BETA Version, October 21st, 2004:
Introduction to Computing and Programming in Java:
A Multimedia Approach

Mark Guzdial and Barbara Ericson
College of Computing/GVU
Georgia Institute of Technology

PRENTICE HALL, Upper Saddle River, New Jersey 07458
Copyright held by Mark Guzdial and Barbara Ericson, 2004.
Dedicated to our children Matthew, Katherine, and Jennifer
Contents

1 Introduction 7

1 Introduction to Computer Science and Media Computation 8
  1.1 What is Computer Science About?  8
  1.2 What Computers Understand  13
  1.3 Media Computation: Why digitize media?  15
  1.4 Computer Science for Everyone  16
    1.4.1 It’s about communication  17
    1.4.2 It’s about process  17

2 Introduction to Programming 19
  2.1 Programming is about Naming  20
    2.1.1 Files and their Names  21
  2.2 Programming in Java  22
    2.2.1 History of Java  22
    2.2.2 Introduction to Objects and Classes  23
    2.2.3 Introduction to DrJava  24
    2.2.4 Starting DrJava  24
    2.2.5 Using DrJava  26
  2.3 Using Variables  30
  2.4 Class and Object Methods  31
    2.4.1 Invoking Class Methods  32
    2.4.2 Invoking Object Methods  33
  2.5 Invoking Media Methods  36
    2.5.1 Showing a Picture  38
    2.5.2 Variable Substitution  39
    2.5.3 Object references  41
    2.5.4 Playing a Sound  42
    2.5.5 Naming your Media (and other Values)  43
    2.5.6 Types Matter  45
    2.5.7 Naming the Result of a Method  46
  2.6 Making a Program  48
    2.6.1 Variable Programs: Real functions that Take Input  55
II Pictures

3 Modifying Pictures using Loops
  3.1 How Pictures are Encoded .......................... 62
  3.1.1 Color Representations .......................... 66
  3.2 Manipulating Pictures ............................. 70
    3.2.1 Exploring pictures ............................ 75
  3.3 Changing color values ............................. 76
    3.3.1 Using while loops in pictures .................. 77
    3.3.2 Increasing/decreasing red (green, blue) ........ 81
    3.3.3 Creating a sunset ............................ 94
    3.3.4 Making sense of methods ....................... 96
    3.3.5 Variable name scope .......................... 99
    3.3.6 Using a for loop ............................. 103
    3.3.7 Lightening and darkening ....................... 105
    3.3.8 Creating a negative ........................... 106
    3.3.9 Converting to grayscale ....................... 108

4 Modifying Pixels in a Matrix 115
  4.1 Copying Pixels ..................................... 115
    4.1.1 Looping across the pixels with a nested loop .... 116
    4.1.2 Mirroring a picture ........................... 118
  4.2 Copying and transforming pictures .................. 126
    4.2.1 Copying ...................................... 127
    4.2.2 Creating a Collage ............................ 133
    4.2.3 Blending Pictures ............................. 139
    4.2.4 Rotation ..................................... 142
    4.2.5 Scaling ...................................... 146
  4.3 Color Figures ...................................... 153

5 Conditionally Modifying Pixels 162
  5.1 Conditional Pixel Changes ......................... 163
    5.1.1 Comparing Colors .............................. 164
    5.1.2 Replacing Colors ............................... 164
    5.1.3 Turning class methods into object methods ........ 167
    5.1.4 Reducing red eye .............................. 169
  5.2 Simple Edge Detection: conditionals with two options ... 171
  5.3 Sepia toned and posterized pictures: Using multiple conditionals to choose the color .......... 175
  5.4 Highlighting lightest and darkest areas .............. 182
  5.5 Combining pixels: Blurring ......................... 183
  5.6 Background subtraction ............................ 187
  5.7 Chromakey ......................................... 191
III Sounds 199

6 Modifying all Samples in a Sound 200
  6.1 How Sound is Encoded 200
    6.1.1 The Physics of Sound 201
    6.1.2 Exploring how sounds look 204
    6.1.3 Encoding the Sound 206
  6.2 Manipulating sounds 210
    6.2.1 Open sounds and manipulating samples 211
    6.2.2 Using MediaTools for looking at captured sounds 214
    6.2.3 Introducing the loop 215
  6.3 Changing the volume of sounds 220
    6.3.1 Increasing volume 220
    6.3.2 Did that really work? 222
    6.3.3 Decreasing volume 226
    6.3.4 Using a for loop 228
    6.3.5 Making sense of methods, in sounds 229
  6.4 Normalizing sounds 230
    6.4.1 Generating clipping 233

7 Modifying Samples using Ranges 237
  7.1 Manipulating Different Sections of the Sound Differently 237
  7.2 Create a Sound Clip 239
  7.3 Splicing sounds 242
  7.4 Reversing a sound 247
  7.5 Mirroring a sound 248
List of Figures

1.1 A cooking recipe–you can always double the ingredients, but throwing in an extra cup of flour won’t cut it, and don’t try to brown the chicken after adding the tomato sauce! .................................. 11
1.2 Comparing programming languages: A common simple programming task is to print the words “Hello, World!” to the screen. ...... 12
1.3 Eight wires with a pattern of voltages is a byte, which gets interpreted as a pattern of eight 0’s and 1’s, which gets interpreted as a decimal number. ........................................... 14
2.1 DrJava Preferences Window .................................................. 25
2.2 DrJava Splash Screen .......................................................... 25
2.3 DrJava (with annotations) ..................................................... 27
2.4 Declaring primitive variables and memory assignment ............. 32
2.5 Showing the parent and child relationship between mammal and dog (left) and Object and String (right) ................................ 34
2.6 Declaring object variables and memory assignment .................. 35
2.7 The File Chooser ............................................................... 36
2.8 File chooser with media types identified ............................... 38
2.9 Picking, making, and showing a picture, using the result of each method in the next method. The picture used is matt-spaceman.jpg. 39
2.10 Picking, making, and showing a picture, when naming the pieces. The picture shown is katie.jpg. ............................... 42
2.11 Shows creation and reuse of an object variable. ......................... 44
2.12 Defining and executing pickAndShow() ................................. 51
3.1 A depiction of the first five elements in an array ....................... 63
3.2 An example matrix (two-dimensional array) of numbers ............ 64
3.3 Upper left corner of DrJava window with part magnified 600% .......................... 65
3.4 Image shown in the picture explorer: 100% image on left and 500% on right (close-up of the branch over the mountain) ................ 65
3.5 Merging red, green, and blue to make new colors ..................... 66
3.6 Picking colors using the HSB color model .............................. 67
3.7 The ends of this figure are the same colors of gray, but the middle two quarters contrast sharply so the left looks darker than the right 68
3.8 The Macintosh OS X RGB color picker .................................. 68
3.9 Picking a color using RGB sliders from Java ........................... 69
3.10 RGB triplets in a matrix representation ................................. 69
3.11 Directly modifying the pixel colors via commands: Note the small black line on the left under the line across the leaf ....................... 75
3.12 Exploring the caterpillar with the line ................................. 75
3.13 Using the MediaTools image exploration tools ......................... 76
3.14 Flowchart of a while loop .................................................. 79
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.15</td>
<td>The original picture (left) and red-decreased version (right)</td>
</tr>
<tr>
<td>3.16</td>
<td>Using the picture explorer to convince ourselves that the red was decreased</td>
</tr>
<tr>
<td>3.17</td>
<td>Overly blue (left) and red increased by 30% (right)</td>
</tr>
<tr>
<td>3.18</td>
<td>Original (left) and blue erased (right)</td>
</tr>
<tr>
<td>3.19</td>
<td>Original beach scene (left) and at (fake) sunset (right)</td>
</tr>
<tr>
<td>3.20</td>
<td>Flowchart of a for loop</td>
</tr>
<tr>
<td>3.21</td>
<td>Original picture, lightened picture, and darkened picture</td>
</tr>
<tr>
<td>3.22</td>
<td>Negative of the image</td>
</tr>
<tr>
<td>3.23</td>
<td>Color picture converted to grayscale</td>
</tr>
<tr>
<td>4.1</td>
<td>Once we pick a mirror point, we can just walk x/2 halfway and subtract/add to the mirror point</td>
</tr>
<tr>
<td>4.2</td>
<td>Original picture (left) and mirrored along the vertical axis (right)</td>
</tr>
<tr>
<td>4.3</td>
<td>A motorcycle mirrored horizontally, top to bottom (left) and bottom to top (right)</td>
</tr>
<tr>
<td>4.4</td>
<td>Temple of Hephaistos from the ancient agora in Athens, Greece</td>
</tr>
<tr>
<td>4.5</td>
<td>Coordinates where we need to do the mirroring</td>
</tr>
<tr>
<td>4.6</td>
<td>The manipulated temple</td>
</tr>
<tr>
<td>4.7</td>
<td>Copying a picture to a canvas</td>
</tr>
<tr>
<td>4.8</td>
<td>Copying a picture midway into a canvas</td>
</tr>
<tr>
<td>4.9</td>
<td>Copying part of a picture onto a canvas</td>
</tr>
<tr>
<td>4.10</td>
<td>Flowers in the <em>mediasources</em> folder</td>
</tr>
<tr>
<td>4.11</td>
<td>Collage of flowers</td>
</tr>
<tr>
<td>4.12</td>
<td>Blending the picture of Katie and Jenny</td>
</tr>
<tr>
<td>4.13</td>
<td>Rotating some numbers in a table to the left 90 degrees</td>
</tr>
<tr>
<td>4.14</td>
<td>Copying a picture to a canvas rotated to the left 90 degrees</td>
</tr>
<tr>
<td>4.15</td>
<td>Scaling the picture down</td>
</tr>
<tr>
<td>4.16</td>
<td>Scaling up a picture</td>
</tr>
<tr>
<td>4.17</td>
<td>Merging red, green, and blue to make new colors</td>
</tr>
<tr>
<td>4.18</td>
<td>Color: RGB triplets in a matrix representation</td>
</tr>
<tr>
<td>4.19</td>
<td>Color: The original picture (left) and red-decreased version (right)</td>
</tr>
<tr>
<td>4.20</td>
<td>Color: Overly blue (left) and red increased by 30% (right)</td>
</tr>
<tr>
<td>4.21</td>
<td>Color: Original (left) and blue erased (right)</td>
</tr>
<tr>
<td>4.22</td>
<td>Original beach scene (left) and at (fake) sunset (right)</td>
</tr>
<tr>
<td>4.23</td>
<td>Color: Lightening and darkening the original picture</td>
</tr>
<tr>
<td>4.24</td>
<td>Color: Negative of the image</td>
</tr>
<tr>
<td>4.25</td>
<td>Color: Color picture converted to grayscale</td>
</tr>
<tr>
<td>4.26</td>
<td>Color: Increasing reds in the browns</td>
</tr>
<tr>
<td>4.27</td>
<td>Color: Increasing reds in the browns, within a certain range</td>
</tr>
<tr>
<td>4.28</td>
<td>Finding the range where Jenny’s eyes are red, then changing them to black</td>
</tr>
<tr>
<td>4.29</td>
<td>Frames from the slow sunset movie</td>
</tr>
<tr>
<td>4.30</td>
<td>Frames from the slow fade-out movie</td>
</tr>
<tr>
<td>4.31</td>
<td>Frames from the Mommy watching Katie movie</td>
</tr>
<tr>
<td>4.32</td>
<td>Frames from the original too dark movie</td>
</tr>
<tr>
<td>4.33</td>
<td>Frames from the modified lighter movie</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

4.34 Frames from the original movie with kids crawling in front of a blue screen .......................................................... 160
4.35 Frames from the kids on the moon movie ................................................. 161
5.1 Flowchart of an if statement ............................................................... 163
5.2 Increasing reds in the browns .............................................................. 165
5.3 On left the couch color changes, on right the couch color doesn’t change 167
5.4 Finding the range of where Jenny’s eyes are red ...................................... 170
5.5 After fixing red-eye. .............................................................................. 171
5.6 Flowchart of an if with an else .............................................................. 173
5.7 Original picture and after edge detection ................................................. 174
5.8 Flowchart of an if, else if, and an else .................................................... 176
5.9 Original scene (left) and using our sepia-tone program ........................... 177
5.10 Reducing the colors (right) from the original (left) .................................... 179
5.11 Pictures posterized to two levels (left) and four levels (right) ............... 182
5.12 Original picture (left) and light or dark areas highlighted (right) ......... 184
5.13 Making the flower bigger, then blurring to reduce pixellation .............. 186
5.14 A picture of a child (Katie), and her background without her ................ 187
5.15 A new background, the moon .............................................................. 187
5.16 Katie on the moon .............................................................................. 189
5.17 Two people in front of a wall, and a picture of the wall ......................... 191
5.18 Swapping a beach for the wall, using background subtraction, with a threshold of 50 ........................................................... 191
5.19 Mark in front of a blue sheet .................................................................. 192
5.20 Mark on the moon ................................................................................ 193
5.21 Mark on the beach ................................................................................ 193
5.22 Student in front of a red background, and with flash on ....................... 195
5.23 Using chromakey program with a red background ................................... 195
6.1 Raindrops causing ripples in the surface of the water, just as sound causes ripples in the air ............................................... 201
6.2 One cycle of the simplest sound, a sine wave ........................................ 202
6.3 The note A above middle C is 440 Hz .................................................... 203
6.4 Some synthesizers using triangular (or sawtooth) or square waves ....... 204
6.5 Sound editor main tool ......................................................................... 204
6.6 Viewing the sound signal as it comes in ................................................ 205
6.7 Viewing the sound in a spectrum view .................................................. 206
6.8 Viewing a sound in spectrum view with multiple “spikes” ..................... 207
6.9 Viewing the sound signal in a sonogram view ....................................... 208
6.10 Area under a curve estimated with rectangles ...................................... 208
6.11 A depiction of the first five elements in a real sound array .................. 210
6.12 A sound recording graphed in the MediaTools ..................................... 210
6.13 The sound editor open menu in MediaTools application ....................... 215
6.14 MediaTools application open file dialog .............................................. 215
6.15 A sound opened in the editor in MediaTools application ....................... 216
6.16 Exploring the sound in the editor in MediaTools application ................. 216
6.17 Comparing the graphs of the original sound (left) and the louder one (right) ................................................................. 222
LIST OF FIGURES

6.18 Comparing specific samples in the original sound (left) and the louder one (right) .................................................. 223
6.19 Comparing the original sound with the normalized one ............ 232
6.20 Comparing the original sound with one with all values set to extremes. 234
7.1 Exploring the "This is a test" to find the end of the first word ....... 240
7.2 Exploring the sound clip .............................................. 241
7.3 Comparing the original sound (left) to the spliced sound (right) .... 246
7.4 Comparing the original sound (left) to the reversed sound (right) ... 249
7.5 Comparing the mirror point in the original sound (left) to the mirrored sound (right) .............................................. 250
Preface

One of the clearest lessons from the research on computing education is that one doesn’t just “learn to program.” One learns to program something [4, 14]. How motivating that something is can make the difference between learning to program or not [5].

In this book students will learn about programming by writing programs to manipulate media. Students will create and modify images, such as the image collage on the front cover of this book. Students will modify sounds, like splicing words into sentences or reversing sounds to make interesting effects. Students will write programs to generate web pages from data in databases, in the same way that CNN.com and Amazon.com do. They will create animations and movies using special effects like the ones students see on television and in movies.

Students in courses taught at Georgia Tech have found these programs interesting and motivating. Students have even reported that they sometimes turn in their programs and then keep working on them to see what else they can do.

People want to communicate. We are social creatures, and the desire to communicate is one of our primal motivations. Increasingly, the computer is used as a tool for communication, even more than as a tool for calculation. Virtually all published text, images, sounds, music, and movies today are prepared using computing technology.

This book is about teaching people to program in order to communicate. The book focuses on how to manipulate images, sounds, text, and movies as professionals might, but with programs written by the students. We realize that most people will use professional-grade applications to perform these same manipulations. But knowing how to do it with your own programs means that you can do it if you need to. You may want to say something with your media, but you may not know how to make PhotoShop or Final Cut Pro do what you want. Knowing how to program means that you have power of expression that is not limited by your application software.

This book is not just about programming in media. Media manipulation programs can be hard to write, or behave in unexpected ways. Questions arise like “Why is this same image filter faster in Photoshop?” and “That was hard to debug–are there ways of writing programs that are easier to debug?” Answering questions like these is what computer scientists do. There are several chapters at the end of the book that are about computing, not just programming.

The computer is the most amazingly creative device that humans have ever conceived of. It is literally completely made up of mind-stuff. The notion “Don't just dream it, be it” is really possible on a computer. If you can imagine it, you can make it “real” on the computer. Playing with programming can be and should be enormous fun.

TO TEACHERS

The media computation approach used in this book starts with what students use computers for: image manipulation, digital music, web pages, games, and so on.
We then explain programming and computing in terms of these activities. We want students to visit Amazon (for example) and think, “Here’s a catalog website— and I know that this is implemented with a database and a set of programs that format the database entries as Web pages.” Starting from a relevant context makes transfer of knowledge and skills more likely, and it also helps with retention.

The media computation approach spends about 2/3 of the time on giving students experiences with a variety of media in contexts that they find motivating. After that 2/3, though, they start to develop questions. “Why is it that Photoshop is faster than my program?” and “Movie code is slow—how slow do programs get?” are typical. At that point, we introduce the abstractions and the valuable insights from Computer Science that answer their questions. That’s what the last part of this book is about.

Researchers in computing education have been exploring why withdrawal or failure rates in college-level introductory computing courses have been so high. The rate of students withdrawing from college-level introductory computing courses or receiving a D or F grade (commonly called the WDF rate) has been reported in the 30–50% range, or even higher. One of the common themes from research into why the WDF rate is so high is that computing courses seem “irrelevant” and unnecessarily focusing on “tedious details” such as efficiency [19][1].

However, students have found media computation to be relevant as evidenced by survey responses and the reduction in our WDF rate from an average of 28% to 11.5% for the pilot offering of this course. Spring 2004 was the first semester taught by instructors other than Mark Guzdial, and the WDF rate dropped to 9.5% for the 395 students who enrolled. Charles Fowler at Gainesville College in Georgia has been having similar results in his courses there.

The approach in this book is different than in many introductory programming books. We teach the same computing concepts but not necessarily in the usual order. For example, while we create and use objects early we don’t have students defining new classes till fairly late. Studies have shown that learning to program is hard and that students often have trouble with the basics (variables, iteration, and conditionals). We focus on the basics for six chapters with three on images and three on sounds. We introduce new concepts only after setting the stage for why we would need them. For example, we don’t introduce iteration until after we change pixel colors one-by-one. We don’t introduce procedural decomposition until after our methods get too long to easily be debugged.

Memory is associative—we remember things based on what else we relate to those things. People can learn concepts and skills on the promise that it will be useful some day, but those concepts and skills will be related only to those promises, not to everyday life. The result has been described as “brittle knowledge” [6]—the kind of knowledge that gets you through the exam, but promptly gets forgotten because it doesn’t relate to anything but being in that class. If we want students to gain transferable knowledge (knowledge that can be applied in new situations), we have to help them to relate the knowledge to more general problems, so that the memories get indexed in ways that associate with those kinds of problems [17]. Thus, we teach with concrete experiences that students can explore and relate to (e.g., iteration for removing red-eye in pictures).

We do know that starting from the abstractions doesn’t really work for stu-
Students. Ann Fleury has shown that novice students just don’t buy what we tell them about encapsulation and reuse (e.g., [9]). Students prefer simpler code that they can trace easily, and actually think that code that an expert would hate is better. It takes time and experience for students to realize that there is value in well-designed systems, and without experience, it’s very difficult for students to learn the abstractions.

