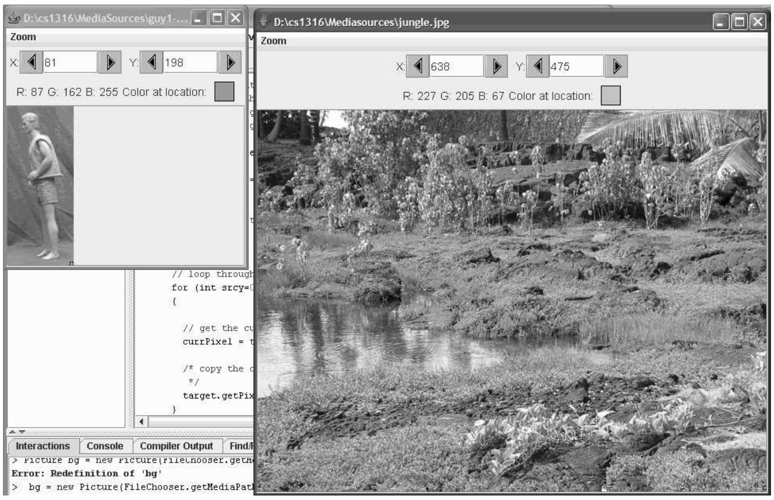

FIGURE 2.6: Using the `explore` method to see the sizes of the guy and the jungle

```
/**
 * 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());
          }
        }
      }
    }
```

We can then compose the guy into the jungle like this (Figure 2.7).

```
> 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 2.7: Composing the guy into the jungle

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

```
> 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 2.8: Chromakeying the monster into the jungle using different levels of bluescreening

Program 5: Methods for general chromakey and bluescreen

```
/**
 * Method to do chromakey using an input color for background
 * at a given point.
 * @param target the picture onto which we chromakey this picture
 * @param bgcolor the color to make transparent
 * @param threshold within this distance from bgcolor, make transparent
 * @param targetx target X position to start at
 * @param targety target Y position to start at
 */
public void chromakey(Picture target, Color bgcolor, int threshold,
                      int targetx, int targety)
{
  Pixel currPixel = null;
  Pixel newPixel = null;
```

```
    // loop through the columns
    for (int srcx=0, trgx=targetx; srcx<getWidth() && trgx<target.getWidth(); srcx++, trgx+
    {

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

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

        /* if the color at the current pixel is within threshold of
         * the input color, then don't copy the pixel
         */
        if (currPixel.colorDistance(bgcolor)>threshold)
        {
          target.getPixel(trgx,trgy).setColor(currPixel.getColor());
        }
      }
    }
  }

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

    // loop through the columns
    for (int srcx=0, trgx=targetx;
         srcx<getWidth() && trgx<target.getWidth();
         srcx++, trgx++)
    {

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

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

        /* if the color at the current pixel mostly blue (blue value is
```

```
           * greater than red and green combined), then don't copy pixel
           */
          if (currPixel.getRed() + currPixel.getGreen() > currPixel.getBlue())
          {
            target.getPixel(trgx,trgy).setColor(currPixel.getColor());
          }
        }
      }
    }
```

2.3 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 2.9). 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.

```
> 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")
```

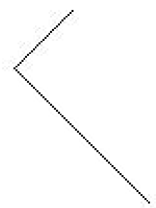

FIGURE 2.9: A drawing with a turtle

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 2.10).

```
> 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 2.10: Dropping the monster character

We'll rotate the turtle and drop again (Figure 2.11).

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

We can drop using loops and patterns, too (Figure 2.12). Why don't we see 12 monsters here? I'm not sure – there may be limits to how much we can rotate.

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

2.4 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.

FIGURE 2.11: Dropping the monster character after a rotation

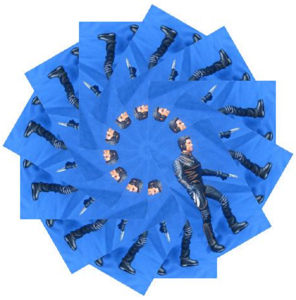

FIGURE 2.12: An iterated turtle drop of a monster

```
> Sound s = new Sound(FileChooser.getMediaPath("gonga-2.wav"));
> Sound s2 = new Sound(FileChooser.getMediaPath("gongb-2.wav"));
> s.play();
> s2.play();
> s.reverse().play(); // Play first sound in reverse
> s.append(s2).play(); // Play first then second sound
> 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();
```

Program 6: Sound methods

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

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

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

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

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

  return target;
```

```
    }

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

          int sampleValue, mixValue,newValue;

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

     /**
     * 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*.

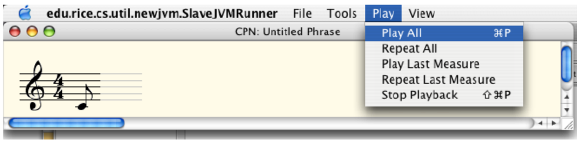

FIGURE 2.13: Playing all the notes in a score

- 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`.

2.5 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 2.13):

```
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 2.14 shows the relation between frequencies, keys, and MIDI notes[1]. A simpler summary is in Table 2.1.