Another unusual thing about this book is that we start using arrays in chapter 3, in our first significant programs. Typically, introductory computing courses push arrays off until later, since they’re obviously more complicated than variables with simple values. But a relevant context is very powerful [14]. The matrices of pixels in images occur in the students’ everyday life—a magnifying glass on a computer monitor or television makes that clear.

Our goal is to teach programming in a way that students find relevant, motivating, and social. To be relevant we have the students write programs to do things that students currently use computers for: i.e. image, sound, and text manipulation. For motivation we assign open-ended creative assignments such as: create an image collage with the same image at least 4 times using 3 different image manipulations and a mirroring. As for the social aspect we encourage collaboration on assignments and posting of student work. Students learn from each other and try to outdo each other.

Our publisher, Alan Apt of Prentice-Hall, recognizes that this book represents a new and radical approach to teaching introductory computing. The publisher is willing to provide textbooks at no cost for a trial offering of a course (or a section of a large course) to encourage you to try this approach in your own school.

**Ways to Use This Book**

This book contains the content that we teach at Georgia Tech. Individual teachers may skip some sections (e.g., the section on additive synthesis, MIDI, and MP3), but all of the content here has been tested with our students.

However, we can imagine using this material in many other ways:

- A short introduction to computing could be taught with just chapters 2 (introduction to programming) and 3 (modifying pictures using loops), perhaps with some material from chapters 4 and 5. We have taught even single day workshops on media computation using just this material.

- Chapters 6 through 8 basically replicate the computer science basics from chapters 3 through 5, but in the context of sounds rather than images. We find the replication useful—some students seem to relate better to the concepts of iteration and conditionals better when working with one medium than the other. Further, it gives us the opportunity to point out that the same algorithm can have similar effects in different media (e.g., scaling a picture up or down and shifting a sound higher or lower in pitch is the same algorithm). But it could certainly be skipped to save time.

- Chapter 12 (on movies) introduces no new programming or computing concepts. While motivating, movie processing could be skipped for time.
LIST OF FIGURES

- We do recommend getting to at least some of the chapters in the last unit, in order to lead students into thinking about the computing and programming in a more abstract manner, but clearly not all of the chapters have to be covered.

TYPOGRAPHICAL NOTATIONS

Examples of Java code look like this: \( x = x + 1; \). Longer examples look like this:

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

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

```
> 3 + 4
7
```

User interface components of DrJava will be specified using a smallcaps font, like FILE menu item and the COMPILe ALL button. DrJava is the development environment we use in the book and we do recommend that you use it. It is a free development environment that has an interactions pane which allows users to type Java statements and get immediate results. Of course, you can use other development environments with this book.

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

Program 1: An Example Program

Programs (recipes) appear like this:

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

The programming language used in this book is Java. Java is a high-level object-oriented programming language that runs on most computers and many small electronic devices. It is widely used in industry and in universities.

The development environment used in this book is DrJava. It is a free and easy to use development environment. DrJava lets the student focus on learning to program in Java and not on how to use the development environment. An advantage of DrJava is that you can try out Java code in the interactions pane without having to write a "main" method.

You don’t have to use this development environment. There are many development environments that are available for use with Java. If you use another development environment just add the directory that has the Java classes developed for this book to the classpath. See the documentation for your development environment for how to do this.

ACKNOWLEDGEMENTS

Our sincere thanks go out to the following:

- Adam Wilson built the MediaTools that are so useful for exploring sounds and images and processing video.
6 LIST OF FIGURES

- Matthew, Katherine, and Jennifer Guzdial all contributed pictures for use in this book.

- Thanks for permission to use their snapshots from class in examples are former *Media Computation* students Constantino Kombosch, Joseph Clark, and Shannon Joiner.

The cover image was created by Rachel Cobb who was a first year Architecture student at Georgia Institute of Technology in Spring 2003 when she created the image for a homework assignment to build a collage. The original arch image came from the National Park Service gallery at [http://www.nps.gov/arch/gallery/index.htm](http://www.nps.gov/arch/gallery/index.htm). The original image and the collage are both used with permission, and our thanks!
PART ONE

INTRODUCTION

Chapter 1  Introduction to Computer Science and Media Computation

Chapter 2  Introduction to Programming
CHAPTER 1

Introduction to Computer Science and Media Computation

1.1 WHAT IS COMPUTER SCIENCE ABOUT?

Computer science is the study of process: How we do things, how we specify what we do, how we specify what the stuff is that you’re processing. But that’s a pretty dry definition. Let’s try a metaphorical one.

Computer Science Idea: Computer science is the study of recipes (programs)

They’re a special kind of recipe—one that can be executed by a computational device, but that point is only of importance to computer scientists. The important point overall is that a computer science program defines exactly what’s to be done.

If you’re a biologist who wants to describe how migration works or how DNA replicates, or if you’re a chemist who wants to explain how an equilibrium is reached in a reaction, or if you’re a factory manager who wants to define a machine-and-belt layout and even test how it works before physically moving heavy things into
Section 1.1 What is Computer Science About?

position, then being able to write a program that specifies exactly what happens, in terms that can be completely defined and understood, is very useful. This exactness is part of why computers have radically changed so much of how science is done and understood.

It may sound funny to call programs or algorithms a recipe, but the analogy goes a long way. Much of what computer scientists study can be defined in terms of recipes:

- Some computer scientists study how recipes are written: Are there better or worse ways of doing something? If you’ve ever had to separate whites from yolks in eggs, you know that knowing the right way to do it makes a world of difference. Computer science theoreticians worry about the fastest and shortest recipes, and the ones that take up the least amount of space (you can think about it as counter space — the analogy works). How a recipe works, completely apart from how it’s written, is called the study of algorithms. Software engineers worry about how large groups can put together recipes that still work. (The recipe for some programs, like the one that keeps track of Visa/MasterCard records has literally millions of steps!)

- Other computer scientists study the units used in recipes. Does it matter whether a recipe uses metric or English measurements? The recipe may work in either case, but if you have the read the recipe and you don’t know what a pound or a cup is, the recipe is a lot less understandable to you. There are also units that make sense for some tasks and not others, but if you can fit the units to the tasks well, you can explain yourself more easily and get things done faster—and avoid errors. Ever wonder why ships at sea measure their speed in knots? Why not use things like meters per second? There are places, like at sea, where more common terms aren’t appropriate or don’t work as well. The study of computer science units is referred to as data structures. Computer scientists who study ways of keeping track of lots of data in lots of different kinds of units are studying databases.

- Can recipes be written for anything? Are there some recipes that can’t be written? Computer scientists actually do know that there are recipes that can’t be written. For example, you can’t write a recipe that can absolutely tell, for any other recipe, if the other recipe will actually work. How about intelligence? Can we write a recipe that, when a computer followed it, the computer would actually be thinking (and how would you tell if you got it right)? Computer scientists in theory, intelligent systems, artificial intelligence, and systems worry about things like this.

- There are even computer scientists who worry about whether people like what the recipes produce, like the restaurant critics for the newspaper. Some of these are human-computer interface specialists who worry about whether people like how the recipes work (those “recipes” that produce an interface that people use, like windows, buttons, scrollbars, and other elements of what we think about as a running program).
Chapter 1  Introduction to Computer Science and Media Computation

- Just as some chefs specialize in certain kinds of recipes, like crepes or barbecue, computer scientists also specialize in special kinds of recipes. Computer scientists who work in graphics are mostly concerned with recipes that produce pictures, animations, and even movies. Computer scientists who work in computer music are mostly concerned with recipes that produce sounds (often melodic ones, but not always).

- Still other computer scientists study the emergent properties of recipes. Think about the World Wide Web. It’s really a collection of millions of recipes (programs) talking to one another. Why would one section of the Web get slower at some point? It’s a phenomena that emerges from these millions of programs, certainly not something that was planned. That’s something that networking computer scientists study. What’s really amazing is that these emergent properties (that things just start to happen when you have many, many recipes interacting at once) can also be used to explain non-computational things. For example, how ants forage for food or how termites make mounds can also be described as something that just happens when you have lots of little programs doing something simple and interacting.

The recipe metaphor also works on another level. Everyone knows that some things in a recipe can be changed without changing the result dramatically. You can always increase all the units by a multiplier (say, double) to make more. You can always add more garlic or oregano to the spaghetti sauce. But there are some things that you cannot change in a recipe. If the recipe calls for baking powder, you may not substitute baking soda. If you’re supposed to boil the dumplings then saute’ them, the reverse order will probably not work well (Figure 1.1).

Similarly, for software recipes (programs), there are usually things you can easily change: The actual names of things (though you should change names consistently), some of the constants (numbers that appear as plain old numbers, not as variables), and maybe even some of the data ranges (sections of the data) being manipulated. But the order of the commands to the computer, however, almost always has to stay exactly as stated. As we go on, you’ll learn what can be changed safely, and what can’t.

Computer scientists specify their programs with programming languages (Figure 1.2). Different programming languages are used for different purposes. Some of them are wildly popular, like Java and Visual Basic. Others are more obscure, like Squeak and T. Others are designed to make computer science ideas very easy to learn, like Scheme or Python, but the fact that they’re easy to learn doesn’t always make them very popular nor the best choice for experts building larger or more complicated programs. It’s a hard balance in teaching computer science to pick a language that is easy to learn and is popular and useful enough that students are motivated to learn it.

Why don’t computer scientists just use natural human languages, like English or Spanish? The problem is that natural languages evolved the way that they did to enhance communications between very smart beings, humans. As we’ll go into more in the next section, computers are exceptionally dumb. They need a level of specificity that natural language isn’t good at. Further, what we say to one another in natural communication is not exactly what you’re saying in a computational
FIGURE 1.1: A cooking recipe—you can always double the ingredients, but throwing in an extra cup of flour won’t cut it, and don’t try to brown the chicken after adding the tomato sauce!

There are so many different kinds of programming languages because there are so many different kinds of programs to write. Programs written in the programming language $C$ tend to be very fast and efficient, but they also tend to be hard to read, hard to write, and require units that are more about computers than about bird migrations or DNA or whatever else you want to write your program about. The programming language Lisp (and its related languages like Scheme, T, and Common Lisp) is very flexible and is well suited to exploring how to write programs that have never been written before, but Lisp looks so strange compared to languages like $C$ that many people avoid it and there are (natural consequence) few people who know it. If you want to hire a hundred programmers to work on your project, you’re going to find it easier to find a hundred programmers who know a popular language than a less popular one—but that doesn’t mean that the popular language is the best one for your task!


Java is known for being object-oriented, platform neutral (runs on many com-
Chapter 1  Introduction to Computer Science and Media Computation

Python/Jython

```python
def hello():
    print "Hello World"
```

Java

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

C++

```cpp
#include <iostream.h>

main() {
    cout << "Hello World!" << endl;
    return 0;
}
```

Scheme

```scheme
(define helloworld
    (lambda ()
        (display "Hello World")
        (newline)))
```

FIGURE 1.2: Comparing programming languages: A common simple programming task is to print the words “Hello, World!” to the screen.

Computers and electronic devices), robust, and secure. An early drawback to Java was that programs written in Java often had a slower execution time than ones written in C or C++. However, current Java compilers and interpreters have essentially eliminated this problem.

Let’s make clear some of our language that we’ll be using in this book. A program is a description of a process in a particular programming language that achieves some result that is useful to someone. A program could be small (like one that implements a calculator), or could be huge (like the program that your bank uses to track all of its accounts). An algorithm (in contrast) is a description of a process apart from any programming language. The same algorithm might be implemented in many different languages in many different ways in many different programs–but it would all be the same process if we’re talking about the same algorithm.
1.2 WHAT COMPUTERS UNDERSTAND

Programs are written to run on computers. What does a computer know how to do? What can we tell the computer to do in the program? The answer is “Very, very little.” Computers are exceedingly stupid. They really only know about numbers.

Actually, even to say that computers know numbers is a myth, or more appropriately, an encoding. Computers are electronic devices that react to voltages on wires. We group these wires into sets (a set of eight of these wires is called a byte and one wire is called a bit). If a wire has a voltage on it, we say that it encodes a 1. If it has no voltage on it, we say that it encodes a 0. So, from a set of eight wires (a byte), we get a pattern of eight 0’s and 1’s, e.g., 01001010. Using the binary number system, we can interpret this byte as a decimal number (Figure 1.3).

The computer has a memory filled with bytes. Everything that a computer is working with at a given instant is stored in its memory. That means that everything that a computer is working with is encoded in its bytes: JPEG pictures, Excel spreadsheets, Word documents, annoying Web pop-up ads, and the latest spam email.

A computer can do lots of things with numbers. It can add them, subtract them, multiply them, divide them, sort them, collect them, duplicate them, filter them (e.g., “make a copy of these numbers, but only the even ones”), and compare them and do things based on the comparison. For example, a computer can be told in a program “Compare these two numbers. If the first one is less than the second one, jump to step 5 in this program. Otherwise, continue on to the next step.”

It sounds like computers are incredible calculators, and that’s certainly why they were invented. The first use of computers was during World War II for calculating trajectories of projectiles (“If the wind is coming from the SE at 15 MPH, and you want to hit a target 0.5 miles away at an angle of 30 degrees East of North, then incline your launcher to . . .”). The computer is an amazing calculator. But

---

1 We’ll talk more about this level of the computer in Chapter 13.
what makes it useful for general programs is the concept of *encodings*.

Computer Science Idea: Computers can layer encodings

Computers can layer encodings to virtually any level of complexity. Numbers can be interpreted as characters, which can be interpreted in groups as Web pages. But at the bottommost level, the computer only “knows” voltages which we interpret as numbers.

If one of these bytes is interpreted as the number 65, it could just be the number 65. Or it could be the letter *A* using a standard encoding of numbers-to-letters called the *American Standard Code for Information Interchange (ASCII)*. If that 65 appears in a collection of other numbers that we’re interpreting as text, and that’s in a file that ends in “.html” it might be part of something that looks like this `<a href=`, which a Web browser will interpret as the definition of a link. Down at the level of the computer, that *A* is just a pattern of voltages. Many layers of programs up, at the level of a Web browser, it defines something that you can click on to get more information.

If the computer understands only numbers (and that’s a stretch already), how does it manipulate these encodings? Sure, it knows how to compare numbers, but how does that extend to being able to alphabetize a class list? Typically, each layer of encoding is implemented as a piece or layer of software. There’s software that understands how to manipulate characters. The character software knows how to do things like compare names because it has encoded that *a* comes before *b* and so on, and that the numeric comparison of the order of numbers in the encoding of the letters leads to alphabetical comparisons. The character software is used by other software that manipulates text in files. That’s the layer that something like Microsoft Word or Notepad or TextEdit would use. Still another piece of software
knows how to interpret HTML (the language of the Web), and another layer of
that software knows how to take HTML and display the right text, fonts, styles,
and colors.

We can similarly create layers of encodings in the computer for our specific
tasks. We can teach a computer that cells contain mitochondria and DNA, and that
DNA has four kinds of nucleotides, and that factories have these kinds of presses
and these kinds of stamps. Creating layers of encoding and interpretation so that
the computer is working with the right units (recall back to our recipe analogy)
for a given problem is the task of data representation or defining the right data
structures.

If this sounds like lots of software, it is. When software is layered like this,
it slows the computer down some. But the amazing thing about computers is that
they’re amazingly fast—and getting faster all the time!

<table>
<thead>
<tr>
<th>Computer Science Idea: Moore’s Law</th>
</tr>
</thead>
</table>
| Gordon Moore, one of the founders of Intel (maker of com-
puter processing chips for computers running Windows op-
erating systems), made the claim that the number of tran-
sistors (a key component of computers) would double at
the same price every 18 months, effectively meaning that
the same amount of money would buy twice as much com-
puting power every 18 months. This Law has continued to
hold true for decades. |

Computers today can execute literally BILLIONS of program steps per sec-
ond! They can hold in memory entire encyclopedias of data! They never get tired
nor bored. Search a million customers for a particular card holder? No problem!
Find the right set of numbers to get the best value out of an equation? Piece of
cake!

Process millions of picture elements or sound fragments or movie frames?
That’s media computation.

1.3 MEDIA COMPUTATION: WHY DIGITIZE MEDIA?

Let’s consider an encoding that would be appropriate for pictures. Imagine that
pictures were made up of little dots. That’s not hard to imagine: Look really closely
at your monitor or at a TV screen and see that your images are already made up of
little dots. Each of these dots is a distinct color. You may know from physics that
colors can be described as the sum of red, green, and blue. Add the red and green
to get yellow. Mix all three together to get white. Turn them all off, and you get
a black dot.

What if we encoded each dot in a picture as a collection of three bytes, one
each for the amount of red, green, and blue at that dot on the screen? We could
collect a bunch of these three-byte-sets to specify all the dots of a given picture.
That’s a pretty reasonable way of representing pictures, and it’s essentially how
we’re going to do it in Chapter 3.

Manipulating these dots (each referred to as a pixel or picture element) can
take a lot of processing. There can be thousands or even millions of them in a
picture. But, the computer doesn’t get bored and it’s mighty fast.

The encoding that we will be using for sound involves 44,100 two-byte-sets (called a sample) for each second of time. A three minute song requires 158,760,000 bytes. Doing any processing on this takes a lot of operations. But at a billion operations per second, you can do lots of operations to every one of those bytes in just a few moments.

Creating these kinds of encodings for media requires a change to the media. Look at the real world: It isn’t made up of lots of little dots that you can see. Listen to a sound: Do you hear thousands of little bits of sound per second? The fact that you can’t hear little bits of sound per second is what makes it possible to create these encodings. Our eyes and ears are limited: We can only perceive so much, and only things that are just so small. If you break up an image into small enough dots, your eyes can’t tell that it’s not a continuous flow of color. If you break up a sound into small enough pieces, your ears can’t tell that the sound isn’t a continuous flow of auditory energy.

The process of encoding media into little bits is called digitization, sometimes referred to as “going digital.” Digital means (according to the American Heritage Dictionary) “Of, relating to, or resembling a digit, especially a finger.” Making things digital is about turning things from continuous, uncountable, to something that we can count, as if with our fingers.

Digital media, done well, feel the same to our limited human sensory apparatus as the original. Phonograph recordings (ever seen one of those?) capture sound continuously, as an analog signal. Photographs capture light as a continuous flow. Some people say that they can hear a difference between phonograph recordings and CD recordings, but to my ear and most measurements, a CD (which is digitized sound) sounds just the same—maybe clearer. Digital cameras at high enough resolutions produce photograph-quality pictures.

Why would you want to digitize media? Because it’s easier to manipulate, to replicate exactly, to compress, and to transmit. For example, it’s hard to manipulate images that are in photographs, but it’s very easy when the same images are digitized. This book is about using the increasingly digital world of media and manipulating it—and learning computation in the process.

Moore’s Law has made media computation feasible as an introductory topic. Media computation relies on the computer doing lots and lots of operations on lots and lots of bytes. Modern computers can do this easily. Even with slow (but easy to understand) languages, even with inefficient (but easy to read and write) programs, we can learn about computation by manipulating media.

1.4 COMPUTER SCIENCE FOR EVERYONE

But why should you? Why should anyone who doesn’t want to be a computer scientist learn about computer science? Why should you be interested in learning about computation through manipulating media?

Most professionals today do manipulate media: Papers, videos, tape recordings, photographs, drawings. Increasingly, this manipulation is done with a computer. Media are very often in a digitized form today.

We use software to manipulate these media. We use Adobe Photoshop for
manipulating our images, and Macromedia SoundEdit to manipulate our sounds, and perhaps Microsoft PowerPoint for assembling our media into slideshows. We use Microsoft Word for manipulating our text, and Netscape Navigator or Microsoft Internet Explorer for browsing media on the Internet.