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

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

[1] Taken from `http://www.phys.unsw.edu.au/~jw/notes.html`

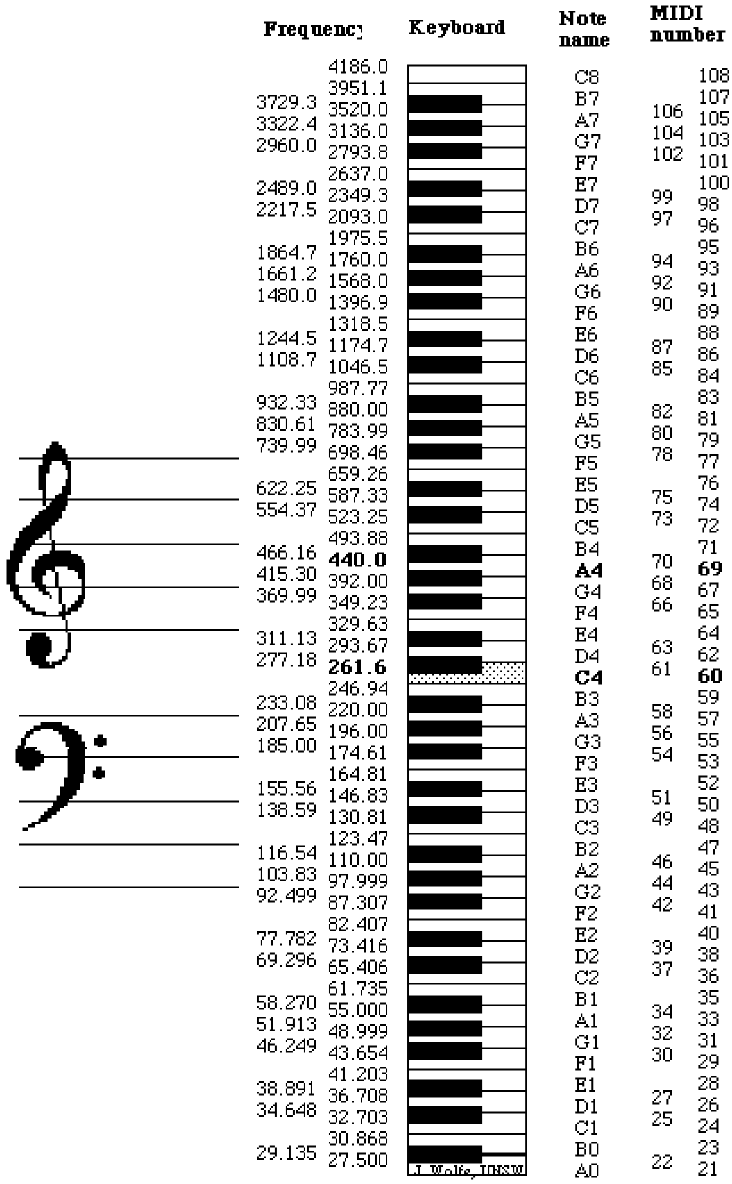

FIGURE 2.14: Frequencies, keys, and MIDI notes

Octave #	Note Numbers											
	C	C#	D	D#	E	F	F#	G	G#	A	A#	B
-1	0	1	2	3	4	5	6	7	8	9	10	11
0	12	13	14	15	16	17	18	19	20	21	22	23
1	24	25	26	27	28	29	30	31	32	33	34	35
2	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59
4	60	61	62	63	64	65	66	67	68	69	70	71
5	72	73	74	75	76	77	78	79	80	81	82	83
6	84	85	86	87	88	89	90	91	92	93	94	95
7	96	97	98	99	100	101	102	103	104	105	106	107
8	108	109	110	111	112	113	114	115	116	117	118	119
9	120	121	122	123	124	125	126	127				