So why should anyone who does not want to be a computer scientist study computer science? Why should you learn to program? Isn’t it enough to learn to use all this great software? The following two sections provide two answers to these questions.

1.4.1 It’s about communication

Digital media are manipulated with software. If you can only manipulate media with software that someone else made for you, you are limiting your ability to communicate. What if you want to say something or say it in some way that Adobe, Microsoft, Apple, and the rest don’t support you in saying? If you know how to program, even if it would take you longer to do it yourself, you have that freedom.

What about learning those tools in the first place? In my years in computers, I’ve seen a variety of software come and go as the package for drawing, painting, word-processing, video editing, and beyond. You can’t learn just a single tool and expect to be able to use that your entire career. If you know how the tools work, you have a core understanding that can transfer from tool to tool. You can think about your media work in terms of the algorithms, not the tools.

Finally, if you’re going to prepare media for the Web, for marketing, for print, for broadcast, for any use whatsoever, it’s worthwhile for you to have a sense of what’s possible, what can be done with media. It’s even more important as a consumer of media that you know how the media can be manipulated, to know what’s true and what could be just a trick. If you know the basics of media computation, you have an understanding that goes beyond what any individual tool provides.

1.4.2 It’s about process

In 1961, Alan Perlis gave a talk at MIT where he made the argument that computer science, and programming explicitly, should be part of a general, liberal education [11]. Perlis is an important figure in the field of computer science. The highest award that a computer scientist can be honored with is the ACM Turing Award. Perlis was the first recipient of that award. He’s also an important figure in software engineering, and he started several of the first computer science departments in the United States.

Perlis’ argument can be made in comparison with calculus. Calculus is generally considered part of a liberal education: Not everyone takes calculus, but if you want to be well-educated, you will typically take at least a term of calculus. Calculus is the study of rates, which is important in many fields. Computer science, as we said before (page 8), is the study of process. Process is important to nearly every field, from business to science to medicine to law. Knowing process formally is important to everyone.
Chapter 1 Introduction to Computer Science and Media Computation

PROBLEMS

1.1. Every profession uses computers today. Use a Web browser and a search engine like Google to find sites that relate your field of study with computer science or computing or computation. For example, search for “biology computer science” or “management computing.”

1.2. Find an ASCII table on the Web: A table listing every character and its corresponding numeric representation. Write down the sequence of numbers whose ASCII values make up your name.

1.3. Find a Unicode table on the Web. What’s the difference between ASCII and Unicode?

1.4. Consider the representation for pictures described in Section 1.3, where each “dot” (pixel) in the picture is represented by three bytes, for the red, green, and blue components of the color at that dot. How many bytes does it take to represent a 640x480 picture, a common picture size on the Web? How many bytes does it take to represent a 1024x768 picture, a common screen size? (What do you think is meant now by a “3 megapixel” camera?)

1.5. How many different numbers can be represented by one byte? In other words, eight bits can represent from zero to what number? What if you have two bytes? Four bytes?

*1.6. How might you represent a floating point number in terms of bytes? Do a search on the Web for “floating point” and see what you find.

1.7. Look up Alan Kay and the Dynabook on the Web. Who is he, and what does he have to do with media computation?

1.8. Look up Alan Turing on the Web. Who was he, and what does he have to do with our notion of what a computer can do and how encodings work?

1.9. Look up Kurt Goedel on the Web. Who was he, and what amazing things did he do with encodings?

TO DIG DEEPER

James Gleick’s book Chaos describes more on emergent properties—how small changes can lead to dramatic effects, and the unintended impacts of designs because of difficult-to-foresee interactions.

Mitchel Resnick’s book Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds [20] describes how ants, termites, and even traffic jams and slime molds can be described pretty accurately with hundreds or thousands of very small processes (programs) running and interacting all at once.

CHAPTER 2

Introduction to Programming

2.1 PROGRAMMING IS ABOUT NAMING
2.2 PROGRAMMING IN JAVA
2.3 USING VARIABLES
2.4 CLASS AND OBJECT METHODS
2.5 INVOKING MEDIA METHODS
2.6 MAKING A PROGRAM

Chapter Learning Objectives
The media learning goals for this chapter are:

- To create Picture objects (objects of the class Picture) and show them.

- To create sound objects and play them.

The computer science goals for this chapter are:

- To use DrJava to execute Java statements.

- To use Java math operators.

- To recognize different types (encodings) of data, such as integers, floating point numbers, characters, strings, and media objects.

- To introduce casting.

- To create and use variables to store values and objects, such as numbers, strings, pictures and sounds.

- To invoke class and object methods.

- To create objects using the new keyword.

- To create methods (functions).
2.1 PROGRAMMING IS ABOUT NAMING

Computer Science Idea: Much of programming is about naming
A computer can associate names, or symbols, with just about anything: With a particular byte; with a collection of bytes making up a numeric variable or a bunch of letters; with a media element like a file, sound, or picture; or even with more abstract concepts, like a named recipe (a program or method) or a named encoding (a type or class).

A computer scientist sees a choice of names as being high quality in the same way that a philosopher or mathematician might: If the names are elegant, parsimonious, and usable.

Obviously, the computer itself doesn’t care about names. Names are for the humans. If the computer were just a calculator, then remembering names and the names’ association with values would be just a waste of the computer’s memory. But for humans, it’s very powerful. It allows us to work with the computer in a natural way.

A programming language is really a set of names that a computer has encodings for, such that those names make the computer do expected actions and interpret our data in expected ways. Some of the programming language’s names allow us to define new namings—which allows us to create our own layers of encoding.

We can associate a name with a location in memory, this is called declaring a variable. We can associate a name with a group of Java statements, we call this defining a method (function). In Java you can also assign a name to a group of related variables and methods (functions) when you define a class (type).

Computer Science Idea: Programs are for people, not computers.
Remember names are only meaningful for people, not computers. Computers just take instructions. A good program is meaningful (understandable and useful) for humans.

A program is a set of names and their values, where some of these names have values of instructions to the computer (“code”). Our instructions will be in the Java programming language. Combining these two definitions means that the Java programming language gives us a set of useful names that have a meaning to the computer, and our programs are then made up of Java’s useful names as a way of specifying what we want the computer to do.

There are good names and bad names. Bad names aren’t curse words, or TLA’s (Three Letter Acronyms), but names that aren’t understandable or easy to use. A good set of encodings and names allow one to describe methods in a way that’s natural, without having to say too much. The variety of different programming languages can be thought of as a collection of sets of namings-and-encodings. Some are better for some tasks than others. Some languages require you
to write more to describe the same program (function) than others—but sometimes that “more” leads to a much more (human) readable program that helps others to understand what you’re saying.

Philosophers and mathematicians look for very similar senses of quality. They try to describe the world in few words, using an elegant selection of words that cover many situations, while remaining understandable to their fellow philosophers and mathematicians. That’s exactly what computer scientists do as well.

How the units and values (data) of a program can be interpreted is often also named. Remember how we said in Section 1.2 (page 13) that everything is stored in groups of eight bits called bytes, and we can interpret those bytes as numbers? In some programming languages, you can say explicitly that some value is a byte, and later tell the language to treat it as a number, an integer (or sometimes int). Similarly, you can tell the computer that these series of bytes is a collection of numbers (an array of integers), or a collection of characters (a String), or even as a more complex encoding of a single floating point number (any number with a decimal point in it).

In Java, we will explicitly tell the computer how to interpret our values. Languages such as Java, C++, and C# are strongly typed. Names are strongly associated with certain types or encodings. They require you to say that this name will only be associated with integers, and that one with floating point numbers. In Java, C++, and C# you can also create your own types which is part of what makes object-oriented languages so powerful. We do this in Java by defining classes such as Picture which represents a simple digital picture. An object of the Picture class has a width and height and you can get and set the pixels of the Picture object. This isn’t a class that is part of the Java language, but a class that we have defined using Java to make it easier for students to work with digital pictures.

2.1.1 Files and their Names

A programming language isn’t the only place where computers associate names and values. Your computer’s operating system takes care of the files on your disk, and it associates names with those files. Operating systems you may be familiar with include Windows XP, Windows 2000 (Windows ME, NT, . . .), MacOS, and Linux. A file is a collection of values (bytes) on your hard disk (the part of your computer that stores things after the power gets turned off). If you know the name of a file, you can tell it to the operating system, and it can give you the values associated with that name.

You may be thinking, “I’ve been using the computer for years, and I’ve never ‘given a file name to the operating system.’ ” Maybe you didn’t realize that you were doing it, but when you pick a file from a file choosing dialog in Photoshop, or double-click a file in a directory window (or Explorer or Finder), you are asking some software somewhere to give the name you’re picking or double-clicking to the operating system, and get the values back. When you write your own programs, though, you’ll be explicitly getting file names and asking for the values stored in that file.

Files are very important for media computation. Disks can store acres and acres of information on them. Remember our discussion of Moore’s Law (Page 15)?
Disk capacity per dollar is increasing faster than computer speed per dollar! Computer disks today can store whole movies, hours (days?) of sounds, and the equivalent of hundreds of film rolls of pictures.

These media are not small. Even in a compressed form, screen size pictures can be over a million bytes large, and songs can be three million bytes or more. You need to keep them someplace where they’ll last past the computer being turned off and where there’s lots of space. This is why they are stored on your hard disk.

In contrast, your computer’s memory (RAM) is impermanent (the contents disappear when the power does) and is relatively small. Computer memory is getting larger all the time, but it’s still just a fraction of the amount of space on your disk. When you’re working with media, you will load the media from the disk into memory, but you wouldn’t want it to stay in memory after you’re done. It’s too big.

Think about your computer’s memory as your desk. You would want to keep books that you are currently working with on your desk but when you are done you will probably move those books to a book shelf. You may have many more books on your book shelf than can fit on your desk. A computer can fit much more data on the hard disk than can fit in memory. However, data must be read from disk into memory before you can work with it.

When you bring things into memory, you usually will name the value, so that you can retrieve it and use it later. In that sense, programming is something like algebra. To write generalizable equations and functions (those that work for any number or value), you wrote equations and functions with variables, like \( PV = nRT \) or \( e = Mc^2 \) or \( f(x) = \sin(x) \). Those P’s, V’s, R’s, T’s, e’s, M’s, c’s, and x’s were names for values. When you evaluated \( f(30) \), you knew that the \( x \) was the name for 30 when computing \( f \). We’ll be naming values when we program.

### 2.2 PROGRAMMING IN JAVA

The programming language that we’re going to be using in this book is called Java. It’s a language invented by James Gosling (http://java.sun.com/people/jag/) at Sun Microsystems.

#### 2.2.1 History of Java

Back in 1990 Sun created project Green to try and predict the next big thing in computers. The goal of the project was to try and develop something to position Sun ahead of its competitors. They thought that the next big thing would be networked consumer electronics devices like set-top boxes for downloading video on demand. They tried to develop a prototype using C++ but after many problems they decided to develop a new object-oriented language which they originally named Oak, after a tree outside James Gosling’s office. They created a demonstration but the cable companies weren’t really interested and the future of the project was in doubt.

At a brainstorming session they decided to try to reposition the language for use with the internet. They created a web browser that had Java programs (applets) embedded in HTML pages to do 3D rotation of a molecule and animation of a sorting algorithm. They showed this at a conference. At that time web pages
didn’t respond to user action. They simply displayed text and unchanging graphics. The audience was amazed to see the user rotate the 3d molecule on a web page. Later they renamed Oak to Java and released it for free in 1995. Since then it has become one of the fastest adopted technologies of all times. It is now used for more than just web pages. It is used in many devices from cell phones to web servers. For more on the history of Java see http://java.sun.com/features/1998/05/birthday.html.

2.2.2 Introduction to Objects and Classes

Java is an object-oriented programming language. This means that the focus for programmers is on objects (who) as well as procedures (what). Objects are persons, places, or things that do the action in a situation or are acted upon.

An example might help you to understand what focusing on the objects means. When customers enter a restaurant a greeter will welcome them to the restaurant and show them to their table. A waiter will take the order and bring the drinks and food. One or more chefs will cook the food. The waiter will create the bill and give it to the customers. The customers will pay the bill.

How many people does it take to get a customer fed in a restaurant? Well, you need at least a customer, greeter, waiter, and a chef. What other things are doing action or being acted upon? We mentioned order, table, drink, food, and bill. Each of these are objects. The objects in this situation are working together to feed the customer.

What types of objects are they? We have given names to each thing we mentioned: customer, waiter, food, etc. The names we gave are how we classify these objects. You probably know what I mean by a customer or food. But the computer doesn’t know what we mean by these things. The way that we get the computer to understand what we mean is by defining a class. A class in Java tells the computer what data we expect objects of the class to have and what things it can do. We would expect that food will have a name, a price, and a way to prepare it. We would expect that a customer would know what they can afford to pay and how to pay a bill.

Every object of the same class will have the same skills or operations (things it can do) and data or variables (things it knows about). For example, each object of the order class should know which customer placed that order and what food is in the order. An object of the chef class should know how to prepare the food.

There can be many objects of a class. A restaurant might have 3 chefs, 10 waiters, 2 greeters, and 100 food objects on its menu. On a given day and time it might have 100 customers.

Why don’t restaurants just have one type of employee? One person could greet the customers, take the orders, cook the food and deliver the food. That might be okay if there is only one customer but what about when there are many customers? You can imagine that one person wouldn’t be able to handle so many tasks and food would get burnt, orders would take too long to fill, and customers wouldn’t be happy. Restaurants break the tasks into different jobs so that they can be efficient and effective. Object-oriented programs also try to distribute the tasks to be done so that no one object does all the work. This makes it easier to
maintain and extend the program. It can also make the program more efficient.

2.2.3 Introduction to DrJava

You'll actually be programming using a tool called DrJava. DrJava is a simple editor (tool for entering program text) and interaction space so that you can try things out in DrJava and create new programs (methods) and classes. DrJava is available for free under the DrJava Open Source License, and it is under active development by the JavaPLT group at Rice University.

If you don't wish to use DrJava you can certainly use this book with another development environment. Simply set the classpath (place to look for classes that you are using in your program) to include the classes used in this book. Check your documentation for your development environment to see how to do this. We recommend using DrJava because it is free, it is easy to use, it has an interactions pane for trying out Java statements, and it includes a debugger.

To install DrJava, you'll have to do these things:

1. Make sure that you have Java 1.4 or above installed on your computer. If you don't have it, load it from the CD, or you can get it from the Sun site at http://www.java.sun.com.
2. You'll need to install DrJava. You can either load it from the CD or get it from http://drjava.org/.
3. Add the Java classes that come with the book to the extra classpaths for DrJava. Start DrJava (see the next section for how to do this), click on Edit and then Preferences. This will show the Preferences window (Figure 2.1). Click on the Add button below the Extra Classpath textarea and add the following path: c:/intro-prog-java/bookClasses.

2.2.4 Starting DrJava

How you start DrJava depends on your platform. In Windows, you’ll have a DrJava icon that you’ll simply double-click. In Linux, you’ll probably cd into your DrJava directory and type a command like java -jar drjava-DATE-TIME.jar where DATE-TIME are values for the release of DrJava that you are using. On the Macintosh, you’ll probably have to type commands in your Terminal application where you cd to the correct directory then type ./DrJava. See the instructions on the CD for what will work for your kind of computer.

Common Bug: DrJava is slow to start

DrJava will take a while to load on all platforms. Don’t worry—you’ll see the splash screen for a long time, but if you see the splash screen (Figure 2.2), it will load.
Section 2.2  Programming in Java

Common Bug: Making DrJava run faster

As we’ll talk more about later, when you’re running DrJava, you’re actually running Java. Java needs memory. If you’re finding that DrJava is running slowly, give it more memory. You can do that by quitting out of other applications that you’re running. Your email program, your instant messenger, and your digital music player all take up memory (sometimes lots of it!). Quit out of those and DrJava will run faster.

Once you start DrJava, it will look something like Figure 2.3. There are three main areas in DrJava (the bars between them move so that you can resize the areas):

- The top left window pane is the files pane. It has a list of the open files in DrJava. In Java each class that you create is usually stored in its own file. Java programs often consist of more than one class thus more than one file. You can click on a file name in the Files pane to view the contents of that file in the top right window pane (definitions pane).
Chapter 2  Introduction to Programming

• The top right part is the definitions pane. This is where you write your classes: a collection of related data and methods. This area is simply a text editor—think of it as Microsoft Word for your programs. The computer doesn’t actually try to interpret the names that you type up in the program area until you compile it. You can compile all the current files open in the files pane by clicking on the Compile All button near the top of the DrJava window. Compiling your code changes it into instructions that the computer understands and can execute.

Don’t worry if you hit Compile All before you save changes to a file. DrJava won’t compile files until they are saved, so it will give you the chance to save the changes then.

• The bottom part is the interactions pane. This is where you can literally command the computer to do something. You type your commands at the > prompt, and when you hit return, the computer will interpret your words (i.e., apply the meanings and encodings of the Java programming language) and do what you have told it to do. This interpretation will include whatever you typed and compiled in the definitions pane as well.

In English you end sentences with a period. In Java you typically end a programming statement with a semicolon. However, in the interactions pane you can leave off the semicolon and it will print the result of whatever you have typed. If you do add the semicolon at the end of a Java statement in the interactions pane it will execute the statement but not automatically print the result in the interactions pane. Even though you do not have to type the semicolon after statements in the interactions pane you must type the semicolon at the end of your statements in the definitions pane or the code will not compile.

There are other features of DrJava visible in Figure 2.3. The Open button will let you open a file, it will add the file name to the files pane, and show the code in that file in the definitions pane. The Save button will save the file that is currently displayed in the definitions pane. The Javadoc button creates HTML documentation from the Javadoc comments in your files (comments that start with ‘/**’ and end with ‘*/’).

Making it Work Tip: Get to know your Help!
An important feature to already start exploring is the Help. If you click on Help and then click on Help again when a menu is displayed you will see a help window. Start exploring it now so that you have a sense for what’s there.

2.2.5 Using DrJava

We’re going to start out by simply typing commands in the interactions pane—not defining new names yet, but simply using the names and symbols that the computer
Section 2.2 Programming in Java

FIGURE 2.3: DrJava (with annotations)

already knows from within Java (keywords, operators, and classes that come with a release of Java) in addition to a few classes we have written for you to use.

Making it Work Tip: Java Keywords, Operators, and Classes

In Java the useful names that the computer understands are keywords, operators, and classes. All of the keywords defined in Java are completely lowercase. Some example keywords are public, class, static, main, new, and instanceof. Operators in Java include the standard math operators like addition (+), multiplication (*), division (/), subtraction (-) and others. There are also classes that have been defined and are included with a version of Java for you to use. Some of the classes included with Java are String, System, Math, and JFrame. Notice that class names start with an uppercase letter. This is a Java convention (usual way something is done).

Since you will need to provide the semicolon at the end of statements in the definitions pane you should get used to using them in the interactions pane too.
But, then how do you show the result of a statement in the interactions pane? The phrase `System.out.println()` is an important one to know. The meaning for `System.out.println()` is “Use the `PrintStream` object known as `out` on the `System` class to display a readable representation of whatever is in the parentheses, on the console window, followed by an end-of-line character.” DrJava will also print the result in the interactions pane when you use `System.out.println()`.

You can have nothing in the parentheses which will just move the output to a new line, or it can be a name that the computer knows, or an expression (literally, in the algebraic sense). Try typing `System.out.println(34 + 56)` by clicking in the interactions area, typing the command, and hitting return—like this:

```> System.out.println(34 + 56);
90```

Making it Work Tip: Using Another Development Environment
If you aren’t using DrJava you will need to type all code that we show in the interaction pane in a main method instead. Then you will need to execute the class with the main method. So to get the above example to work in another development environment I could written the following class definition, compiled it, and executed it.

```public class Test {
    public static void main (String [] args) {
        System.out.println(34 + 56);
    }
}
```

The ability to try things in the interactions pane without having to create a main method is one of the major advantages to using DrJava.

34 + 56 is a numeric expression that Java understands. Obviously, it’s composed of two numbers and an operation (in our sense, a name) that Java knows how to do, + meaning “add.” Java understands other kinds of expressions, not all numeric. In Java we call math symbols like ‘+’ and ‘-‘ operators.

```> System.out.println(34.1/46.5);
0.7333333333333334
> System.out.println(22 * 33);
726
> System.out.println(14 - 15);
-1
> System.out.println(5 % 2);
1
> System.out.println("Hello");
Hello```
Java understands a bunch of standard math operations. As you might expect '/', is divide, '*' is multiply, '-' is subtract. Java also uses '%' for remainder as in 5 divided by 2 has a remainder of 1. This is also called the modulo operator. Java knows how to recognize different kinds of numbers, both integer and floating point (numbers with a decimal point in them). It also knows how to recognize strings (lists of characters) that are started and ended with " (double quotes). It even knows what it means to “add” two strings together: It simply puts one right after the other (appends them).

Common Bug: Java's types can produce odd results
Java takes types seriously. If it sees you using integers, it thinks you want an integer result from your expressions. If it sees you use floating point numbers, it thinks you want a floating point result. Sounds reasonable, no? But how about:

> System.out.println(1.0/2.0);
0.5
> System.out.println(1/2);
0

The answer to 1/2 is 0? Well, sure! The numbers 1 and 2 are integers. There is no integer equal to 1/2, so the answer must be 0 (the part after the decimal point is thrown away)! Simply by adding “.0” to a number convinces Java that we’re talking about floating point numbers (specifically the Java primitive type double), so the result is in floating point form.

We could also have used casting to get the correct result from the division of two integers. Casting is telling the compiler to use a particular type. To cast you put the type that you want the value changed to inside parentheses. There are two floating point types in Java: float and double. The type double is larger than the type float and thus more accurate. We will use this type for most of the floating point numbers in this book. Notice that we can cast either the 1 or 2 to double and the answer will then be given as a double. We could cast both the 1 and 2 to double and the result would be fine. However, if we cast the result of the integer division to a double it is too late since the result of integer division of 1 by 2 is 0 since the result is an integer.

> System.out.println((double) 1 / 2);
0.5
> System.out.println(1 / (double) 2);
0.5
> System.out.println(((double) (1/2));
Chapter 2 Introduction to Programming

0.0

Making it Work Tip: Java Primitive Types

- Integers are numbers without a decimal point in them. Integers are represented by the types: `int`, `byte`, `short`, or `long`. Example integers are: 3, -2350, 502893. In this book we will use only int to represent integers.

- Floating point numbers are numbers with a decimal point in them. Floating point numbers can be represented by the types: `double` or `float`. Example doubles are 3.0, -19.23, and 548.675. In this book we will use mostly use double to represent floating point numbers.

- Characters are individual characters such as can be made with one key stroke on your keyboard. Characters are represented by the type: `char`. Characters are specified inside single quotes, like 'a' or 'A'.

- True and false values are represented by the type `boolean`. Variables of type boolean can only have `true` or `false` as values.

2.3 USING VARIABLES

We have used Java to do calculations but we have not stored the results. The results would be in memory but we don’t know where they are in memory and we can’t get back to them. On a calculator we can store the result of one calculation using memory store. We can then use that stored value using memory recall. On a calculator you also have access to the result of the last calculation.

On a computer we can store many calculated values by naming them. We can then access those values by using the variable names. The computer takes care of mapping the name to the address that stores the value. We call naming values declaring a variable.

When you declare a variable in Java you specify the type for the variable and a name. You can also assign a value to a variable using the ‘=’ operator and provide a value or an expression. The type you declare tells the computers how much memory to use for storing the value of the variable and how to interpret the bits that make up the value. We will use the type `int` for storing integer values (numbers without decimal points) and `double` for storing floating point values (numbers with decimal points).

What if you want to calculate the total bill for a meal including the tip? You
Section 2.4 Class and Object Methods

would start with the bill value and multiply it by the percentage you want to tip (20%), that would give you the tip amount. You could then add the tip amount to the bill total to get the total amount to leave. We will use type type double to store the bill amount, tip, and total amount since they need to be floating point values. If we also wanted to calculate the cost per person we could divide the total by the number of people. We could use an integer variable to hold the number of people.

```java
> int numPeople = 2;
> System.out.println(numPeople);
2
> double bill = 32.45;
> System.out.println(bill);
32.45
> double tip = bill * 0.2;
> System.out.println(tip);
6.490000000000001
> double total = bill + tip;
> System.out.println(total);
38.940000000000005
> double totalPerPerson = total / numPeople;
> System.out.println(totalPerPerson);
19.470000000000002
```

We don’t have to print out the value of the variable after we assign a value to it. We are doing that so that you see that the computer does return a value when you use the name of a variable. What about the extra amount for the final answer? The answer should be just $19.47 per person. If we look back at the printing of the tip amount we see where this first occurred. Floating point numbers do not always give exact results.

In Java when you declare variables to be of the type `int` or `double` you are asking the computer to set aside space for a variable of that type (32 bits for `int` and 64 for `double`) and to remember the address of that space. When you assign a value to a variable using the `=` operator you change the value in that space to represent the new value. The code `int numPeople` reserves 32 bits of space and associates the name “numPeople” with that reserved space (Figure 2.4). The code `numPeople = 2` sets the value of that space to the integer value 2. The code `double bill` reserves 64 bits of space and associates the name “bill” with that space. The `bill = 32.45` changes the values in the reserved space to represent the value 32.45. When we print out a variable name using `System.out.println(bill);` the computer looks up the name to find the address of that variable in memory and prints the value stored in that space. It knows how many bytes to use in calculating the value based on the declared type of the variable.

2.4 CLASS AND OBJECT METHODS

Java also understands about functions. Remember functions from algebra? They're a “machine or box” into which you put one value, and out comes another. Java
2.4.1 Invoking Class Methods

Class methods can be invoked (executed) by using the class name followed by a period and then the method name: `ClassName.methodName()`. By convention, class names in Java start with an uppercase letter: like `Character`. The `Character` class is a wrapper class for the primitive type `char`. It also provides general class methods.

Making it Work Tip: Wrapper Classes

Wrapper classes are classes that you use to wrap around primitive types in order to have an object to work with. Many general purpose classes in Java such as the collection classes (such as List and Set) require the values that you add to the collections to be objects. Since primitive types are not objects you wouldn’t be able to use them in collections (prior to Java version 1.5). However, if you wrap a primitive type with a wrapper class you will be able to use it with classes that require objects. As of Java version 1.5 the wrapping of a primitive value is automatically done when it is needed.

One of the class methods for the Character class takes a character as the
input value (the value that goes into the box) and returns (the value that comes out of the box) the number that is the integer value for that character. Characters in Java are specified between single quotes: 'A'. The name of that function is **getNumericValue()** and you can use **System.out.println** to display the value that the method **getNumericValue()** returns:

```
> System.out.println(Character.getNumericValue('A'));
10
```

Another class method that's built in to the Math class in Java is named **abs**—it's the absolute value function. It returns the absolute value of the input numeric value.

```
> System.out.println(Math.abs(1));
1
> System.out.println(Math.abs(-1));
1
```

### Debugging Tip: Common typos

If you type a class name and Java can't figure out what class you are taking about you will get an undefined class error.

```
> Mat.abs(-3);
Error: Undefined class 'Mat'
```

If you mistype a method (function) name you will get the following error:

```
> Math.ab(-3);
Error: No 'ab' method in 'java.lang.Math'
```

### 2.4.2 Invoking Object Methods

Object methods are methods that **must** be invoked on an object using:

```
objectName.methodName()
```

You can't invoke object methods using the class name like you can with class methods. You can find out what methods you can invoke (execute) on an object by checking the documentation for the declared class of the object. In Java there is a **String** class which is how you represent lists of characters (letters), like the letters of a name. Objects of the **String** class are created by the compiler whenever it sees string literals (characters enclosed with double quotes), like "Barbara" or "cat.jpg". The double quotes tell the compiler that this is an object of the **String** class and not a variable name.

There are many object methods in the **String** class, such as **toLowerCase()** and **toUpperCase**. These methods actually create and return new **String** objects (objects of the class **String**). See the API (application program interface) for the **String** class for a full listing of the available object methods.
Chapter 2  Introduction to Programming

```java
> String name = "Fred Farmer";
> System.out.println(name);
Fred Farmer
> String lowerName = name.toLowerCase();
> System.out.println(lowerName);
fred farmer
> String upperName = name.toUpperCase();
> System.out.println(upperName);
FRED FARMER
> System.out.println(name);
Fred Farmer
```

Notice that value of `name` didn’t change even though we invoked `toLowerCase` on it. All of the `String` methods that can modify a string don’t change the original string but instead return a new string with the action done on that string. We say that strings are *immutable*, meaning that they don’t change.

Variables that are declared to be of any of the primitive types: `byte`, `short`, `int`, `long`, `float`, `double`, `boolean` or `char` reserve space and associate the variable name with the starting location of that space. Variables that are declared to be of any other type are object variables. This is because all other types inherit from the class `Object`.

![Mammal](image1)

![Dog](image2)

![Object](image3)

![String](image4)

FIGURE 2.5: Showing the parent and child relationship between mammal and dog (left) and `Object` and `String` (right)

You can think of inheritance as saying that a class is a kind of another class like saying that a dog is a kind of mammal (Figure 2.5). If you need a mammal you can use a dog, but if you need a dog any mammal won’t do. Because a dog is a kind of mammal we know that it has the same characteristics that a mammal does such as breathing oxygen, bearing live young, having hair, etc. We say that it inherits characteristics from mammal. The `String` class is a child of the `Object` class so it is a kind of object (Figure 2.5).

Object variables reserve space for something which refers to the address of the object. If the object variable doesn’t refer to any object yet it has the value `null`.

```java
> String test;
> System.out.println(test);
```
null
> test = "Hi";
> System.out.println(test);
Hi
> test = new String("Bye");
> System.out.println(test);
Bye

String test;
Reserves space and
sets the value to null.
Associates "test" with this
space.
test
 test test = "Hi";
Creates a String object and
sets the characters in that
object to "Hi". Changes
the value in the object
reference "test" to refer to the
String object.
test
 test = new String("Bye");
Creates a String object and
sets the characters in that
object to "Bye". Changes
the value in the object
reference "test" to refer to the
String object. The String
object with "Hi" in it still exists
but can be garbage collected
since there is no reference.

FIGURE 2.6: Declaring object variables and memory assignment

When the variable test was declared as type String space was reserved for
an object reference (a way to find the address of the object) and the value was set
to null (Figure 2.6). The default value for an object variable is null which means
it isn’t referring to any object yet. The compiler will create a String object when
it sees characters enclosed in double quotes so the "Hi" creates an object of the
String class and sets the characters in that String object to be the characters
"Hi". The code test = "Hi" changes the value of the space reserved for the object
reference from null to a reference to the String object with the characters "Hi".
You can also create a String object using new String("some letters") where
some letters are the characters that you want in your String object.

What happens to the String object with the characters "Hi" in it when you
changed the variable test to refer to the new String object with the characters
"Bye"? Java keeps track of used space and if there are no valid references to the
used space it will put it back into available space. This is called garbage collection.
The fact that Java automatically handles freeing used memory when it is no longer needed is one of the advantages to Java over languages like C++ which required the programmer to free memory when it was no longer needed. Programmers weren’t very good at keeping track of when memory was no longer needed and so many programs never freed memory when it was no longer needed. This is called a memory leak and it is why some programs use more and more memory while they are running. Sometimes programmers freed memory when it was still being used which can cause major problems such as incorrect results and even cause your computer to crash.

2.5 INVOKING MEDIA METHODS

The method `pickAFile()` is a class method on the FileChooser class. The FileChooser class is one that we created to make it easy for you do things like pick a file name and return a string which represents the full path name of that file. The name of the function (method) that does this is `pickAFile()`. Java is very picky about capitalization—neither `pickafile` nor `Pickafile` will work! Try it like this: `System.out.println(FileChooser.pickAFile());`. When you do, you will get something that looks like Figure 2.7.

**Common Bug: File Chooser doesn’t appear**

If you don’t see the window with the file chooser in it after typing in the code above, try minimizing your DrJava window. Sometimes the file chooser comes up behind the DrJava window.

You’re probably already familiar with how to use a file chooser or file dialog like this:

- Double-click on folders/directories to open them.
Section 2.5  Invoking Media Methods

- Click on the top right iconic button to see the details about the files such as the types of files they are (if you put the cursor over the button and leave it there it will show “Details”). To create a picture we want to pick a file with a type of “JPEG Image”. To create a sound we would pick a file with a type of “Wave Sound”.

- Click on the file name to select it and then click Open, or double-click, to select a file.

Once you select a file, what gets returned is the full file name as a string (a sequence of characters). (If you click Cancel, pickAFile() returns null which is a predefined value in Java that means nothing. Try it, type the code below after the > in the interactions pane and select a file by clicking the mouse button when the cursor points to the desired file name, then click on the Open button.

> System.out.println(FileChooser.pickAFile());
C:\intro-prog-java\mediasources\flower1.jpg

What you get when you finally select a file will depend on your operating system. On Windows, your file name will probably start with C: and will have backslashes in it (e.g., \). There are really two parts to this file name:

- The character between words (e.g., the \ between “intro-prog-java” and “mediasources”) is called the path separator. Everything from the beginning of the file name to the last path separator is called the path to the file. That describes exactly where on the hard disk (in which directory) a file exists. A directory is like a drawer of a file cabinet and it can hold many files. A directory can even hold other directories.

- The last part of the file (e.g. “cat.jpg”) is called the base file name. When you look at the file in the Finder/Explorer/Directory window (depending on your operating system), that’s the part that you see. Those last three characters (after the period) is called the file extension. It identifies the encoding of the file. You may not see the extension depending on the settings you have. But, if you show the detail view (top right iconic button on the file chooser) you will see the file types. Look for files of type “JPEG Image”.

Files that have an extension of “.jpg” or a type of “JPEG Image” are JPEG files. They contain pictures. (To be picky, they contain data that can be interpreted to be a representation of a picture – but that’s close enough to “they contain pictures.”) JPEG is a standard encoding (a representation) for any kind of image. The other kind of media files that we’ll be using frequently are “.wav” files (Figure 2.8). The “.wav” extension means that these are WAV files. They contain sounds. WAV is a standard encoding for sounds. There are many other kinds of extensions for files, and there are even many other kinds of media extensions. For example, there are also GIF (“.gif”) files for images and AIFF (“.aif” or “.aiff”) files for sounds. We’ll stick to JPEG and WAV in this text, just to avoid too much complexity.
2.5.1 Showing a Picture

So now we know how to get a complete file name: Path and base name. This doesn’t mean that we have the file itself loaded into memory. To get the file into memory, we have to tell Java how to interpret this file. We know that JPEG files are pictures, but we have to tell Java explicitly to read the file and make a Picture object from it (an object of the Picture class).

The way we create and initialize new objects in Java is to ask the class to create a new object by new ClassName(parameters). So, to create a new object of the Picture class from a file name use new Picture(fileName). The fileName is the name of a file as a string. We know how to get a file name using FileChooser.pickAFile().

```java
> System.out.println(new Picture(FileChooser.pickAFile()));

Picture, filename
C:\intro-prog-java\mediasources\partFlagSmall.jpg height 217 width 139
```

The result from System.out.println suggests that we did in fact make a Picture object, from a given filename and with a given height and width. Success! Oh, you wanted to actually see the picture? We’ll need another method! (Did I mention somewhere that computers are stupid?) The method to show the picture is named show().

You ask a Picture object to show itself using the method show(). It may seem strange to say that a picture knows how to show itself but in object-oriented
programming we treat objects as intelligent beings that know how to do the things that we would expect an object to be able to do, or that someone would want to do to it. We typically show pictures, so in object-oriented programming Picture objects know how to show themselves (make themselves visible).

2.5.2 Variable Substitution

We can now pick a file, make a picture, and show it in a couple of different ways.

- We can do it all at once because the result from one method can be used in the next method: `new Picture(FileChooser.pickAFile()).show()`. That’s what we see in figure 2.9. This code will first invoke the `pickAFile()` class method of the class `FileChooser` and that will return the name of the selected file as a string. Next it will create a new `Picture` object with the selected file name. And finally it will ask the created `Picture` object to show itself.

![Figure 2.9: Picking, making, and showing a picture, using the result of each method in the next method. The picture used is matt-spaceman.jpg.](image)

- The second way is to name each of the pieces by declaring variables. To declare a variable (a name for data) use `Type name;` or `Type name=something;`.
Making it Work Tip: Types in Java

A type in Java can be any of the predefined primitive types (char, byte, int, short, long, float, double, or boolean) or the name of a class. Java is not a completely object-oriented language in that the primitive types are not objects.

Why are there so many primitive types? The answer has to do with how many bits you want to use to represent a value. The more bits you use the larger the number that you can store. We will only use int, float, double, and boolean in this book. The type int is for integer numbers and takes up 32 bits. The type float is for floating point numbers and takes up 32 bits. The type double is for floating point numbers and takes up 64 bits. The type boolean is for things that are just true or false so a boolean value could be stored in just 1 bit. However, how much space a boolean takes isn’t specified in the Java language specifications (it depends on the virtual machine). Java uses primitive types to speed calculations.

A class name used as a type can be either a class defined as part of the Java language like (String, JFrame, or BufferedImage) or a class that you or someone else created (like the Picture class we created).

We can name the file that we get from FileChooser.pickAFile() by using using (String fileName =). This says that the variable named fileName will be of type String (will refer to an object of the String class) and that the String object that it will refer to will be returned from FileChooser.pickAFile(). In a similar fashion we can create a variable named picture that will refer to an object of the Picture class that we get from creating a new Picture object with the fileName using Picture picture = new Picture(fileName). We can then ask that Picture object to show itself by sending it the show() message using
picture.show(). That’s what we see in figure 2.10.

Making it Work Tip: Java Conventions
By convention all class names in Java begin with an uppercase letter, all variable and method names begin with a lowercase letter. This will help you tell the difference between a class name and a variable or method name. So, Picture is a class name since it starts with an uppercase letter and picture is a variable name since it starts with a lowercase letter. These are two different names to Java. If a name has several words in it the convention is to uppercase the first letter of each additional word like pickAFile(). A convention is the usual way of doing something which means that the compiler won’t care if you don’t do it this way but other programmers will tar and feather you because it will make your programs harder to understand.

Debugging Tip: Methods names must be followed by parentheses!
In Java all methods (functions) have to have parentheses after the method name both when you declare the method and when you invoke it. You can’t leave off the parentheses even if the method doesn’t take any parameters. So, you must type picture.show() not picture.show.

If you try picture.show(), you’ll notice that there is no output from this method. Methods in Java don’t have to return a value, unlike real mathematical functions. A method may just do something (like display a picture).

2.5.3 Object references
When the type of a variable is int or double or boolean we call that a primitive variable. As you have seen when a primitive variable is declared space is reserved to represent that variable’s value and the name is used to find the address of that reserved space.

When the type of a variable is the name of a class (like String) then this is called an object variable or object reference. Unlike primitive variables, object variables do not reserve space for the value of the variable. How could they? How much space do you need for an object? How about an object of the class String? How about an object of the class Picture? The amount of space you need for an object depends on the number and types of fields (data) each object of that class has.