TABLE 2.1: MIDI notes

```
> 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 2.2.
- 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 2.15). 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.

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

TABLE 2.2: MIDI Program numbers

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

> phr2
-------- jMusic PHRASE: 'Untitled Phrase' contains 2 notes.  Start
time: 0.5 --------
jMusic NOTE: [Pitch = 64][RhythmValue = 0.5][Dynamic = 85][Pan = 0.5][Duration
= 0.45]
jMusic NOTE: [Pitch = 67][RhythmValue = 2.0][Dynamic = 85][Pan = 0.5][Duration
= 1.8]

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

FIGURE 2.15: 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 2.16). JMusic is documented in this way. You can get to the JMusic Javadoc at `http://jmusic.ci.qut.edu.au/jmDocumentation/index.html`, or you can download it onto your own computer `http://jmusic.ci.qut.edu.au/GetjMusic.html`.

Table 2.3 lists the constant names in `JMC` for accessing instrument names.

AAH	**BREATHNOISE**	**EL_BASS**
ABASS	**BRIGHT_ACOUSTIC**	**EL_GUITAR**
AC_GUITAR	**BRIGHTNESS**	**ELECTRIC_BASS**
ACCORDION	**CALLOPE**	**ELECTRIC_GRAND**
ACOUSTIC_BASS	**CELESTA**	**ELECTRIC_GUITAR**
ACOUSTIC_GRAND	**CELESTE**	**ELECTRIC_ORGAN**
ACOUSTIC_GUITAR	**CELLO**	**ELECTRIC_PIANO**
AGOGO	**CGUITAR**	**ELPIANO**
AHHS	**CHARANG**	**ENGLISH_HORN**
ALTO	**CHIFFER**	**EPIANO**
ALTO_SAX	**CHIFFER_LEAD**	**EPIANO2**
ALTO_SAXOPHONE	**CHOIR**	**FANTASIA**
APPLAUSE	**CHURCH_ORGAN**	**FBASS**
ATMOSPHERE	**CLAR**	**FIDDLE**
BAG_PIPES	**CLARINET**	**FINGERED_BASS**
BAGPIPE	**CLAV**	**FLUTE**
BAGPIPES	**CLAVINET**	**FRENCH_HORN**
BANDNEON	**CLEAN_GUITAR**	**FRET**
BANJO	**CONCERTINA**	**FRET_NOISE**
BARI	**CONTRA_BASS**	**FRETLESS**
BARI_SAX	**CONTRABASS**	**FRETLESS_BASS**
BARITONE	**CRYSTAL**	**FRETNOISE**
BARITONE_SAX	**CYMBAL**	**FRETS**
BARITONE_SAXOPHONE	**DGUITAR**	**GLOCK**
BASS	**DIST_GUITAR**	**GLOCKENSPIEL**
BASSOON	**DISTORTED_GUITAR**	**GMSAW_WAVE**
BELL	**DOUBLE_BASS**	**GMSQUARE_WAVE**
BELLS	**DROPS**	**GOBLIN**
BIRD	**DRUM**	**GT_HARMONICS**
BOTTLE	**DX_EPIANO**	**GUITAR**
BOTTLE_BLOW	**EBASS**	**GUITAR_HARMONICS**
BOWED_GLASS	**ECHO**	**HALO**
BRASS	**ECHO_DROP**	**HALO_PAD**
BREATH	**ECHO_DROPS**	**HAMMOND_ORGAN**

TABLE 2.3: JMusic constants in `JMC` for MIDI program changes, Part 1

HARMONICA	PANFLUTE	SLAP
HARMONICS	PBASS	SLAP_BASS
HARP	PHONE	SLOW_STRINGS
HARPSICHORD	PIANO	SOLO_VOX
HELICOPTER	PIANO_ACCORDION	SOP
HONKYTONK	PIC	SOPRANO
HONKYTONK_PIANO	PICC	SOPRANO_SAX
HORN	PICCOLO	SOPRANO_SAXOPHONE
ICE_RAIN	PICKED_BASS	SOUNDEFFECTS
ICERAIN	PIPE_ORGAN	SOUNDFX
JAZZ_GUITAR	PIPES	SOUNDTRACK
JAZZ_ORGAN	PITZ	SPACE_VOICE
JGUITAR	PIZZ	SQUARE
KALIMBA	PIZZICATO_STRINGS	STAR_THEME
KOTO	POLY_SYNTH	STEEL_DRUM
MARIMBA	POLYSYNTH	STEEL_DRUMS
METAL_PAD	PSTRINGS	STEEL_GUITAR
MGUITAR	RAIN	STEELDRUM
MUSIC_BOX	RECORDER	STEELDRUMS
MUTED_GUITAR	REED_ORGAN	STR
MUTED_TRUMPET	REVERSE_CYMBAL	STREAM
NGUITAR	RHODES	STRINGS
NYLON_GUITAR	SAW	SWEEP
OBOE	SAWTOOTH	SWEEP_PAD
OCARINA	SAX	SYN_CALLIOPE
OGUITAR	SAXOPHONE	SYN_STRINGS
OOH	SBASS	SYNTH_BASS
OOHS	SEA	SYNTH_BRASS
ORCHESTRA_HIT	SEASHORE	SYNTH_CALLIOPE
ORGAN	SFX	SYNTH_DRUM
ORGAN2	SGUITAR	SYNTH_DRUMS
ORGAN3	SHAKUHACHI	SYNTH_STRINGS
OVERDRIVE_GUITAR	SHAMISEN	SYNVOX
PAD	SHANNAI	TAIKO
PAN_FLUTE	SITAR	TELEPHONE

TABLE 2.4: JMusic constants in `JMC` for MIDI program changes, Part 2

TENOR		
TENOR_SAX		
TENOR_SAXOPHONE		
THUMB_PIANO		
THUNDER		
TIMP		
TIMPANI		
TINKLE_BELL		
TOM		
TOM_TOM		
TOM_TOMS		
TOMS		
TREMOLO		
TREMOLO_STRINGS		
TROMBONE		
TRUMPET		
TUBA		
TUBULAR_BELL		
TUBULAR_BELLS		
VIBES		
VIBRAPHONE		
VIOLA		
VIOLIN		
VIOLIN_CELLO		
VOICE		
VOX		
WARM_PAD		
WHISTLE		
WIND		
WOODBLOCK		
WOODBLOCKS		
XYLOPHONE		

TABLE 2.5: JMusic constants in `JMC` for MIDI program changes, Part 3

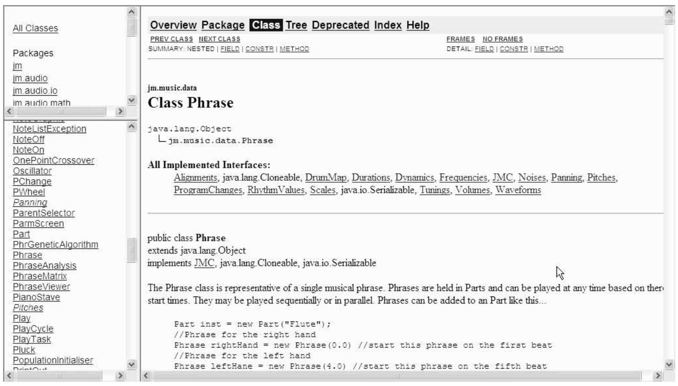

FIGURE 2.16: JMusic documention for the class `Phrase`

PART TWO

STRUCTURING MEDIA

Chapter 3 Structuring Music
Chapter 4 Structuring Images
Chapter 5 Structuring Sounds
Chapter 6 Generalizing Lists and Trees
Chapter 7 User Interface Structures
Chapter 8 Objects in Graphics: Animation

C H A P T E R 3

Structuring Music

3.1 STARTING OUT WITH JMUSIC
3.2 MAKING A SIMPLE SONG OBJECT
3.3 SIMPLE STRUCTURING OF NOTES WITH AN ARRAY
3.4 MAKING THE SONG SOMETHING TO EXPLORE
3.5 MAKING ANY SONG SOMETHING TO EXPLORE
3.6 STRUCTURING MUSIC

3.1 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(phrase);
Error: Undefined class 'phrase'
> View.notate(phr);
```

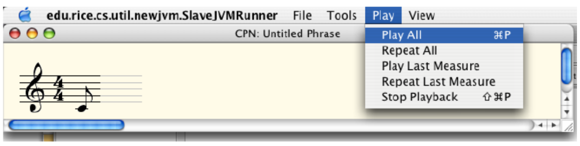

FIGURE 3.1: Playing all the notes in a score

3.2 MAKING A SIMPLE SONG OBJECT

Program 7: *Amazing Grace* as a Song Object

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

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

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

    double[] phrase1data =
    {JMC.G4, JMC.QN,
      JMC.C5, JMC.HN, JMC.E5,JMC.EN, JMC.C5,JMC.EN,
      JMC.E5,JMC.HN,JMC.D5,JMC.QN,
      JMC.C5,JMC.HN,JMC.A4,JMC.QN,
      JMC.G4,JMC.HN,JMC.G4,JMC.EN,JMC.A4,JMC.EN,
      JMC.C5,JMC.HN,JMC.E5,JMC.EN,JMC.C5,JMC.EN,
      JMC.E5,JMC.HN,JMC.D5,JMC.EN,JMC.E5,JMC.EN,
      JMC.G5,JMC.DHN};
    double[] phrase2data =
    {JMC.G5,JMC.HN,JMC.E5,JMC.EN,JMC.G5,JMC.EN,
      JMC.G5,JMC.HN,JMC.E5,JMC.EN,JMC.C5,JMC.EN,
      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, `double`s.

  Using the program (Figure 3.2):

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

3.3 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 3.3).