Object variables (references) reserve space for a reference to an object of the given class. A reference allows the computer to determine the address of the actual object (it isn’t just the address of the object). If the object variable is declared but not assigned to an object the reference is set to null which means that it doesn’t
2.5.4 Playing a Sound

We can replicate this entire process with sounds.

- We still use `FileChooser.pickAFile()` to find the file we want and get its file name.

- We use `new Sound(fileName)` to make a `Sound` object using the passed `fileName` as the file to read the sound information from.

- We will use `play()` to play the sound. The method `play()` is an object method (invoked on a `Sound` object). It plays the sound one time. It doesn’t return anything.

Here are the same steps we saw previously with pictures:

```java
> System.out.println(FileChooser.pickAFile());
C:\intro-prog-java\mediasources\croak.wav
> System.out.println(new Sound(FileChooser.pickAFile()));
```
Section 2.5 Invoking Media Methods

Sound file: croak.wav length: 17616
> new Sound(FileChooser.pickAFile()).play();

The `System.out.println(FileChooser.pickAFile());` allows you to pick a file with a file chooser and displays the full file name. The code `System.out.println(new Sound(FileChooser.pickAFile()));` also allows you to pick a file and creates a sound object from the full file name and displays information about the sound object: the file name, and the length of the sound. We'll explain what the length of the sound means in the next chapter. The code `new Sound(FileChooser.pickAFile()).play();` has you pick a file name, creates the sound object using that file name, and tells that sound object to play.

Please do try this on your own, using WAV files that you have on your own computer, that you make yourself, or that came on your CD. (We talk more about where to get the media and how to create it in future chapters.)

Congratulations! You've just worked your first media computation!

2.5.5 Naming your Media (and other Values)

The code `new Sound(FileChooser.pickAFile()).play();` looks awfully complicated and long to type. You may be wondering if there are ways to simplify it. We can actually do it just the way that mathematicians have for centuries: We name the pieces! The results from methods (functions) can be named, and these names can be used as the inputs to other methods.

Since we have already mentioned naming so often, it probably doesn’t come as any surprise that you can create your own names. Later, we’ll show how to name your own methods (functions). Right now, let’s name our data. We call our names for data variables.

We create names (declare variables) by using `type name;` or `type name = expression`. The first way just declares the variable and doesn’t set its’ value. The second way creates the variable and assigns a value to the variable. You can also use `name = expression` to assign a value to the variable. We can check the values of our names (variables) using `System.out.println()`, just as we have been doing.

> int myVariable=12;
> System.out.println(myVariable);
12
> double anotherVariable=34.5;
> System.out.println(anotherVariable);
34.5
> String myName="Mark";
> System.out.println(myName);
Mark

Don’t read = as “equals.” That’s what it means in mathematics, but that’s not at all what we’re doing here. Read = as “set the value for the space associated with this name.” `int myVariable=12` thus means reserve space in memory for an
int (32 bits) and set the value stored in that space to 12.” The reverse (putting the expression on the left and the name on the right) is not allowed and wouldn’t make sense. Variable names usually start with a lowercase alphabetic character.

```java
> int x = 2 * 8;
> System.out.println(x);
16
> 2 * 8 = x;
Syntax Error: ";
```

We can easily reuse names.

```java
> String myName = "Mark";
> System.out.println(myName);
Mark
> myName = "Barb";
> System.out.println(myName);
Barb
```

This actually means to first set the variable myName to refer to the `String` object with the characters "Mark" in it. Then it changes the variable myName to refer to another `String` object with the characters "Barb" in it. The first `String` object with the characters "Mark" in it still exists and can be garbage collected (reused as available space).

You can’t declare the same variable name twice. Declare the name one time (by specifying the type and name) and then you can use it many times.
> String myName = "Mark";
> System.out.println(myName);
Mark
> String myName = "Sue";
Error: Redefinition of 'myName'

The *binding* between the name and the data only exists (a) until the name gets assigned to something else or (b) you quit DrJava or (c) you reset the interactions pane. The relationship between names and data only exists during a session of DrJava.

### Common Bug: Redefinition Error

You can’t declare a variable with the same name more than once in the interactions pane. If you do you will get a “Redefinition Error” like this:

```java
> String fileName = FileChooser.pickAFile();
> Picture picture = new Picture(fileName);
> picture.show();
> String fileName = FileChooser.pickAFile();
Error: Redefinition of 'fileName'
```

If you wanted to “start over” then click the **Reset** button in DrJava to let it know that you want to get rid of all the currently defined variables. Or, just remove the types and you won’t be redeclaring the variables, just changing their values (reusing them):

```java
> String fileName = FileChooser.pickAFile();
> Picture picture = new Picture(fileName);
> picture.show();
> fileName = FileChooser.pickAFile();
> picture = new Picture(fileName);
> picture.show();
```

### 2.5.6 Types Matter

Remember that variables (data) do have encodings or types. Some operators behave differently depending on the types of data they are given to work with.

```java
> System.out.println(2 + 6);
8
> System.out.println("The answer is " + 2 + 6);
The answer is 26
> System.out.println("The answer is " + (2 + 6));
The answer is 8
```

The “+” sign is interpreted as *addition* when it is used between numbers and as *append* when it is used after a string. So `System.out.println(2 + 6)` means...
to add the numbers and print out the answer. But, `System.out.println("The answer is " + 2 + 6)` means append the character 2 to the previous string and then append the character 6 to the resulting string. We can fix this to mean addition by adding parentheses to make our meaning clear: `System.out.println("The answer is " + (2 + 6))`.

Also, notice the difference between how the integer 12 is treated and the string "12" is treated.

```java
> int myVariable=12;
> System.out.println(myVariable+4);
16
> System.out.println(myVariable);
12
> String myOtherVariable="12";
> System.out.println(myOtherVariable+4);
124
> System.out.println(myOtherVariable);
12
> myVariable = myVariable + 10;
> System.out.println(myVariable);
22
```

In `System.out.println(myVariable+4)` the variable `myVariable` has been declared to be an integer and so the “+” operator is interpreted as meaning addition. In `System.out.println(myOtherVariable+4)` the variable `myOtherVariable` has been declared to be an object of the `String` class so the “+” is interpreted to mean append.

Also notice that the value of a variable is substituted for its name in an expression, as it is in `System.out.println(myVariable + 4);`. Doing something with the value stored at a variable doesn’t change the value of that variable. If you want to change the value of a variable you can do this by using the “=” as in `myVariable = myVariable + 10;`. This does change the value stored at `myVariable` as shown by the code `System.out.println(myVariable);`.

### 2.5.7 Naming the Result of a Method

We can assign names to the results of methods (functions). If we name the result from `FileChooser.pickAFile()`, each time we print the name, we get the same result. We don’t have to re-run `FileChooser.pickAFile()`. Naming code in order to re-execute it is what we’re doing when we define methods (functions), which comes up in Section 2.6.

```java
> String fileName = FileChooser.pickAFile();
> System.out.println(fileName);
C:\intro-prog-java\mediasources\beach-smaller.jpg
> System.out.println(fileName);
C:\intro-prog-java\mediasources\beach-smaller.jpg
```
In the below example, we declare variables (assign names) for the file name (a String object) and the Picture object.

```java
> String myFileName = FileChooser.pickAFile();
> System.out.println(myFileName);
C:\intro-prog-java\mediasources\katie.jpg
> Picture myPicture = new Picture(myFileName);
> System.out.println(myPicture);
Picture, filename C:\intro-prog-java\mediasources\katie.jpg height 360 width 381
```

Notice that the algebraic notions of substitution and evaluation work here as well. Picture `myPicture = new Picture(myFileName)` causes the exact same picture to be created as if we had executed `Picture myPicture = new Picture(Chooser.pickAFile())`, because we set `myFileName` to be equal to the result of `FileChooser.pickAFile()`. The values get substituted for the names when the expression is evaluated. `new Picture(myFileName)` is an expression which, at evaluation time, gets expanded into `new Picture ("C:\intro-prog-java\mediasources\katie.jpg")` because `C:\intro-prog-java\mediasources\katie.jpg` is the name of the file that was picked when `FileChooser.pickAFile()` was evaluated and the returned value was named `myFileName`.

We can also replace the method (function) invocations ("function calls") with the value returned. `FileChooser.pickAFile()` returns a `String` object—a bunch of characters enclosed inside of double quotes. We can make the last example work like this, too.

```java
> String myFileName = "C:/intro-prog-java/mediasources/katie.jpg";
> System.out.println(myFileName);
C:/intro-prog-java/mediasources/katie.jpg
> Picture myPicture = new Picture(myFileName);
```

Assuming, of course, that you picked the same file.

**Common Bug: Backslashes and Slashes**

You have seen the names of files displayed with backslashes in them, such as `C:\intro-prog-java\mediasources\beach.jpg`. However, when you create an object of the `String` class in Java you might not want to use backslashes because they are used to create special characters in strings like tab or newline. You can use slashes `/` instead as a path separator `C:/intro-prog-java/mediasources/beach.jpg`. Java can still figure out the path name when you use slashes. You can still use backslashes in the full path name, but you need to double each one `C:\intro-prog-java\mediasources\beach.jpg`.
Chapter 2 Introduction to Programming

> System.out.println(myPicture);
Picture, filename C:/intro-prog-java/mediasources/katie.jpg height 360 width 381

Or even substitute for the name.

> Picture aPicture = new Picture("C:/intro-prog-java/mediasources/katie.jpg");
> System.out.println(aPicture);
Picture, filename C:/intro-prog-java/mediasources/katie.jpg height 360 width 381

We actually don’t need to use System.out.println() every time we ask the computer to do something. If we want to call a function that doesn’t return anything (and so is pretty useless to System.out.println()), we can just call the method (function) by typing its name and its input (if any) in parentheses followed by a semicolon and hitting return.

> aPicture.show();

We tend to call these statements to the computer that are telling it to do things commands. System.out.println(aPicture) is a command. So is String myFileName = FileChooser.pickAFile(), and aPicture.show(). These are more than expressions: They’re telling the computer to do something.

2.6 MAKING A PROGRAM

We have now used names to stand for values by declaring variables. The values get substituted for the variable names when the expression is evaluated. We can do the same for programs. We can name a series of commands, so that we can just use the name whenever we want the commands to be executed. This is exactly what defining a program or method is about.

Remember when we said earlier that just about anything can be named in
computers? We’ve seen naming values. Now we’ll see naming programs (methods).

Making it Work Tip: Try every program!
To really understand what’s going on, type in, compile, and execute every program (method) in the book. EVERY one. None are long, and the practice will go a long way towards convincing you that the programs work, developing your programming skill, and helping you understand why they work.

The way that Java defines the name of a new program is by declaring a method inside a class definition. In object-oriented programming we need to decide who (what class) is going to do the program as well as what are the steps to take in doing the program. An object-oriented program is more like a large restaurant where certain chefs specialize in the types of recipes they create. You might have a desert chef and a French chef. Each class in an object-oriented program understands the recipes (methods) defined inside of it.

You have seen how you declare variables in Java
Type name; or Type name = value;

To declare a method in Java use:
visibility Type methodName
(outputList)

The structure of how you declare a method is referred to as the syntax — the words and characters that have to be there for Java to understand what’s going on, and the order of those things.

A method declaration usually has a visibility (usually the keyword public or private), the type of the thing being returned from the method, the method name, and the parameter list in parentheses. This is usually followed by a block which has curly braces around the series of commands you want to have executed when the method is invoked.

Visibility means who can invoke the method (ask for the method to be executed). If the keyword public is used this method can be invoked by any code in any class definition. If the keyword private is used then the method can only be accessed from inside the class definition. You can think of this as a security feature. If you keep your journal on the web (a blog) then it is open and anyone can read it. If you keep it hidden in your room then it is private and only you can read it.

There are two types of methods in Java. One is a class method and the other an object method. Class methods operate on class fields and object methods operate on object fields. Object methods are implicitly passed the current object (accessed by the this keyword). To declare a class method you add the keyword static to the method declaration. To declare an object method you leave off the static keyword. The static keyword is usually given after the visibility.

The code to declare an object method such as show() for the Picture class which doesn’t return a value and has no input parameters would be public void show(). The class method pickAFile() which is a class method in the File-Chooser class and returns a String object is declared as public static String
pickAFile().

The return type is required and is given before the method name. If you leave off a return type you will get a compiler error. If your method returns a value the return type must match the type of the value returned. Remember that types can be any of the primitive types (char, byte, int, short, long, float, double, or boolean) or a class name. Methods that don’t return any value use the Java keyword void for the return type in the method declaration.

By convention method names start with a lowercase letter and the first letter of each additional word is uppercase: FileChooser.pickAFile(). The name of this method is pickAFile. The first word is all lowercase and the first letter of each additional word is capitalized.

A method must have parentheses following the method name. If any parameters are passed to the method then they will be declared inside the parentheses separated by commas. To declare a parameter you must give a type and name.

We create a collection of commands by defining a block. A block is code between an open curly brace '{' and a close curly brace '}'. The block of commands that follow a method declaration are the ones associated with the name of the method (function) and are the ones that will be executed when the method is invoked.

Most real programs that do useful things, especially those that create user interfaces, require the definition of more than one method (function). Imagine that in the definitions pane you have several method declarations. How do you think Java will figure out that one function has ended and a new one begun? Java needs some way of figuring out where the method body ends: Which statements are part of this method and which are part of the next? Java uses curly braces to do this. All statements between the open curly brace and close curly brace are part of the method body.

### Debugging Tip: Proper Method Declarations
All method declarations must be inside a class definition which means that they are defined inside the open '{' and close '}' curly braces that enclose the body of the class definition. If you put a method declaration after the end of the class definition you will get “Error: 'class' or 'interface' expected”. Methods can not be defined inside of other methods. If you accidentally do this you will get “Error: illegal start of expression” at the beginning of the inner method declaration. Statements in a method end in a semicolon (this is not optional in the definitions pane). If you forget to put the semicolon at the end of a statement you will get “Error: ';' expected”. All compiler errors will highlight the line of code that caused the error. If you don’t see the error on that line of code check the preceding line. You can double click on an error in the “Compiler Output” area and it will place the cursor at that line of code and highlight it.

We can now define our first program! Open Picture.java by clicking on the
Section 2.6 Making a Program 51

Test the new method here.

Type the method here before the last ‘}’

Compile by clicking here

FIGURE 2.12: Defining and executing pickAndShow()

Open button near the top of the window and using the file chooser to pick “Picture.java”. Type the following code into the definitions pane of DrJava before the last closing curly brace ‘}’ (which ends the class definition). When you’re done, save the file and click the Compile All button near the top of the window (Figure 2.12).

Compiling a Java class definition “Picture.java” will produce a “Picture.class” file. Compiling translates the Java source code which is in a format that humans understand into a format that computers understand. One of the advantages to Java is that the “.class” files aren’t specific to any particular type of computer. They can be understood by any computer that has a Java run-time environment. So you can create your Java source code on a Window's based computer and run the compiled code on an Apple computer.

Program 2: Pick and show a picture

```java
public static Picture pickAndShow ()
{
    String fileName = FileChooser.pickAFile ();
    Picture picture = new Picture (fileName);
    picture.show();
    return picture;
}
```

This code creates a class method with the name pickAndShow that takes no
parameters and whenever the method is executed it will execute the commands inside of the open and close curly braces. It returns the created Picture object. This method is a class method (the method declaration contains the keyword static) because it creates the picture object inside of it and returns it. Object methods must be called on an object of the class and since we don’t have a Picture object when this method starts it can’t be an object method.

Once the method has successfully compiled you can execute it (Figure 2.12). Click on the INTERACTIONS tab in the interactions pane (near the bottom of the window). Since this is a class method (because of the keyword static in the method declaration) you can execute the method by using the class name (Picture) followed by a dot (period) and then the method name. This method doesn’t take any parameters so just finish with the open and close parenthesis and the semicolon.

Whenever the Java run-time encounters a new class it will first load the class definition from the “.class” file. This creates an object that represents the new class. This object knows how to create objects of that class and contains the code for the methods defined in that class. So we we type Picture.pickAndShow(); the “Picture.class” file will be loaded and an object that represents the Picture class will be created. Next, the location of the method pickAndShow() is found and the instructions at that location are executed.

> Picture.pickAndShow();

We can similarly define our second program, to pick and play a sound. Open the Sound class definition file and type the following before the last closing curly brace ’}’ in the file. Then click the Compile All button to compile the file. You can test this new class method (because of the static keyword) using the class name Sound followed by dot (period) and then the method name. This method doesn’t have any parameters so use Sound.pickAndPlay(); to test the new method.

Program 3: Pick and play a sound

```java
public static Sound pickAndPlay()
{
    String aFileName = FileChooser.pickAFile();
    Sound aSound = new Sound(aFileName);
    aSound.play();
    return aSound;
}
```
Making it Work Tip: Name the names you like
You’ll notice that, in the first method we were using the names `fileName` and `picture`. In this program, we used `aFileName` and `aSound`. Does it matter? Absolutely not!
Well, not to the computer, at any rate. The computer doesn’t care what names you use—they’re entirely for your benefit. Pick names that (a) are meaningful to you (so that you can read and understand your program), (b) are meaningful to others (so that others can read and understand it), and (c) are easy to type. 25-character names, like, `myPictureThatIAmGoingToOpenAfterThis` are meaningful, easy-to-read, but are a pain to type.

Does this mean that you can use names like ”yellow” for a file name and ”orange” for a `Picture` object? Yes, you can but it may be confusing even for you and especially confusing for others. It helps to use variable names that indicate the type of thing the variable represents which is why we are using ”picture” to name a variable that is a `Picture` object and ”fileName” to name a variable that is a name of a file (a `String` object).

While cool, this probably isn’t the most useful thing for you. Having to pick the file over-and-over again is just annoying. But now that we have the power of programs, we can define new ones however we like! Let’s define one that will just open a specific picture, and another that opens a specific sound.

Use `FileChooser.pickAFile()` to get the file name of the sound or picture file. We’re going to need the file name when we define the program to play that specific sound or show that specific picture. We’ll just set the value of `fileName` directly, instead of as a result of `FileChooser.pickAFile();`, by putting the string between double quotes directly in the method.

Type in the following code before the last closing curly brace ‘}’ in the Picture.java file. Be sure to replace `FILENAME` below with the complete path to your own picture file, e.g., “C:/intro-prog-java/mediasources/katie.jpg”. Remember to use slashes instead of backslashes in your file name or double each backslash.

```java
Program 4: Show a specific picture

public static Picture showSpecificPicture()
{
    String myFile = "FILENAME";
    Picture myPicture = new Picture(myFile);
    myPicture.show();
    return myPicture;
}
```
Making it Work Tip: Copying and pasting

Text can be copied and pasted between the interactions pane and definitions pane. You can use `System.out.println(FileChooser.pickAFile());` to print a filename, then select it and copy it (from the Edit menu), then click in the definitions pane and paste it. Similarly, you can copy whole commands from the interactions pane up to the definitions pane: That’s an easy way to test the individual commands, and then put them all in a method once you have the order right and they’re working. You can also copy entire methods in the definitions pane by selecting the text in the method and then copying and pasting it. You can select a method name in the definitions pane and paste it in the interactions pane to execute the method.

Compile Picture.java. The method `showSpecificPicture` is also a class method (it has the keyword ‘static’ in the declaration), so it can be executed using the class name followed by a dot and then the method name.

> Picture.showSpecificPicture()

Type in the following code before the last ‘}’ in the Sound.java file. Be sure to replace `FILENAME` below with the complete path to your own sound file, e.g., "C:/intro-prog-java/mediasources/thisisatest.wav". Remember to use slashes instead of backslashes in the file name or double each backslash.

```
public static Sound playSpecificSound() {
    String myFile = "C:/intro-prog-java/mediasources/thisisatest.wav";
    Sound mySound = new Sound(myFile);
    mySound.play();
    return mySound;
}
```

Compile “Sound.java” using the Compile All button. The method `playSpecificSound` is a class method (it has the keyword static in the declaration), so it can be executed using the class name followed by a dot and then the method name. Do the following in the interactions pane:

> Sound.playSpecificSound()
2.6.1 Variable Programs: Real functions that Take Input

How do we create a method (function) that takes input variables, like Math.abs(-1) or new Picture(fileName)? Why would you want to?

An important reason for using input variables is to make a program more general. Consider method 2.6, Picture.showSpecificPicture(). That’s for a specific file name. Would it be useful to have a function that could take any file name, then make and show the picture? That kind of function handles the general case of making and showing pictures. We call that kind of generalization abstraction. Abstraction leads to general solutions that work in lots of situations.

Defining a method that takes input is very easy. It continues to be a matter of substitution and evaluation. We’ll put a type and name inside those parentheses after the method name. The names given inside the parentheses are called the parameters or input variables.

When you evaluate the function, by specifying its name with an input value (also called the argument) inside parentheses (like new Picture(myFileName) or new Sound(aFileName)), the parameter variable is set to a copy of the argument value. This is called pass by value. All arguments in Java are passed by making a copy of their value. Does this mean that we make a copy of the file name when we pass it as a parameter? No, we just make a copy of the value of the object reference which means we make another reference to that file name which is a String object.

Here’s what a method would look like that takes a String object that is a file name as an input variable:

```
Program 6: Show the picture file whose file name is input

public static Picture showNamed(String fileName)
{
    Picture myPicture = new Picture(fileName);
    myPicture.show();
    return myPicture;
}
```

Add this to Picture.java before the last curly brace that closes the class definition and compile. Execute it with:

```
> Picture.showNamed("C:/intro-prog-java/mediasources/katie.jpg");
```

When I type:

```
Picture.showNamed("C:/intro-prog-java/mediasources/katie.jpg")
```

and hit return, the variable fileName refers to the String object with a value of "C:/intro-prog-java/mediasources/katie.jpg". myPicture will then be assigned to the picture resulting from reading and interpreting the file with that name. Then the picture is shown and the Picture object returned.
Chapter 2  Introduction to Programming

We can play a sound file in the same way.

Program 7: Play the sound file whose file name is input

```java
public static Sound playNamed(String fileName)
{
    Sound mySound = new Sound(fileName);
    mySound.play();
    return mySound;
}
```

Is it better to specify a file name inside a method or input (pass) a filename to a method? And what does “better” mean here, anyway? If you want to show a different file using the method `showSpecificPicture()` you have to edit the method to specify the new filename and then recompile. Or you could copy the method and create another method with a different name that shows a different file. The only differences between these two methods would be the method name and the file name and if you want to show many files you would need many methods that are very similar so this doesn’t sound like a good idea. If you want to show a different file using the method `showNamed(String fileName)` you can simply pass in a different filename. You don’t have to edit the method or recompile. So, `showNamed(String fileName)` is more general and reusable than `showSpecificPicture()` because it takes an input parameter.

OBJECTS AND METHODS SUMMARY

In this chapter, we talk about several kinds of encodings of data (or objects).
Section 2.6 Making a Program

### File name
Java `String` object
A filename is just a string, an instance (object) of the `String` class, whose characters represent a path, path separators, and a base file name.

<table>
<thead>
<tr>
<th>Floating point numbers</th>
<th>Java primitive type <code>double</code> e.g., 5.2, -3.01, 928.3092</th>
<th>Numbers with a decimal point in them.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integers</td>
<td>Java primitive type <code>int</code> e.g., -3, 5239, 0</td>
<td>Numbers without a decimal point—they can’t represent fractions.</td>
</tr>
<tr>
<td>Pictures</td>
<td>objects of our <code>Picture</code> class</td>
<td>Pictures are encodings of images, typically coming from a JPEG file.</td>
</tr>
<tr>
<td>Sounds</td>
<td>objects of our <code>Sound</code> class</td>
<td>Sounds are encodings of sounds, typically coming from a WAV file.</td>
</tr>
<tr>
<td>Strings</td>
<td>Java <code>String</code> object e.g., &quot;Hello!&quot;</td>
<td>A sequence of characters (including spaces, punctuation, etc.) delimited on either end with a double quote character.</td>
</tr>
</tbody>
</table>

Here are the methods (functions) introduced in this chapter:

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Character.getNumericValue(Character character)</code></td>
<td>Returns the equivalent numeric value in Unicode for the input character.</td>
</tr>
<tr>
<td><code>Math.abs(int number)</code></td>
<td>Takes a number and returns the absolute value of it.</td>
</tr>
<tr>
<td><code>FileChooser.pickAFile()</code></td>
<td>Lets the user pick a file and returns the complete path name as a string.</td>
</tr>
<tr>
<td><code>new Picture(String fileName)</code></td>
<td>Takes a name of a file as input, reads the file, and creates a picture from it (an instance of the <code>Picture</code> class). Returns the created <code>Picture</code> object.</td>
</tr>
<tr>
<td><code>picture.show()</code></td>
<td>Shows the <code>Picture</code> object that it is invoked on. No return value.</td>
</tr>
<tr>
<td><code>new Sound(String fileName)</code></td>
<td>Takes a filename as input, reads the file, and creates a sound from it. Returns the sound object.</td>
</tr>
<tr>
<td><code>sound.play()</code></td>
<td>Plays the sound object (object of the <code>Sound</code> class) that it is invoked on.</td>
</tr>
</tbody>
</table>
Chapter 2  Introduction to Programming

PROBLEMS

2.1. Some computer science concept questions:
- What is an algorithm?
- What is a program?
- What’s an encoding?
- What is Moore’s Law?
- What is a variable? Why would you want one?

2.2. Test your understanding of Java with the following:
- What does `System.out.println()` mean?
- What does the statement `System.out.println(3 + 2)` do?
- What does the code `int x = 3; System.out.println("The result is " + x)` do?
- What does the statement `System.out.println(" " + 3 + 2)` do?
- What does `picture.show()` do?
- What does `FileChooser.pickAFile()` do?

2.3. Which of the following are class methods and which are object methods? How do you tell which are which?
- `Math.abs(-3);`
- `sound.play();`
- `FileChooser.pickAFile();`
- `picture.show();`
- `Picture.pickAndShow();`
- `Picture.showNamed();`

2.4. How many and what kind of variables (primitive or object) are created in the code below? Draw what the variables look like.
```java
> String fileName = FileChooser.pickAFile();
> Picture p = new Picture(fileName);
> p.show();
```

2.5. You can combine the sound playing and picture showing commands in the same method. Try playing a sound and then show a picture while a sound is playing. Try playing a sound and opening several pictures while the sound is still playing. The hardest decision is where to put this method. It could go on the Sound or Picture class or perhaps it would go on a new MixedMedia class. Where does it make sense to put it?

2.6. We evaluated the expression `FileChooser.pickAFile()` when we wanted to invoke the method named `pickAFile()`. But what does this do? Open the FileChooser class and find the method declaration.

2.7. Try the `Sound.playNamed()` method. You weren’t given any examples of its use, but you should be able to figure it out from `Picture.showNamed()`.
TO DIG DEEPER

The best (deepest, most material, most elegant) computer science textbook is Structure and Interpretation of Computer Programs [2], by Abelson, Sussman, and Sussman. It’s a hard book to get through, though. Somewhat easier, but in the same spirit is the new book How to Design Programs [8].

Neither of these books are really aimed at students who want to program because it’s fun or because they have something small that they want to do. They’re really aimed at future professional software developers. The best books aimed at the less hardcore user are by Brian Harvey. His book Simply Scheme uses the same programming language as the earlier two, Scheme, but is more approachable. My favorite of this class of books, though, is Brian’s three volume set Computer Science Logo Style [15] which combine good computer science with creative and fun projects.

There is a wealth of material for Java on Sun’s Java web site http://java.sun.com including tutorials, papers, and APIs. To learn more about DrJava see the web site http://www.drjava.org/. Thinking in Java by Bruce Eckel is a good book for those who have some coding experience and like to understand a language deeply. Beginners might want to start with Headfirst Java by Kathy Sierra and Bert Bates.
PART TWO

PICTURES

Chapter 3  Modifying Pictures using Loops
Chapter 4  Modifying Pixels in a Matrix
Chapter 5  Conditionally Modifying Pixels
CHAPTER 3
Modifying Pictures using Loops

3.1 HOW PICTURES ARE ENCODED
3.2 MANIPULATING PICTURES
3.3 CHANGING COLOR VALUES

Chapter Learning Objectives
The media learning goals for this chapter are:

- To understand how images are digitized by taking advantage of limits in human vision.
- To identify different models for color, including RGB, the most common one for computers.
- To manipulate color values in pictures, like increasing or decreasing red values.
- To convert a color picture to grayscale, using more than one method.
- To convert a color picture to its negative representation.

The computer science goals for this chapter are:

- To introduce arrays.
- To write object methods.
- To do iteration with while and for loops.
- To introduce comments.
- To understand the scope of a variable name.
- To introduce breaking a method into smaller methods.

3.1 HOW PICTURES ARE ENCODED

Pictures (images, graphics) are an important part of any media communication. In this chapter, we discuss how pictures are represented on a computer (mostly as bitmap images—each dot or pixel is represented separately) and how they can be manipulated.

Pictures are two-dimensional arrays of pixels (which is short of picture element). In this section, each of those terms will be described.
Section 3.1 How Pictures are Encoded

For our purposes, a picture is an image stored in a JPEG file. JPEG is an international standard for how to store images with high quality but in little space. JPEG is a lossy compression format. That means that it is compressed, made smaller, but not with 100% of the quality of the original format. Typically, though, what gets thrown away is stuff that you don’t see or don’t notice anyway. For most purposes, a JPEG image works fine.

An array is a sequence of elements, each with an index number associated with it. The first element in an array is at index 0, the second at index 1, the third at index 2, and so on. The last element of the array will always be at the length of the array minus one. An array with 5 elements will have its last element at index 4. It may sound strange to say that the first element of an array is at index 0 but the index is based on the distance from the beginning of the array to the element. Since the first item of the array is at the beginning of the array the distance is 0.

![Figure 3.1: A depiction of the first five elements in an array](image)

Every time you join a line (queue) of people you are in something like an array. All you usually care about is how far you are from the front of the line. If you are at the front of the line then that is index 0 (you are next). If you are the second one in line then you are at index 1 (there is one person in front of you). If you are the third person in line then you are at index 2 (there are two people in front of you).

Arrays are a great way to store lots of data of the same type. You wouldn’t want to create a different variable for every pixel in a picture when there are thousands of pixels in a picture. Instead you use an array of pixels. You still need a way to refer to a particular pixel so we use an index for that. You can access elements of an array in Java using `arrayName[index]`. For example, to access the first element in an array variable named pixels use `pixels[0]`. To access the second element use `pixels[1]`. To access the third element use `pixels[2]`. You can get the number of items in an array using `arrayName.length`. So, to access the last element in the
array use `pixels[arrayName.length - 1]`.

Making it Work Tip: Using dot notation for public fields
Notice that there are no parentheses following `arrayName.length`. This is because length is not a method but a public field (data). Public fields can be accessed using dot notation `objectName.fieldName`. Methods always have parenthesis after the method name even if there are no input parameters, such as `FileChooser.pickAFile()`.

A two-dimensional array is a `matrix`. A matrix is a collection of elements arranged in both a horizontal and vertical sequence. For one dimensional arrays you would talk about an element at index `i`, that is `array[i]`. For two-dimensional arrays you talk about an element at column `i` and row `j`, that is, `matrix[i][j]`.

In Figure 3.2, you see an example matrix. At coordinates (0, 0) (horizontal, vertical), you’ll find the matrix element whose value is 15. The element at (1, 1) is 7, (2, 1) is 43, and (3, 1) is 23. We will often refer to these coordinates as `(x, y)` (`(horizontal, vertical)`).

![Figure 3.2: An example matrix (two-dimensional array) of numbers](image)

Have you ever played the game battleship? If you have then you had to specify both the row and column of your guess (I-5) This means row 1 and column 5. Have you ever gone to a play? Usually your ticket has a row and seat number (row E, seat 10). These are both examples of two-dimensional arrays. Picture data is also represented as a two-dimensional array.

What’s stored at each element in the picture is a `pixel`. The word “pixel” is short for “picture element.” It’s literally a dot, and the overall picture is made up of lots of these dots. Have you ever taken a magnifying glass to pictures in the newspaper or magazines, or to a television or even your own computer monitor? (Figure 3.3 was generated by capturing as an image the of the top left part of the DrJava window and then magnifying it 600%. It’s made up of many, many dots. When you look at the picture in the magazine or on the television, it doesn’t look like it’s broken up into millions of discrete spots, but it is.
Section 3.1 How Pictures areEncoded

FIGURE 3.3: Upper left corner of DrJava window with part magnified 600%

You can get a similar view of individual pixels using the picture explorer, which is discussed later in this chapter. The picture explorer allows you to zoom a picture up to 500% so that each individual pixel is visible (Figure 3.4).

FIGURE 3.4: Image shown in the picture explorer: 100% image on left and 500% on right (close-up of the branch over the mountain)

Our human sensor apparatus can’t distinguish (without magnification or other special equipment) the small bits in the whole. Humans have low visual *acuity*—we don’t see as much detail as, say, an eagle. We actually have more than one kind of vision system in use in our brain and our eyes. Our system for processing color is different than our system for processing black-and-white (or *luminance*). We actually pick up luminance detail better with the sides of our eyes than the center of our eye. That’s an evolutionary advantage since it allows you to pick out the sabertooth sneaking up on you from the side.

That lack of resolution in human vision is what makes it possible to digitize pictures. Animals that perceive greater details than humans (e.g., eagles or cats) may actually see the individual pixels. We break up the picture into smaller elements (pixels), but there are enough of them and they are small enough that the picture doesn’t look choppy when looked at it overall. If you can see the effects
of the digitization (e.g., lines have sharp edges, you see little rectangles in some spots), we call that *pixelization*—the effect when the digitization process becomes obvious.

Picture encoding is actually more complex than sound encoding. A sound is inherently linear—it progresses forward in time. It can be represented using a one-dimensional array. A picture has two dimensions, a width and a height.

### 3.1.1 Color Representations

Visible light in continuous—visible light is any wavelength between 370 and 730 nanometers (0.00000037 and 0.00000073 meters). But our perception of light is limited by how our color sensors work. Our eyes have sensors that trigger (peak) around 425 nanometers (blue), 550 nanometers (green), and 560 nanometers (red). Our brain determines what a particular color based on the feedback from these three sensors in our eyes. There are some animals with only two kinds of sensors, like dogs. Those animals still do perceive color, but not the same colors nor in the same way as humans do. One of the interesting implications of our limited visual sensory apparatus is that we actually perceive two kinds of orange. There is a *spectral* vision—a particular wavelength that is natural orange. There is also a mixture of red and yellow that hits our color sensors just right that we perceive as the same orange.

Based on how we perceive color, as long as we encode what hits our three kinds of color sensors, we’re recording our human perception of color. Thus, we can encode each pixel as a triplet of numbers. The first number represents the amount of red in the pixel. The second is the amount of green, and the third is the amount of blue. We can make up any human-visible color by combining red, green, and blue light (Figure 3.5) (replicated at Figure 4.17 (page 153). Combining all three gives us pure white. Turning off all three gives us black. We call this the *RGB color model*.

![Figure 3.5: Merging red, green, and blue to make new colors](image)

There are other models for defining and encoding colors besides the RGB color model. There’s the *HSV color model* which encodes Hue, Saturation, and Value (sometimes also called the *HSB color model* for Hue, Saturation, and Brightness).
The nice thing about the HSV model is that some notions, like making a color “lighter” or “darker” map cleanly to it, e.g., you simply change the saturation (Figure 3.6). Another model is the **CMYK color model**, which encodes Cyan, Magenta, Yellow, and black (“B” could be confused with Blue). The CMYK model is what printers use—those are the inks they combine to make colors. However, the four elements means more to encode on a computer, so it’s less popular for media computation. RGB is the most popular model on computers.

**FIGURE 3.6**: Picking colors using the HSB color model

Each color component (sometimes called a *channel*) in a pixel is typically represented with a single byte, eight bits. Eight bits can represent 256 patterns ($2^8$): 00000000, 00000001, up through 11111111. We typically use these patterns to represent the values 0 to 255. Each pixel, then, uses 24 bits to represent colors. That means that there are $2^{24}$ possible patterns of 0’s and 1’s in those 24 bits. That means that the standard encoding for color using the RGB model can represent 16,777,216 colors. We can actually perceive more than 16 million colors, but it turns out that it just doesn’t matter. Humans have no technology that comes even close to being able to replicate the whole color space that we can see. We do have devices that can represent 16 million distinct colors, but those 16 million colors don’t cover the entire space of color (nor luminance) that we can perceive. So, the 24 bit RGB model is adequate until technology advances.

There are computer models that use more bits per pixel. For example, there are 32 bit models which use the extra 8 bits to represent *transparency*—how much of the color “below” the given image should be blended with this color? These additional 8 bits are sometimes called the *alpha channel*. There are other models that actually use more than 8 bits for the red, green, and blue channels, but they are uncommon.

We actually perceive borders of objects, motion, and depth through a *separate* vision system. We perceive color through one system, and *luminance* (how light/dark things are) through another system. Luminance is not actually the *amount* of light, but our *perception* of the amount of light. We can measure the amount of light (e.g., the number of photons reflected off the color) and show that a red and a blue spot each are reflecting the same amount of light, but we’ll perceive the blue as
darker. Our sense of luminance is based on comparisons with the surroundings—the optical illusion in Figure 3.7 highlights how we perceive gray levels. The two end quarters are actually the same level of gray, but because the two mid quarters end in a sharp contrast of lightness and darkness, we perceive that one end is darker than the other.

FIGURE 3.7: The ends of this figure are the same colors of gray, but the middle two quarters contrast sharply so the left looks darker than the right

Most tools for allowing users to pick out colors let the users specify the color as RGB components. The Macintosh offers RGB sliders in its basic color picker (Figure 3.8). The color chooser in Java offers a similar set of sliders (Figure 3.9).

FIGURE 3.8: The Macintosh OS X RGB color picker

As mentioned a triplet of (0, 0, 0) (red, green, blue components) is black, and (255, 255, 255) is white. (255, 0, 0) is pure red, but (100, 0, 0) is red, too—just darker. (0, 100, 0) is a dark green, and (0, 0, 100) is a dark blue.

When the red component is the same as the green and as the blue, the resultant color is gray. (50, 50, 50) would be a fairly dark gray, and (150, 150, 150) is a lighter gray.

The Figure 3.10 (replicated at Figure 4.18 (page 153) in the color pages) is a representation of pixel RGB triplets in a matrix representation. Thus, the pixel at (1, 0) has color (30, 30, 255) which means that it has a red value of 30, a green value of 30, and a blue value of 255—it’s a mostly blue color, but not pure blue. Pixel at (2, 1) has pure green but also more red and blue ((150, 255, 150)), so it’s a fairly light green.

Images on disk and even in computer memory are usually stored in some kind
Section 3.1  How Pictures are Encoded  69

FIGURE 3.9: Picking a color using RGB sliders from Java

FIGURE 3.10: RGB triplets in a matrix representation

of compressed form. The amount of memory needed to represent every pixel of even small images is pretty large (Table 3.1). A fairly small image of 320 pixels across by 240 pixels wide, with 24-bits per pixel, takes up 230,400 bytes—that’s roughly 230 kilobytes (1000 bytes) or 1/4 megabyte (million bytes). A computer monitor with 1024 pixels across and 768 pixels vertically with 32-bits per pixel takes up over 3
### 3.2 MANIPULATING PICTURES

We manipulate a picture in DrJava by making a picture object out of a JPEG file, then changing the pixels in that picture. We change the pixels by changing the color associated with the pixel—by manipulating the red, green, and blue components.