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

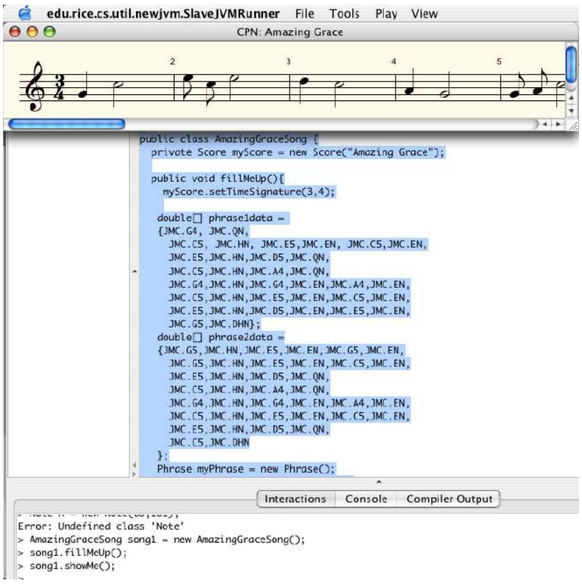

FIGURE 3.2: Trying the Amazing Grace song object

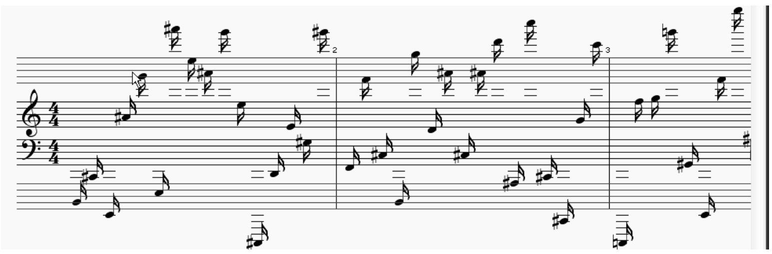

FIGURE 3.3: A hundred random notes

3.4 MAKING THE SONG SOMETHING TO EXPLORE

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

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

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

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

Program 8: Amazing Grace with Multiple Voices

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

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

  public Score getScore() {
    return myScore;
  };

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

     double[] phrase1data =
     {JMC.G4, JMC.QN,
       JMC.C5, JMC.HN, JMC.E5,JMC.EN, JMC.C5,JMC.EN,
       JMC.E5,JMC.HN,JMC.D5,JMC.QN,
       JMC.C5,JMC.HN,JMC.A4,JMC.QN,
       JMC.G4,JMC.HN,JMC.G4,JMC.EN,JMC.A4,JMC.EN,
       JMC.C5,JMC.HN,JMC.E5,JMC.EN,JMC.C5,JMC.EN,
       JMC.E5,JMC.HN,JMC.D5,JMC.EN,JMC.E5,JMC.EN,
       JMC.G5,JMC.DHN};
     double[] phrase2data =
     {JMC.G5,JMC.HN,JMC.E5,JMC.EN,JMC.G5,JMC.EN,
       JMC.G5,JMC.HN,JMC.E5,JMC.EN,JMC.C5,JMC.EN,
       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 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
  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 3.4:

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

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

FIGURE 3.4: Multi-voice *Amazing Grace* notation

(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 9: Amazing Grace as Song Elements

```
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)
  //     to get the end time.
  public double getEndTime(){
    return this.myPart.getPhrase(0).getEndTime();
  }

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

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

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

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

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

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

```
    }

}
```

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

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

That's an awful lot of extra effort just to do this, but here's the cool part. Let's do several other variations on *Amazing Grace* without writing any more programs. Say that you have a fondness for banjo, fiddle, and pipes for *Amazing Grace* (Figure 3.5).

```
> 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();
```

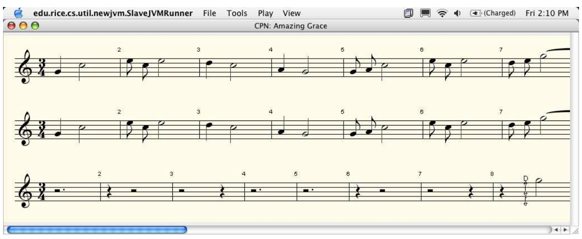

FIGURE 3.5: AmazingGraceSongElements with 3 pieces

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

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

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

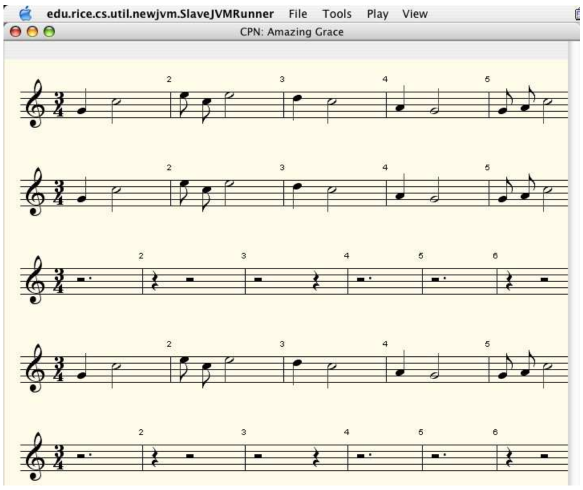

FIGURE 3.6: AmazingGraceSongElements with 3 pieces

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

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

3.5 MAKING ANY SONG SOMETHING TO EXPLORE

What makes `AmazingGraceSongElement` something specific to the song *Amazing-Grace*? 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.

Program 10: 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) starTime
    // We can set it here with whatever setPhrase gets as input
    myPhrase.setStartTime(startTime);
    this.myPart.addPhrase(myPhrase);
    // In MVAmazingGraceSong, we did this when we initialized
    // the part. But we CAN do it later
    this.myPart.setInstrument(instrument);
  }

  // First phrase of Amazing Grace
  public Phrase phrase1() {
      double[] phrase1data =
   {JMC.G4, JMC.QN,
     JMC.C5, JMC.HN, JMC.E5,JMC.EN, JMC.C5,JMC.EN,
     JMC.E5,JMC.HN,JMC.D5,JMC.QN,
     JMC.C5,JMC.HN,JMC.A4,JMC.QN,
     JMC.G4,JMC.HN,JMC.G4,JMC.EN,JMC.A4,JMC.EN,
     JMC.C5,JMC.HN,JMC.E5,JMC.EN,JMC.C5,JMC.EN,
     JMC.E5,JMC.HN,JMC.D5,JMC.EN,JMC.E5,JMC.EN,
     JMC.G5,JMC.DHN};

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

```

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

      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);
  }

```

```
  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.

```
> 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 to make every other song elements' phrase? We can get around this by creating a `static` method.

Static methods are known to the `class`, not to the individual objects (*instances*). We'd write it something like this:

```
// First phrase of Amazing Grace
static public Phrase phrase1() {
    double[] phrase1data =
 {JMC.G4, JMC.QN,
   JMC.C5, JMC.HN, JMC.E5,JMC.EN, JMC.C5,JMC.EN,
   JMC.E5,JMC.HN,JMC.D5,JMC.QN,
   JMC.C5,JMC.HN,JMC.A4,JMC.QN,
   JMC.G4,JMC.HN,JMC.G4,JMC.EN,JMC.A4,JMC.EN,
   JMC.C5,JMC.HN,JMC.E5,JMC.EN,JMC.C5,JMC.EN,
   JMC.E5,JMC.HN,JMC.D5,JMC.EN,JMC.E5,JMC.EN,
   JMC.G5,JMC.DHN};

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

We'd actually use this method like this:

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

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

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

Program 11: General Song Elements and Song Phrases

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

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

  // When we make a new element, the next part is empty, and ours is a blank new part
  public SongElement(){
```

```
    this.next = null;
    this.myPart = new Part();
  }

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


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

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

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

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

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

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

```
    myScore.setTimeSignature(3,4);
    myScore.setTempo(120.0);

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

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

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

    // 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.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;
 }

}
```

We can use this like this:

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

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

3.5.1 Adding More Phrases

Program 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.DS4,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.C4,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;
 }
```

```java

  //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.SN,
     JMC.G4,JMC.SN,JMC.E4,JMC.SN,JMC.G4,JMC.SN,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 3.7).

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

3.5.2 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.

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

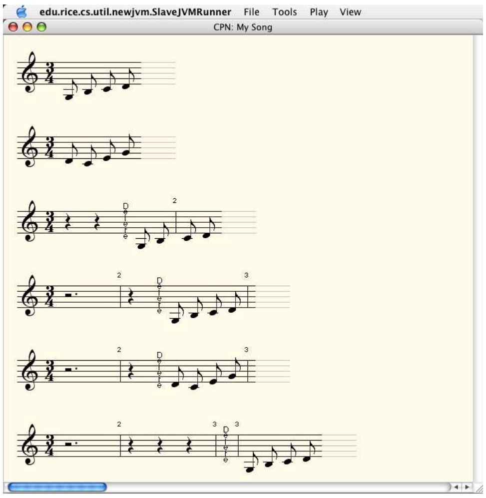

FIGURE 3.7: Playing some different riffs in patterns

Program 13: Computed Phrases

```
//Larger Riff1
static public Phrase pattern1() {
      double[] riff1data =
{JMC.G3,JMC.EN,JMC.B3,JMC.EN,JMC.C4,JMC.EN,JMC.D4,JMC.EN};
 double[] riff2data =
{JMC.D4,JMC.EN,JMC.C4,JMC.EN,JMC.E4,JMC.EN,JMC.G4,JMC.EN};

 int counter1;
 int counter2;

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

   //Larger Riff2
static public Phrase pattern2() {
      double[] riff1data =
```

```
{JMC.G3,JMC.EN,JMC.B3,JMC.EN,JMC.C4,JMC.EN,JMC.D4,JMC.EN};
 double[] riff2data =
{JMC.D4,JMC.EN,JMC.C4,JMC.EN,JMC.E4,JMC.EN,JMC.G4,JMC.EN};

 int counter1;
 int counter2;

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

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

 int counter1;
 int counter2;

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

```
   > import jm.JMC;
> SongElement sax1 = new SongElement();
> sax1.setPhrase(SongPhrase.pattern1(),0.0,JMC.TENOR_SAX);
> sax1.showFromMeOn();
-- Constructing MIDI file from'My Song'... Playing with JavaSound
... Completed MIDI playback --------
> SongElement sax2 = new SongElement();
> sax2.setPhrase(SongPhrase.pattern2(),0.0,JMC.TENOR_SAX);
> sax2.showFromMeOn()
-- Constructing MIDI file from'My Song'... Playing with JavaSound
... Completed MIDI playback --------
```

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

Here's what the sax plus rhythm looked like (Figure 3.8).

FIGURE 3.8: 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 3.9).

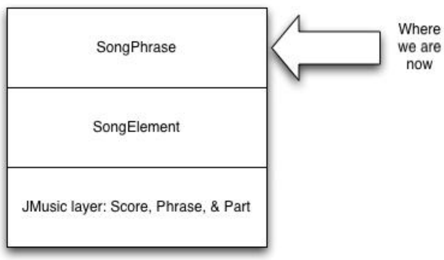

FIGURE 3.9: We now have layers of software, where we deal with only one at a time

3.6 STRUCTURING MUSIC

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

- It's hard to use. We have to specify each phrase's start time and instrument. That's a lot of specification, and it doesn't correspond to how musicians tend to think about music structure. More typically, musicians see a single music *part* as having a single instrument and start time (much as the structure of the class `Part` in the underlying JMusic classes).

- While we have a linked list for connecting the elements of our songs, we don't *use* the linked list for anything. Because each element has its own start time, there is no particular value to having an element before or after any other song element.

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

Computer Science Idea: Refactoring refines a design.
We refactor designs in order to improve them. Our early decisions about where to what aspect of a piece of software might prove to be inflexible or downright *wrong* (in the sense of not describing what we want to describe) as we continue to work. Refactoring is a process of simplifying and improving a design.

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

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

Program 14: SongNode class

```
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.
   * @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();
     };

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

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

  }

}
```

We can use this new class to do some of the things that we did before (Figure 3.10).