We make a picture using `new Picture(fileName)`. We make the picture appear with the object method `show()`. We can also explore a picture with the object method `explore()`. Remember that object methods must be called on an object of the class that contains the method. This means that `show()` and `explore()` must be called on a `Picture` object (object of the `Picture` class) using dot notation as in `pictureObject.show()`.

```java
> String fileName = FileChooser.pickAFile();
> System.out.println(fileName);
c:\intro-prog-java\mediasources\caterpillar.jpg
> Picture pictureObject = new Picture(fileName);
> pictureObject.show();
> System.out.println(pictureObject);
Picture, filename c:\intro-prog-java\mediasources\caterpillar.jpg
height 150 width 329
```

What `new Picture(fileName)` does is to scoop up all the bytes in the input filename, bring them in to memory, reformat them slightly, and place a sign on
them “This is a picture object!” When you execute `Picture pictureObject = new Picture(fileName);`, you are saying “The name pictureObject is referring to a Picture object created from the contents of the file.”

Picture objects know their width and their height. You can query them with the object methods `getWidth()` and `getHeight()`.

```java
> System.out.println(pictureObject.getWidth());
329
> System.out.println(pictureObject.getHeight());
150
```

We can get any particular pixel from a picture using `getPixel(x, y)` where x and y are the coordinates of the pixel desired. The x coordinate starts at 0 at the top left of the picture and increasing horizontally. The y coordinate starts at 0 at the top left of the picture and increases vertically. We can also get a one-dimensional array containing all the pixels in the picture using the method `getPixels()`. This just grabs all the pixels in the first column from top to bottom and then all the pixels in the second column from top to bottom and so on.

```java
> Pixel pixelObject = pictureObject.getPixel(0,0);
> System.out.println(pixelObject);
Pixel red=252 green=254 blue=251
> Pixel[] pixelArray=pictureObject.getPixels();
> System.out.println(pixelArray[0]);
Pixel red=252 green=254 blue=251
```

Pixels know where they came from. You can ask them their x and y coordinates with `getX()` and `getY()`.

```java
> System.out.println(pixelObject.getX());
0
> System.out.println(pixelObject.getY());
0
```

Each pixel object knows how to get the red value `getRed()` and set the red value `setRed(redValue)`. (Green and blue work similarly.)

```java
> System.out.println(pixelObject.getRed());
252
> pixelObject.setRed(0);
> System.out.println(pixelObject.getRed());
0
```

You can ask a pixel object for its color with `getColor()`, and you can ask the pixel object to set the color with `setColor(color)`. Color objects (objects of the class `Color` in package `java.awt`) know their red, green, and blue components. You can also create new Color objects with `new Color(redValue, greenValue, blueValue)` (the color values must be between 0 and 255). The Color class also has several colors predefined that you can use. If you need a color object that represents the
color black you can use `Color.black` or `Color.BLACK`, for yellow use `Color.yellow` or `Color.YELLOW`. Other colors that are predefined are: `Color.blue`, `Color.green`, `Color.red`, `Color.gray`, `Color.orange`, `Color.pink`, `Color.cyan`, `Color.magenta`, and `Color.white` (or use all capitals for the color names). Notice that this is accessing fields on the `Color` class, not invoking class methods (no parentheses). Public class variables (fields) can be accessed using `ClassName.fieldName`.

### Making it Work Tip: Importing Classes from Packages
`Color` is a Java class in the package `java.awt`. A package is a group of related classes. To use classes in packages other than `java.lang` (which contains `System` and `Math`) you will need to `import` them. Importing a class or all classes in a package allows you to use the name of a class without fully qualifying it. To fully qualify a name use the package name followed by a period (dot) and the class name. The `fully qualified name` for the `Color` class is `java.awt.Color`. You can always use the fully qualified name instead of importing but people don’t usually want to type that much. To import all classes in the package `java.awt` use `import java.awt.*`. To import just the `Color` class from the package `java.awt` use `import java.awt.Color;`. Importing doesn’t make your class larger, it is just used to determine what class you mean.

### Debugging Tip: Undefined Class Error
If you get the message “Error: Undefined class Color” it means that you didn’t import the class `Color`. You must either import classes that are in packages other than `java.lang` or fully qualify them. To import just the class `Color` so that you can refer to it using just the class name use `import java.awt.Color;`. If you are using several classes from the same package you can import all classes in a package using `import java.awt.*;`. You can type the import statement in the interactions pane. When you write class files the import statements go before the class definition at the beginning of the file.

```java
> import java.awt.Color;
> Color colorObj=pixelObject.getColor();
> System.out.println(colorObj);
java.awt.Color[r=0,g=254,b=251]
> Color newColorObj=new Color(0,100,0);
> System.out.println(newColorObj);
java.awt.Color[r=0,g=100,b=0]
> pixelObject.setColor(newColorObj);
```
Section 3.2 Manipulating Pictures

> System.out.println(pixelObject.getColor());
java.awt.Color[r=0,g=100,b=0]

If you change the color of a pixel, the picture that the pixel is from does get changed. However you won’t see the change until the picture repaints.

> System.out.println(pictureObject.getPixel(0,0));
Pixel red=0 green=100 blue=0

Common Bug: Seeing changes in the picture
If you show your picture, and then change the pixels, you might be wondering, “Where are the changes???” Picture displays don’t automatically update. If you ask the Picture object to repaint using pictureObject.repaint(), the display of the Picture object will update.

You can automatically get a darker or lighter color from a Color object with colorObj.darker() or colorObj.brighter(). (Remember that this was easy in HSV, but not so easy in RGB. These functions do it for you.)

> Color testColorObj = new Color(168,131,105);
> System.out.println(testColorObj);
java.awt.Color[r=168,g=131,b=105]
> testColorObj = testColorObj.darker();
> System.out.println(testColorObj);
java.awt.Color[r=117,g=91,b=73]
> testColorObj = testColorObj.brighter();
> System.out.println(testColorObj);
java.awt.Color[r=167,g=130,b=104]

Notice that even though we darken the color and then brighten it the final color doesn’t exactly match the original color. This is due to rounding errors. A rounding error is when calculations are done in floating point but the answer is stored in an integer. The floating point result can’t fit in the type of the result (integer) and so some of the detail is lost.

You can also get a color using ColorChooser.pickAColor(), which gives you a variety of ways of picking a color. ColorChooser is a class that we have created to make it easy for you to pick colors using the Java class javax.swing.JColorChooser.

> import java.awt.Color;
> Color pickedColorObj = ColorChooser.pickAColor();
> System.out.println(pickedColorObj);
java.awt.Color[r=51,g=255,b=102]

When you have finished manipulating a picture, you can write it out to a file with write(fileName).

> pictureObject.write("newPicture.jpg");
Chapter 3  Modifying Pictures using Loops

**Common Bug: End with .jpg**
Be sure to end your filename with “.jpg” in order to get your operating system to recognize it as a JPEG file.

**Common Bug: Saving a file quickly—and how to find it again!**
What if you don’t know the whole path to a directory of your choosing? You don’t have to specify anything more than the base name. The problem is finding the file again! In what directory did it get saved? This is a pretty simple bug to resolve. The default directory (the one you get if you don’t specify a path) is wherever DrJava is.

We don’t have to write new functions to manipulate pictures. We can do it from the command area using the methods (functions) just described. Please reset the interactions pane by clicking the Reset button at the top of DrJava before you do the following.

```java
> import java.awt.Color;
> String fileName = "C:/intro-prog-java/mediasources/caterpillar.jpg";
> Picture picture = new Picture(fileName);
> picture.show();
> picture.getPixel(10,100).setColor(Color.black);
> picture.getPixel(11,100).setColor(Color.black);
> picture.getPixel(12,100).setColor(Color.black);
> picture.getPixel(13,100).setColor(Color.black);
> picture.getPixel(14,100).setColor(Color.black);
> picture.getPixel(15,100).setColor(Color.black);
> picture.getPixel(16,100).setColor(Color.black);
> picture.getPixel(17,100).setColor(Color.black);
> picture.getPixel(18,100).setColor(Color.black);
> picture.getPixel(19,100).setColor(Color.black);
> picture.repaint();
> picture.explore();
```

**Making it Work Tip: Reuse the previous line in DrJava**
You can use the up arrow on the keyboard to bring up previous lines you have typed in the interactions pane in DrJava. You can then use the left arrow key to get to a character to correct or change and then execute it by pressing the 'Enter' key.

The result showing a small black line on the left side below the middle of the
leaf appears in Figure 3.11. The black line is 100 pixels down, and the pixels 10 though 19 from the left edge have been turned black.

![Figure 3.11: Directly modifying the pixel colors via commands: Note the small black line on the left under the line across the leaf](image)

3.2.1 Exploring pictures

On your CD, you will find the MediaTools application with documentation for how to get it started. You can also open a picture explorer in DrJava. Both the MediaTools application and the picture explorer will let you get pixel information from a picture. You can see the picture explorer in Figure 3.12 and the MediaTools application appears in Figure 3.13. Both of these will display the x, y, red, green, and blue values for a pixel. They will also both let you zoom in or out.

![Figure 3.12: Exploring the caterpillar with the line](image)

The picture explorer can be opened on a Picture object. Picture p = new Picture(FileChooser.pickAFile()); will allow you to define a Picture object and name it p. You can open a picture explorer on the picture using p.explore(). The picture explorer will make a copy of the current picture and show it. The copy will not be affected by any changes you make to the picture.

The picture explorer allows you to zoom at various levels of magnification, by choosing one in the ZOOM menu. As you move your cursor around in the picture, press down with the mouse button. You’ll be shown the (x, y) (horizontal, vertical) coordinates of the pixel your mouse cursor is currently over, and the red, green, and blue values at that pixel. You can use the next and previous buttons to change
the pixel that you want to examine. You can also type in the x and y values and press 'Enter' to see the pixel information for a particular pixel.

The MediaTools application works from files on the disk. If you want to check out a file before loading into DrJava, use the MediaTools application. Click on the Picture Tools box in MediaTools, and the tools will open. Use the Open button to bring up a file selection box—you click on directories you want to explore on the left, and images you want on the right, then click OK. When the image appears, you have several different tools available. Move your cursor over the picture and press down with the mouse button.

- The red, green, and blue values will be displayed for the pixel you’re pointing at. This is useful when you want to get a sense of how the colors in your picture map to numeric red, green, and blue values. It’s also helpful if you’re going to be doing some computation on the pixels and want to check the values.

- The x and y position will be display for the pixel you’re point at. This is useful when you want to figure out regions of the screen, e.g., if you want to process only part of the picture. If you know the range of x and y coordinates where you want to process, you can tune your program to reach just those sections.

- Finally, a magnifier is available to let you see the pixels magnified. (The magnifier can be clicked and dragged around.)

![Image Exploration Tools](image.png)

**FIGURE 3.13:** Using the MediaTools image exploration tools

### 3.3 Changing Color Values

The easiest thing to do with pictures is to change the color values of their pixels by changing the red, green, and blue components. You can get radically different
Section 3.3 Changing color values

effects by simply tweaking those values. Many of Adobe Photoshop’s *filters* do just what we’re going to be doing in this section.

The way that we’re going to be manipulating colors is by computing a *percentage* of the original color. If we want 50% of the amount of red in the picture, we’re going to set the red channel to 0.50 times whatever it is right now. If we want to increase the red by 25%, we’re going to set the red to 1.25 times whatever it is right now. Recall that the asterisk (*) is the operator for multiplication in Java.

### 3.3.1 Using while loops in pictures

What we could do is to get each pixel in the picture and change its red value. Let’s say that we want to decrease the red by 50%. We can always write code like this:

```java
String fileName = "C:/intro-prog-java/mediasources/caterpillar.jpg";
Picture pict = new Picture(fileName);
pict.show();
Pixel currPixel = pict.getPixel(55,27);
int redValue = currPixel.getRed();
currPixel.setRed((int) (redValue * 0.5));
currPixel = pict.getPixel(55,28);
redValue = currPixel.getRed();
currPixel.setRed((int) (redValue * 0.5));
currPixel = pict.getPixel(55,29);
redValue = currPixel.getRed();
currPixel.setRed((int) (redValue * 0.5));
pict.repaint();
```

**Making it Work Tip: Variable Names**

Earlier we used `pictureObject` and `pixelObject` as variable names for a `Picture` object and pixel object. These are fine names but a little long. We were using them to point out that these are object variables. Above we used `pict` and `currPixel` as variable names for a `Picture` object and pixel object. The names don’t matter to the computer. The computer wouldn’t care if you named the objects yellow and blue. However, it is best to use names that make sense to people and indicate what the variables represent.

The above code is pretty tedious to write, especially for all the pixels in even a small image. What we need is way of telling the computer to do the same thing over and over again. Well, not exactly the same thing—we want to change what’s going on in a well-defined way. We want to take one step each time, or process one additional pixel.

We can do that with a *while* loop. A *while* loop executes a statement (command) or group of commands in a block (inside open and close curly braces). A *while* loop continues executing until a continuation test is false. When the
continuation test is false execution continues with the statement following the while loop.

The syntax for a while loop is:

```
while (test) {
    /* commands to be done go here */
}
```

Let’s talk through the pieces here.

- First comes the required Java keyword while.
- Next we have a required opening parenthesis
- Next is the continuation test. While this test is true the loop will continue to be executed. When this test is false the loop will finish and the statement following the body of the loop will be executed.
- Next is the required closing parenthesis.
- Usually this is followed by a block of commands to be executed each time the expression following the while keyword is true. The block of commands is enclosed by curly braces. This is called the body of the loop. If there is only one command to be executed you may leave off the curly braces but you should still indent the command to show it is in the body of the while loop.

Tell someone to clap their hands 12 times. Did they do it right? How do you know? In order to tell if they did it right you would have to count each time they clapped and when they stopped clapping your count would be 12 if they did it right. A while loop often needs a counter to count the number of times you want something done and an expression that stops when that count is reached. You wouldn’t want to declare the count variable inside the while loop because you want it to change each time through the loop. Typically you declare the count variable just before the while loop and then increment it just before the end of the block of commands you want to repeat.

Computer Science Idea: Flowcharts
Figure 3.14 shows the flowchart of a while loop. A flowchart is a visual representation of the execution of a method or function. It shows the order in which statements are executed and branches or conditional execution. Normal statements are shown in rectangles. Tests are shown in diamonds and have a true branch which is executed when the test is true and a false branch that is executed when the test is false. A flowchart can help you understand what a method is doing.

So, a typical while loop will look like the following code.
Section 3.3 Changing color values

int count = 0;
while (count < target)
{
    // commands to be done inside loop
    count = count + 1;
}

What if you want to write out the same sentence 5 times. You know how to print out a string using System.out.println("some string"); So, put this in the body of the loop. Start the count at 0 and increment it each time after the string is printed. When the count is 5 the string will have been printed 5 times so stop the loop.

> int count = 0;
> while (count < 5)
{  
    System.out.println("This is a test.");
    count = count + 1;
}
This is a test.
This is a test.
This is a test.
This is a test.
This is a test.
Chapter 3  Modifying Pictures using Loops

This is a test.

Debugging Tip: Stopping an Infinite Loop
If you forget to increment the count in the body of the while loop, or if you close the body of the while loop before the count is incremented you will have an infinite loop. An infinite loop is one that will never stop. You can tell that you are in an infinite loop because many more than 5 copies of “This is a test.” will be printed. To stop an infinite loop click on the Reset button near the top of the DrJava window.

What if we want to change the color of all the pixels in a picture? Picture objects understand the method getPixels() which returns a one dimensional array of pixel objects. Even though the pixels are really in a two-dimensional array (a matrix) getPixels() puts the pixels in a one-dimensional array to make them easy to process if we just want to process all the pixels. We can get a pixel at a position in the array using pixelArray[index] with the index starting at 0 and changing each time through the loop by one until it is equal to the length of the array of pixels. Instead of calling the variable “count” we will call it “index” since that is what we are using it for. It doesn’t matter to the computer but it makes the code easier for other people to understand.

Here is the while loop that simply sets each pixel’s color to black in a picture.

```java
> import java.awt.Color;
> String fileName = "C:/intro-prog-java/mediasources/caterpillar.jpg";
> Picture pict = new Picture(fileName);
> pict.show();
> Pixel[] pixelArray = pict.getPixels();
> Pixel pixel = null;
> int index = 0;
> while (index < pixelArray.length)
> {
>     pixel = pixelArray[index];
>     pixel.setColor(Color.black);
>     index++;
> }
> pict.repaint();
```

Let’s talk through this code.

- We will be using the Color class so we need to either use the fully qualified name (java.awt.Color) or import the Color class using import java.awt.Color;
- Next we declare a variable with the name fileName to refer to the string object that has a particular file name “C:/intro-prog-java/mediasources/caterpillar.jpg” stored in it.
- The variable pict is created and refers to the new Picture object created from the picture information in the file named by the variable fileName.
We tell the `Picture` object to show (display) itself using `pict.show();`.

Next we declare a variable `pixelArray` that references an array of `Pixel` objects (`Pixel[]`). We get the array of `Pixel` objects by asking the `Picture` object for them using the `getPixels()` method.

We declare an object variable, `Pixel pixel`, that will refer to a pixel object but initialize it to `null` to show that it isn’t referring to to any pixel object yet.

We declare a primitive variable `index` and initialize its value to 0.

Next we have the `while` loop. First we test if the value of `index` is less than the length of the array of pixels with `while (index < pixelArray.length)`. While it is, we set the variable `pixel` to refer to the pixel object at the current value of `index` in the array of pixel objects. Next we set the color of that pixel to the color black. Finally, we increment the variable index. Eventually the value of the variable `index` will equal the length of the array of pixels and then execution will continue after the body of the loop. Remember that in an array of 5 items the valid indexes are 0-4 so when the index is equal to the length of the array you need to stop the loop.

The statement after the body of the `while` loop will ask the `Picture` object `pict` to repaint so that we can see the color change.

### Debugging Tip: Loops and Variable Declarations

Declare any variables that you will need before you start the loop. While loops typically need some sort of counter or index declared outside the loop but changed inside the loop. If you forgot to change the counter or index you will end up with a loop that never stops. This is called an infinite loop. Use the Reset button to stop if your code is in an infinite loop.

Now that we see how to get the computer to do thousands of commands without writing thousands of individual lines, let’s do something useful with this.

#### 3.3.2 Increasing/decreasing red (green, blue)

A common desire when working with digital pictures is to shift the redness (or greenness or blueness—but most often, redness) of a picture. You might shift it higher to “warm” the picture, or to reduce it to “cool” the picture or deal with overly-red digital cameras.

The method below decreases the amount of red by 50% in the current picture.