```
> 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);
```

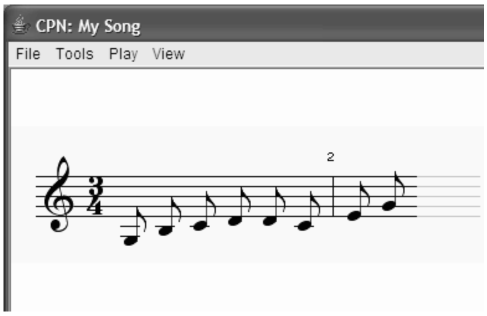

FIGURE 3.10: First score generated from ordered linked list

Remember the documentation for the JMusic classes that we saw earlier in the book? That documentation can actually be automatically generated from the comments that we provide. *Javadoc* is the name for the specialized commenting structure and the tool that generates HTML documentation from that structure. The commenting structure is: (XXX-TO-DO See DrJava docs for now.) (Figure 3.11

3.6.1 Now Let's Play!

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

Program 15: Repeating and weaving methods

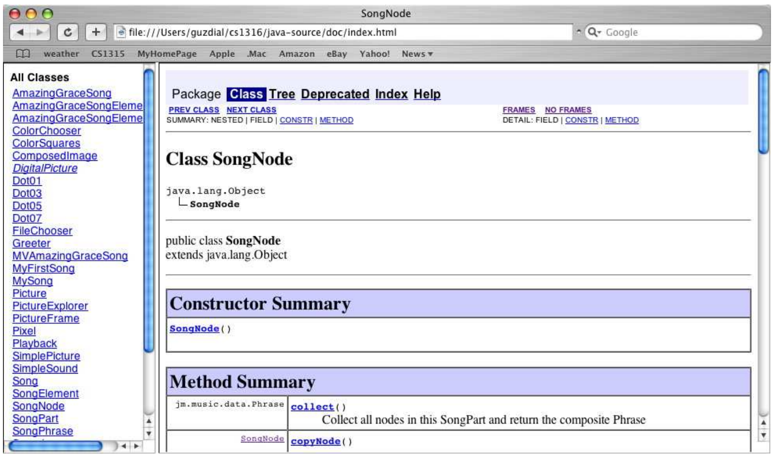

FIGURE 3.11: Javadoc for the class SongNode

```
/*
 * 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++;
      };

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

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

First, let's make 15 copies of one pattern (Figure 3.12).

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

```
> first.showFromMeOn(JMC.FLUTE);
```

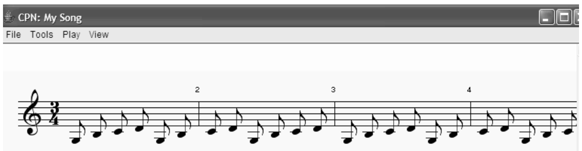

FIGURE 3.12: Repeating a node several times

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

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

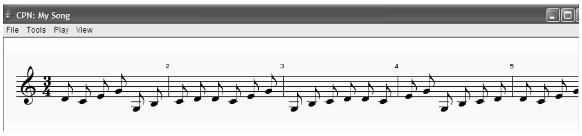

FIGURE 3.13: Weaving a new node among the old

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);
```

3.6.2 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.

Program 16: SongPart class

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

```java
public class SongPart {

  /*
   * SongPart has a Part
   */
  public Part myPart;
  /*
   * SongPart has a SongNode that is the beginng 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
  }

   /**
   * 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 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);
  }

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

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

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

     // At the end, let's see it!
```

```
        View.notate(myScore);

    }


}
```

While our new structure is very flexible, it's not the easiest thing to use. We don't want to have to type everything into the Interactions Pane every time. So, we'll create a class that has its `main` method that will run on its own. You can execute it using Run Document's Main Method (F2) in the Tools menu. Using `MySong`, we can get back to having multi-part music in a single score (Figure 3.14).

Program 18: MySong class with a main metho0d

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

public class MyFirstSong {
  public static void main(String [] args) {
    Song songroot = new Song();

    SongNode node1 = new SongNode();
    SongNode riff3 = new SongNode();
    riff3.setPhrase(SongPhrase.riff3());
    node1.repeatNext(riff3,16);
    SongNode riff1 = new SongNode();
    riff1.setPhrase(SongPhrase.riff1());
    node1.weave(riff1,7,1);
    SongPart part1 = new SongPart(JMC.PIANO, node1);

    songroot.setFirst(part1);

    SongNode node2 = new SongNode();
    SongNode riff4 = new SongNode();
    riff4.setPhrase(SongPhrase.riff4());
    node2.repeatNext(riff4,20);
    node2.weave(riff1,4,5);
    SongPart part2 = new SongPart(JMC.STEEL_DRUMS, node2);

    songroot.setSecond(part2);
    songroot.show();
  }
}
```

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.

FIGURE 3.14: Multi-part song using our classes

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.

PROBLEMS

3.1. 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.

3.2. 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.

3.3. 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.

3.4. The current implementation of `Song` implements *two* channels. Channel *nine* is the *MIDI Drum Kit* where the notes are different percussion instruments (Figure 3.1). Modify the `Song` class take a third channel, which gets assigned to MIDI channel 9 and plays a percussion `SongPart`.

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

TABLE 3.1: MIDI Drum Kit Notes

CHAPTER 4

Structuring Images

4.1 SIMPLE ARRAYS OF PICTURES
4.2 LISTING THE PICTURES, LEFT-TO-RIGHT
4.3 LISTING THE PICTURES, LAYERING
4.4 REPRESENTING SCENES WITH TREES

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

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

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

4.1 SIMPLE ARRAYS OF PICTURES

The simplest thing to do is to simply list all the pictures we want in array. We then compose them each into a background (Figure 4.1).

```
> Picture [] myarray = new Picture[5];
> myarray[0]=new Picture(FileChooser.getMediaPath("katie.jpg"));
> myarray[1]=new Picture(FileChooser.getMediaPath("barbara.jpg"));
> myarray[2]=new Picture(FileChooser.getMediaPath("flower1.jpg"));
> myarray[3]=new Picture(FileChooser.getMediaPath("flower2.jpg"));
> myarray[4]=new Picture(FileChooser.getMediaPath("butterfly.jpg"));
> Picture background = new Picture(400,400)
> for (int i = 0; i < 5; i++)
   {myarray[i].scale(0.5).compose(background,i*10,i*10);}
> background.show();
```

4.2 LISTING THE PICTURES, LEFT-TO-RIGHT

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

FIGURE 4.1: 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 4.2).

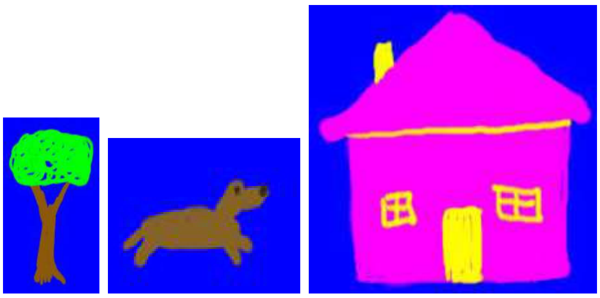

FIGURE 4.2: Elements to be used in our scenes

Program 19: Elements of a scene in position order

```
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
   * @param bg Picture to draw drawing on
   **/
  public void drawFromMeOn(Picture bg) {
    PositionedSceneElement current;
    int currentX=0, currentY = bg.getHeight()-1;
```

```
    current = this;
    while (current != null)
    {
      current.drawMeOn(bg,currentX, currentY);
      currentX = currentX + current.getPicture().getWidth();
      current = current.getNext();
    }
  }

  /**
   * Method to draw from this picture, using bluescreen.
   * @param bg Picture to draw drawing on
   * @param left x position to draw from
   * @param bottom y position to draw from
   **/

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

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 4.3).

```
> FileChooser.setMediaPath("D:/cs1316/MediaSources/");
> PositionedSceneElement tree1 = new PositionedSceneElement(new Picture(FileChooser.getMediaP
> PositionedSceneElement tree2 = new PositionedSceneElement(new Picture(FileChooser.getMediaP
> PositionedSceneElement tree3 = new PositionedSceneElement(new Picture(FileChooser.getMediaP
> PositionedSceneElement doggy = new PositionedSceneElement(new Picture(FileChooser.getMediaP
> PositionedSceneElement house = new PositionedSceneElement(new Picture(FileChooser.getMediaP
> Picture bg = new Picture(FileChooser.getMediaPath("jungle.jpg"));
> tree1.setNext(tree2); tree2.setNext(tree3); tree3.setNext(doggy);
doggy.setNext(house);
> tree1.drawFromMeOn(bg);
> bg.show();
> bg.write("D:/cs1316/first-house-scene.jpg");
```

This successfully draws a scene, but is it easy to recompose into new scenes? Let's say that we decide that we actually want the dog between trees two and three, instead of tree three and the house. To change the list, we need `tree2` to point at the `doggy` element, `doggy` to point at `tree3`, and `tree3` to point at the `house` (what the `doggy` used to point at). Then redraw the scene on a new background (Figure 4.4).

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

FIGURE 4.3: Our first scene

```
> tree1.drawFromMeOn(bg);
> bg.show();
> bg.write("D:/cs1316/second-house-scene.jpg");
```

FIGURE 4.4: Our second scene

4.2.1 Generalizing moving the element

Let's consider what happened in this line:

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

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

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

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

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

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

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

The result is that `doggy` is removed entirely (Figure 4.5).

FIGURE 4.5: Removing the doggy from the scene

Now we can re-insert the `doggy` wherever we want, say, after `tree1` (Figure 4.6):

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

4.3 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 21: LayeredSceneElements

```
public class LayeredSceneElement {

  /**
   * the picture that this element holds
```

FIGURE 4.6: Inserting the doggy into the scene

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

  /**
   * Methods to set and get next elements
   * @param nextOne next element in list
   **/
```

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

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

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

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

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

  /**
   * Method to draw from this picture, using bluescreen.
   * @param bg Picture to draw drawing on
   **/

  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.

```
> Picture bg = new Picture(400,400);
> LayeredSceneElement tree1 = new LayeredSceneElement(
new Picture(FileChooser.getMediaPath("tree-blue.jpg")),10,10);
> LayeredSceneElement tree2 = new LayeredSceneElement(
new Picture(FileChooser.getMediaPath("tree-blue.jpg")),100,10);
> LayeredSceneElement tree3 = new LayeredSceneElement(
new Picture(FileChooser.getMediaPath("tree-blue.jpg")),200,100);
> LayeredSceneElement house = new LayeredSceneElement(
new Picture(FileChooser.getMediaPath("house-blue.jpg")),175,175);
> LayeredSceneElement doggy = new LayeredSceneElement(
new Picture(FileChooser.getMediaPath("dog-blue.jpg")),150,325);
> tree1.setNext(tree2); tree2.setNext(tree3); tree3.setNext(doggy);
doggy.setNext(house);
> tree1.drawFromMeOn(bg);
> bg.show();
> bg.write("D:/cs1316/first-layered-scene.jpg");
```

The result (Figure 4.7) shows the house in front of a tree and the dog. In the upper left, we can see one tree overlapping the other.

FIGURE 4.7: First rendering of the layered sene

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

```
> 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");
```

Have you ever used a drawing program like *Visio* or even *PowerPoint* where you brough 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

FIGURE 4.8: Second rendering of the layered sene

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?

4.3.1 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.

4.4 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.

PROBLEMS

4.1. Set up a scene with `PositionedSceneElement`, then change the layering of just a single element using `remove` and `insertAfter`.

CHAPTER 5

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.

CHAPTER 6

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.

CHAPTER 7

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.

CHAPTER 8

Objects in Graphics: Animation

8.1 BASIC FRAMESEQUENCE

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.

8.1 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 8.1).

```
> 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);
```

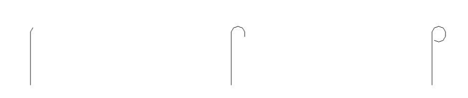

FIGURE 8.1: Three frames from the simple `FrameSequence` example

P A R T T H R E E

SIMULATIONS

Chapter 9 Continuous Simulation
Chapter 10 Discrete Event Simulation

CHAPTER 9

Continuous Simulation

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.

CHAPTER 10

Discrete Event Simulation

10.1 DISTRIBUTIONS AND EVENTS

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.

10.1 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.

APPENDIX A

Utility Classes

Utility 1: Turtle

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

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


public class Turtle  {

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

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

    // turtles are created on pictures
    public Turtle(Picture newPicture) {
      myPicture = newPicture;
      myGraphics = (Graphics2D) myPicture.getBufferedImage().createGraphics();
      myGraphics.setColor(foreground);
      height = myPicture.getHeight();
      width = myPicture.getWidth();
    };
```

```
// accessor methods
public double x()           { return x;           }
public double y()           { return y;           }

public double heading() { return heading; }
public void setHeading(double newhead) {
  heading = newhead;
}

public void setColor(Color color) {
    foreground = color;
    myGraphics.setColor(foreground);
}

//Pen Stuff
public void penUp(){
  penDown = false;
}

public void penDown(){
  penDown = true;
}

public boolean pen(){
  return penDown;
}

public float getPenWidth(){
  BasicStroke bs = (BasicStroke) myGraphics.getStroke();
  return bs.getLineWidth();
}

public void setPenWidth(float width){
  BasicStroke newStroke = new BasicStroke(width);
  myGraphics.setStroke(newStroke);
};

public void go(double x, double y) {
    if (penDown)
      myGraphics.draw(new Line2D.Double(this.x, this.y, x, y));
    this.x = x;
    this.y = y;
}

// draw w-by-h rectangle, centered at current location
public void spot(double w, double h) {
    myGraphics.fill(new Rectangle2D.Double(x - w/2, y - h/2, w, h));
```

```
}

// draw circle of diameter d, centered at current location
public void spot(double d) {
   if (d <= 1) myGraphics.drawRect((int) x, (int) y, 1, 1);
   else myGraphics.fill(new Ellipse2D.Double(x - d/2, y - d/2, d, d));
}


// draw spot using jpeg/gif - fix to be at (x, y)
public void spot(String s) {
    Picture spotPicture = new Picture(s);
    Image image = spotPicture.getImage();

    int w = image.getWidth(null);
    int h = image.getHeight(null);

    myGraphics.rotate(Math.toRadians(heading), x, y);
    myGraphics.drawImage(image, (int) x, (int) y, null);
    myGraphics.rotate(Math.toRadians(heading), x, y);
}

// draw spot using gif, left corner on (x, y), scaled of size w-by-h
public void spot(String s, double w, double h) {
    Picture spotPicture = new Picture(s);
    Image image = spotPicture.getImage();

    myGraphics.rotate(Math.toRadians(heading), x, y);
    myGraphics.drawImage(image, (int) x, (int) y,
                            (int) w, (int) h, null);
    myGraphics.rotate(Math.toRadians(heading), x, y);
}

public void pixel(int x, int y) {
    myGraphics.drawRect(x, y, 1, 1);
}

// rotate counterclockwise in degrees
public void turn(double angle) { heading = (heading + angle) % 360; }

// walk forward
public void forward(double d) {
    double oldx = x;
    double oldy = y;
    x += d * -Math.cos(Math.toRadians(heading));
    y += d * Math.sin(Math.toRadians(heading));
    if (penDown)
```

```
        myGraphics.draw(new Line2D.Double(x, y, oldx, oldy));
    }


    // write the given string in the current font
    public void write(String s) {
        FontMetrics metrics = myGraphics.getFontMetrics();
        int w = metrics.stringWidth(s);
        int h = metrics.getHeight();
        myGraphics.drawString(s, (float) (x - w/2.0), (float) (y + h/2.0));
    }

    // write the given string in the given font
    public void write(String s, Font f) {
        myGraphics.setFont(f);
        write(s);
    }


}
```

Bibliography

Index

*/, 13
++, 26
–, 26
/*, 13

abstraction, 4
AmazingGraceSong, 38
AmazingGraceSongElement, 42
and, 9

behavior, 2
binary trees, 91
block, 9
bluescreen, 18
break, 9
buttons, 87

Call and response, 40
cast, 26
chromakey, 18
class, 10
class method, 11
common bug
    Don't try to change the input variables, 18
    Java may be hidden on Macintosh, 11
    Width is the size, not the coordinate, 15
compiled, 8
computer science idea
    Better or worse structures depend on use, 2
    Layering software makes it easier to change, 46
    Layering software makes it easier to change, Part 2, 59
    Refactoring refines a design., 60
connections, 4
constructor, 25, 26
curly braces, 9

declaring, 8
discrete event simulations, 91
dot notation, 11
double, 8
DrJava, 4
drop, 22

edges, 4
Element, 84
expert musicians, 42
explore, 16

file paths, 10
FileChooser, 10
for, 9
FrameSequence, 88
functions, 7

getMediaPath, 16
graph, 4
graphical user interface (GUI), 87
graphs, 88
GUI, 87

hierarchy, 3

import, 26
increaseRed, 12
instances, 10, 50
instrument, 26, 28
Interface, 84

jar file, 4
Java, 2
Javadoc, 30, 63
JMC, 28
JMC.C4, 28
JMC.FLUTE, 28
JMC.QN, 28
JMusic, 4
    Javadoc, 30

LayeredSceneElement, 81
layering, 78
layout managers, 87
linked list, 3, 72, 73
    images, 73
    music, 42
    traversal, 82
list, 60, 73
lists, 87
literal, 8

machine language, 8
main, 69

making it work
    The command history isn't reset, 13
Math.random(), 38
method, 12
methods, 7
MIDI
    drum kit, 70
MIDI channels, 28
MIDI Drum Kit, 70
MIDI music, 22
MIDI note, 26
MIDI program, 26
MMList, 86
model, 42
    ordering, 42
MTree, 86
MySong, 69

new, 8
new Picture(), 11
node, 60
nodes, 4
Note, 26
null, 11

or, 9
ordered list, 91
ordering, 42

panes, 87
Part, 42, 60, 70
Phrase, 26, 28, 42, 70
pickAFile(), 10
Picture, 7, 18
pile, 83
Pixel, 11
Pixels [], 12
PositionedSceneElement, 75, 81
PowerPoint, 82
predator and prey, 90
print, 10
println, 41
private, 13
program
    Amazing Grace as a Song Object, 37
    Amazing Grace as Song Elements, 42
    Amazing Grace as Song Elements, Take 2, 47

Amazing Grace with Multiple Voices, 40
Computed Phrases, 57
Elements of a scene in position order, 73
General Song Elements and Song Phrases, 50
LayeredSceneElements, 78
Method for Picture to scale by a factor, 16
Method to compose this picture into a target, 16
Method to flip an image, 14
Method to increase red in Picture, 12
Methods for general chromakey and bluescreen, 19
Methods to remove and insert elements in a list, 77
More phrases to play with, 53
MySong class with a main metho0d, 69
Repeating and weaving methods, 63
Song class–root of a tree-like music structure, 68
SongNode class, 60
SongPart class, 66
Sound methods, 24
property, 83
public, 13

queue, 90

random, 91
refactoring, 60
render, 88
rendering, 82
representations, 2
resources, 90
Rotation, 84
rotations, 84

Sample, 7
sampled sounds, 22
scene graph, 84
scenes, 72
Score, 28, 42, 70
self, 12
sequence, 42
setMediaPath, 16
show(), 11
Song, 70
SongElement, 50, 60
SongNode, 60
SongPart, 66
SongPhrase, 50, 66
sort, 91

Sound, 7, 25
square brackets, 10
Stack, 84
stack, 83
static, 10, 49
stream, 41
string concatenation, 42
structure, 2
System.out.println, 10
System.out.println(), 41

this, 12
time order, 91
Translation, 84
translations, 84
traverses, 82
tree, 3, 60, 66
Turtle, 84, 90
types, 7

utility
    Turtle, 92

Visio, 82
void, 13

WAV files, 22
while, 9