### Program 8: Decrease the amount of red in a picture by 50%
Chapter 3  Modifying Pictures using Loops

```java
/**
 * Method to decrease the red by half in the current picture
 */
public void decreaseRed()
{
    Pixel[] pixelArray = this.getPixels();
    Pixel pixel = null;
    int value = 0;
    int index = 0;

    // loop through all the pixels
    while(index < pixelArray.length)
    {
        // get the current pixel
        pixel = pixelArray[index];

        // get the value
        value = pixel.getRed();

        // decrease the red value by 50% (1/2)
        value = (int) (value * 0.5);

        // set the red value of the current pixel to the new value
        pixel.setRed(value);

        // increment the index
        index = index + 1;
    }
}
```

Go ahead and type the above into your DrJava definitions pane before the last curly brace in the Picture.java file. Click Compile ALL to get DrJava to compile the new method. Why do we have to compile the file before we can use the new method? Computers don’t understand the Java source code directly. We must compile it which translates the class definition from something people can read and understand into something a computer can read and understand. When we successfully compile a ClassName.java file the compiler outputs a ClassName.class file which contains the instructions that a computer can understand. If our compile is not successful we will get error messages that explain what is wrong. We have
Section 3.3 Changing color values

### Making it Work Tip: Comments in Java

You may notice that there are some interesting characters in the `reduceRed` method. The `/**` and `//'` are comments in Java. Comments are descriptions of what your code is doing. Use comments to make the code easier to read and understand (not only for yourself but also for others). There are actually three kinds of comments in Java. The `//'` starts a comment and tells the computer to ignore everything following till the end of the line. You can use `/*` followed at some point by `*/` for a multi-line comment. The `/**` followed at some point by `*/` creates a JavaDoc comment. JavaDoc is a utility that pulls the JavaDoc comments from your class files and creates hyperlinked documentation from them. All of the Java class files written by Sun have JavaDoc comments in them and that is how the API documentation was created.

This program works on a `Picture` object—the one that we’ll use to get the pixels from. To create a `Picture` object, we pass in the filename. After we ask the picture to `decreaseRed()`, we’ll want to repaint the picture to see the effect. Therefore, the `decreaseRed` method can be used like this:

```java
> String fileName = "C:/intro-prog-java/mediasources/caterpillar.jpg";
> Picture picture = new Picture(fileName);
> picture.explore();
> picture.decreaseRed();
> picture.explore();
```

### Common Bug: Patience: loops can take a long time

The most common bug with this kind of code is to give up and quit because you don’t think the loop is working. It might take a full minute (or two!) for some of the manipulations we’ll do—especially if your source image is large.

The original picture and its red-decreased version appear in Figure 3.15 (and at Figure 4.19 on page 154). 50% is obviously a lot of red to reduce! The picture looks like it was taken through a blue filter.
Chapter 3  Modifying Pictures using Loops

FIGURE 3.15: The original picture (left) and red-decreased version (right)

Computer Science Idea: Changing memory doesn’t change the file
If you create another Picture object from the same file will you get the original picture or the picture with red decreased? You will get the original picture. The Picture object picture was created by reading the file data into memory. The change to the Picture object was done in memory, but the file wasn’t changed. If you want to save your changes write them out to a file using the method pictObj.write(String fileName); where pictObj is the name of the Picture object and fileName is the full path name of the file. So to save the changed Picture object above use picture.write("c:/caterpillarChanged.jpg");.

Tracing the program: How did that work?.

Computer Science Idea: The most important skill is tracing
The most important skill that you can develop in programming is the ability to trace your program. (This is sometimes also called stepping or walking through your program. To trace your program is to walk through it, line-by-line, and figure out what happens. Looking at a program, can you predict what it’s going to do? You should be able to by thinking through what it does.

Let’s trace the method to decrease red and see how it worked. We want to start tracing at the point where we just called decreaseRed()

> String fileName = "C:/intro-prog-java/mediasources/caterpillar.jpg";
> Picture picture = new Picture(fileName);
> picture.show();
> picture.decreaseRed();

What happens now? `picture.decreaseRed()` really means invoking the `decreaseRed` method which you have just added to the Picture.java file on the `Picture` object referred to by the variable `picture`. The picture object is implicitly passed to the `decreaseRed` method and can be referenced by the keyword `this`. What does “implicitly passed” mean? It means that even though `decreaseRed` doesn’t have any parameters listed it is passed the `Picture` object it was invoked on. So, `picture.decreaseRed()` is like `decreaseRed(Picture this)`. All object methods (methods without the keyword `static` in them) are implicitly passed the object that they are invoked on and that object can be referred to as `this`.

```java
/**
 * Method to decrease the red by half in the current picture
 */
public void decreaseRed ()
{
    Pixel[] pixelArray = this.getPixels();
    Pixel pixel = null;
    int value = 0;
    int index = 0;

    // loop through all the pixels
    while (index < pixelArray.length)
    {
        // get the current pixel
        pixel = pixelArray[index];

        // get the value
        value = pixel.getRed();

        // decrease the red value by 50% (1/2)
        value = (int) (value * 0.5);

        // set the red value of the current pixel to the new value
        pixel.setRed(value);

        // increment the index
        index = index + 1;
    }
}
```

The first line we execute is `Pixel[] pixelArray = this.getPixels()`. Let's break this down.

- The `Pixel[] pixelArray` is a declaration of a variable `pixelArray` that references an array of `Pixel` objects. The `=` means that the variable `pixelArray` will be initialized to the result of the right side expression which is `this.getPixels()` which returns a one-dimensional array of `Pixel` object in the current `Picture` object.
• The `this` is a keyword that represents the current object. Since the method declaration doesn’t have the keyword `static` in it this is an object method. Object methods are always implicitly passed the current object (the object the method was invoked on). In this case the method `decreaseRed()` was invoked by `picture.decreaseRed();` so the `Picture` object referenced by the variable `picture` is the current object. We could leave off the `this` and get the same result. If you don’t reference any object when invoking a method the compiler will assume you mean the current object (referenced by the `this` keyword).

• The `this.getPixels()` invokes the method `getPixels()` on the current object. This method returns a one-dimensional array of `Pixel` objects which are the pixels in the current `Picture` object.

So at the end of the first line we have a variable `pixelArray` that refers to an array of `Pixel` objects. The `Pixel` objects came from the `Picture` object which was referred to as `picture` in the interaction pane and as `this` in the method `decreaseRed()`.

```
    pixelArray
      ↓
Pixel  Pixel  Pixel
   r=252  r=253  r=254
  gr=254  gr=255  gr=254
 b=251  b=254  b=254
```

Next is a declaration of a couple of variables that we will need in the for loop. We will need something to represent the current `Pixel` object so we declare a variable `pixel` of type `Pixel` by `Pixel pixel = null;` We start it off referring to nothing by using the defined value `null`. We also will need a variable to hold the current red value and we declare that as `int value = 0;`. We initialize the variable `value` to be 0. Finally we declare a variable to be the index into the array and the value that changes in the loop `int index = 0;`. Remember that array element are indexed starting with 0 and ending at the length of the array minus one.

Variables that you declare inside methods are not automatically initialized for you so you **should** initialize them when you declare them.
Computer Science Idea: Scope

The names inside a method like pixel and value are completely different than the names in the interactions pane or any other method. We say that they have a different scope. The scope of a variable is the area in which the variable is known. The variables that we declare inside of a method are only known from where they are declared until the end of the method. Variables declared in the interactions pane are known in the interactions pane until it is reset or you exit DrJava.

Next comes the loop while (index < pixelArray.length). This tests if the value of the variable index is less than the length of the array of pixels referred to by pixelArray. If the test is true the body of the loop will be executed. The body of the loop is all the code between the open and close curly braces following the test. If the test is false, execution continues after the body of the loop.

In the body of the loop we have pixel = pixelArray[index];. This will set the pixel variable to point to a Pixel object in the array of pixels with an index equal to the current value of index. Since index is initialized to 0 before the loop the first time through this loop the pixel variable will point to the first Pixel object in the array.

Next in the body of the loop is value = pixel.getRed();. This sets the variable value to the amount of red in the current pixel. Remember that the amount of red can vary from a minimum of 0 to a maximum of 255.
Next in the body of the loop is \texttt{value = (int) (value * 0.5);}. This sets the variable \texttt{value} to the integer amount that you get from multiplying the current contents of \texttt{value} by 0.5. The \texttt{(int)} is a cast to integer so that the compiler doesn’t complain about losing precision since we are storing a floating point number in an integer number. Any numbers after the decimal point will be discarded. We do this because colors are represented as integers. The \texttt{(int) (value * 0.5)} is needed because the variable \texttt{value} is declared of type \texttt{int} and yet the calculation of \texttt{(value * 0.5)} contains a floating point number and so will automatically be done in floating point. However, a floating point result (say of 1.5) won’t fit into a variable of type \texttt{int}. So, the compiler won’t let us do this without telling it that we really want it to by including the \texttt{(int)}. This is called casting and is required whenever a larger value is being placed into a smaller variable. So if the result of a multiplication has a fractional part that fractional part will just be thrown away so that the result can fit in an \texttt{int}.

The next step in the body of the loop is \texttt{pixel.setRed(value);} . This changes the amount of red in the current pixel to be the same as what is stored in variable \texttt{value}. The current pixel is the first one so we see that the red value has changed from 252 to 126 after this line of code is executed.

After the statements in the body of the loop are executed the \texttt{index = index}
Section 3.3  Changing color values  89

+ 1; will be executed which will add one to the current value of index. Since index was initialized to 0 this will result in index holding the value 1.

What happens next is very important. The loop starts over again. The continuation test will again check that the value in variable index is less than the length of the array of pixels and since the value of index is less than the length of the array, the statements in the body of the loop will be executed again. The variable pixel will be set to the pixel object in the array of pixels at index 1. This is the second Pixel object in the array pixelArray.

The variable value will be set to the red amount in the current pixel referred to by the variable pixel, which is 253.

The variable value will be set to the result of casting to integer the result of multiplying the amount in value by 0.5. This results in \((253 \times 0.5) = 126.5\) and after we drop the digits after the decimal this is 126. We drop the digits after the decimal point because of the cast to the type int (integer). We cast to integer because colors are represented as integer values from 0 to 255.
The red value in the current pixel is set to the same amount as what is stored in \textit{value}. So the value of red in the second pixel changes from 253 to 126.

The variable \textit{index} is set to the result of adding 1 to its current value. This adds 1 to 1 resulting in 2.

At the end of the loop body we go back to the continuation test. The test will be evaluated and if the result is true the commands in the loop body will be executed again. If the continuation test evaluates to false execution will continue with the first statement after the body of the loop.

Eventually, we get Figure 3.15 (and at Figure 4.19 on page 154). We keep going through all the pixels in the sequence and changing all the red values.

\textbf{Testing the program: Did that really work?}.

How do we know that that really worked? Sure, \textit{something} happened to the
Section 3.3  Changing color values

FIGURE 3.16: Using the picture explorer to convince ourselves that the red was decreased picture, but did we really decrease the red? By 50%?

Making it Work Tip: Don’t just trust your programs!
It’s easy to mislead yourself that your programs worked. After all, you told the computer to do a particular thing, you shouldn’t be surprised if the computer did what you wanted. But computers are really stupid—they can’t figure out what you want. They only do what you actually tell them to do. It’s pretty easy to get it almost right. Actually check.

We can check it several ways. One way is with the picture explorer. Create two Picture objects: Picture p = new Picture(Chooser.pickAFile()); and Picture p2 = new Picture(Chooser.pickAFile()); and pick the same picture each time. Decrease red in one of them. Then open a picture explorer on each of the Picture objects using p.explore(); and p2.explore();.

We can also use the functions that we know in the Interactions pane to check the red values of individual pixels.

```java
> String fileName = "C:/intro-prog-java/mediasources/caterpillar.jpg";
> Picture pict = new Picture(fileName);
> Pixel pixel = pict.getPixel(0,0);
> System.out.println(pixel);
Pixel red=252 green=254 blue=251
> pict.decreaseRed();
> Pixel newPixel = pict.getPixel(0,0);
> System.out.println(newPixel);
Pixel red=126 green=254 blue=251
> System.out.println( 252 * 0.5);
126.0
```
Increasing red.

Let’s increase the red in the picture now. If multiplying the red component by 0.5 decreased it, multiplying it by something over 1.0 should increase it. I’m going to apply the increase to the exact same picture, to see if we can reduce the blue (Figure 3.17 and Figure 4.20).

Program 9: Increase the red component by 30%

```java
/**
 * Method to increase the amount of red by 30%
 */
public void increaseRed()
{
    Pixel[] pixelArray = this.getPixels();
    Pixel pixel = null;
    int value = 0;
    int index = 0;

    // loop through all the pixels
    while (index < pixelArray.length)
    {
        // get the current pixel
        pixel = pixelArray[index];

        // get the value
        value = pixel.getRed();

        // change the value to 1.3 times what it was
        value = (int) (value * 1.3);

        // set the red value to 1.3 times what it was
        pixel.setRed(value);

        // increment the index
        index++;
    }
}
```

This method works much the same was as the method decreaseRed. We set up some variables that we will need such as the array of pixel objects, the current pixel, the current value, and the current index. We loop through all the pixels in the array of pixels and change the red value for each pixel to 1.3 times its original.
Section 3.3 Changing color values

FIGURE 3.17: Overly blue (left) and red increased by 30% (right)

value.

Making it Work Tip: Shortcuts for Increment and Decrement
Adding one or subtracting one from a current value is something that is done frequently in programs. Programmers have to do lots of typing so they try to reduce the amount of typing that they have to do for things they do frequently. Notice the `index++;` in the increase red program. This has the same result as `index = index + 1;` and can also be written as `++index;`. You can also use `index--;` or `--index;` which will have the same result as `index = index - 1;`. Be careful of using this when you are also assigning the result to a variable. If you do `int x = index++;` x will be assigned the original value of index and then index will be incremented. If you do `int x = ++index;` first index will be incremented and then the value assigned to x.

Compile the new method `increaseRed` and first use `decreaseRed` and then `increaseRed` on the same picture. Explore the picture objects to check that `increaseRed` worked. Remember that the method explore makes a copy of the picture and allows you to check the color values of individual pixels.

```java
> String fileName = "C:/intro-prog-java/mediasources/caterpillar.jpg";
> Picture picture = new Picture(fileName);
> picture.decreaseRed();
> picture.explore();
> picture.increaseRed();
> picture.explore();
```

We can even get rid of a color completely. The method below erases the blue component from a picture by setting the blue value to 0 in all pixels(Figure 3.18...
Chapter 3  Modifying Pictures using Loops

and Figure 4.21).

Program 10: Clear the blue component from a picture

```java
/**
 * Method to clear the blue from the picture (set the blue to 0 for all pixels)
 */
public void clearBlue() {
    Pixel[] pixelArray = this.getPixels();
    Pixel pixel = null;
    int index = 0;

    // loop through all the pixels
    while (index < pixelArray.length) {
        // get the current pixel
        pixel = pixelArray[index];

        // set the blue on the pixel to 0
        pixel.setBlue(0);

        // increment index
        index++;
    }
}
```

Compile the new method `clearBlue` and invoke it on a `Picture` object. Explore the picture object to check that all the blue values are indeed 0.

> String fileName = "C:/intro-prog-java/mediasources/caterpillar.jpg";
> Picture picture = new Picture(fileName);
> picture.explore();
> picture.clearBlue();
> picture.explore();

This method is also similar to the `decreaseRed` and `increaseRed` methods except that we don’t need to get out the current blue value since we are simply setting all the blue values to 0.

3.3.3 Creating a sunset

We can certainly do more than one color manipulation at once. Mark wanted to try to generate a sunset out of a beach scene. His first attempt was to increase the red, but that doesn’t always work. Some of the red values in a given picture are pretty high. If you go past 255 for a channel value it will keep the value at 255.
FIGURE 3.18: Original (left) and blue erased (right)

His second thought was that maybe what happens in a sunset is that there is less blue and green, thus emphasizing the red, without actually increasing it. Here was the program that we wrote for that:

```
/**
 * Method to simulate a sunset by decreasing the green and blue
 */
public void makeSunset()
{
Pixel[] pixelArray = this.getPixels();
Pixel pixel = null;
int value = 0;
int i = 0;

// loop through all the pixels
while (i < pixelArray.length)
{
    // get the current pixel
    pixel = pixelArray[i];

    // change the blue value
    value = pixel.getBlue();
    pixel.setBlue((int)(value * 0.7));

    // change the green value
    value = pixel.getGreen();
    pixel.setGreen((int)(value * 0.7));

    // increment the index
    i++;
```
Chapter 3  Modifying Pictures using Loops

FIGURE 3.19: Original beach scene (left) and at (fake) sunset (right)

Making it Work Tip: Using short variable names for loop counters
Notice that instead of using index as the counter for the loop we are using i. Again, programmers like to reduce the amount of typing and so the simple variable name i is commonly used to represent the counter or index for a loop.

Compile the new method makeSunset and invoke it on a Picture object. Explore the picture object to check that the blue and green values have been decreased.

> String fileName = "C:/intro-prog-java/mediasources/beach-smaller.jpg";
> Picture picture = new Picture(fileName);
> picture.explore();
> picture.makeSunset();
> picture.explore();

What we see happening in Method 11 is that we’re changing both the blue and green channels—reducing each by 30%. The effect works pretty well, as seen in Figure 3.19 (and in the color section at Figure 4.22).

3.3.4 Making sense of methods
You probably have lots of questions about methods at this point. Why did we write these methods in this way? How is that we’re reusing variable names like pixel in both the method and Command Area? Are there other ways to write these methods? Is there such a thing as a better or worse method?
Section 3.3  Changing color values  97

Since we’re always picking a file (or typing in a filename) then making a picture, before calling one of our picture manipulation functions, and then showing or repainting the picture, it’s a natural question why we’re not building those in. Why doesn’t every method have `String fileName = FileChooser.pickAFile();` and `new Picture(fileName);` in it?

We actually want to write the methods to make them more general and reusable. We want our methods to do one and only one thing, so that we can use the method again in a new context where we need that one thing done. An example might make that clearer. Consider the program to make a sunset (Method 11). That works by reducing the green and blue, each by 30%. What if we rewrote that method so that it called two smaller methods that just did the two pieces of the manipulation? We’d end up with something like Method 12.

```
Program 12: Making a sunset as three methods

/**
 * Method to decrease the green in the picture by 30%
 */
public void decreaseGreen() {
    Pixel[] pixelArray = this.getPixels();
    Pixel pixel = null;
    int value = 0;
    int i = 0;

    // loop through all the pixels in the array
    while (i < pixelArray.length) {
        // get the current pixel
        pixel = pixelArray[i];

        // get the value
        value = pixel.getGreen();

        // set the green value to 70% of what it was
        pixel.setGreen((int) (value * 0.7));

        // increment the index
        i++;
    }
}

/**
 * Method to decrease the blue in the picture by 30%
 */
public void decreaseBlue() {
    Pixel[] pixelArray = this.getPixels();
```
Pixel pixel = null;
int value = 0;
int i = 0;

// loop through all the pixels in the array
while (i < pixelArray.length)
{
  // get the current pixel
  pixel = pixelArray[i];

  // get the value
  value = pixel.getBlue();

  // set the blue value to 70% of what it was
  pixel.setBlue((int) (value * 0.7));
}

/**
 * Method to make a picture look like it was taken at sunset
 * by reducing the blue and green to make it look more red
 */
public void makeSunset2()
{
  decreaseGreen();
  decreaseBlue();
}

The first thing to note is that this actually does work. makeSunset2() does the same thing here as in the previous method. It’s perfectly okay to have one method (makeSunset2() in this case) use other methods in the same class (decreaseBlue() and decreaseGreen()). You use makeSunset2() just as you had before. It’s the same algorithm (it tells the computer to do the same thing), but with different methods. The earlier program did everything in one method, and this one does it in three. In fact, you can also use decreaseBlue() and decreaseGreen() by themselves too—make a picture in the Command Area and invoke either method on the Picture object. They work just like decreaseRed().

What’s different is that the method makeSunset() is much simpler to read. That’s very important.

Computer Science Idea: Programs are for people.
Computers don’t care about how a program looks. Programs are written to communicate with people. Making programs easy to read and understand means that they are more easily changed and reused, and that they more effectively communicate process to other humans.

What if we had written decreaseBlue() and decreaseGreen() so that each asked you to pick a file and created the picture before changing the color. We would be asked for to pick a file twice—once in each function. Because we wrote
these functions to \textit{only} decrease the blue and decrease the green ("one and only one thing") in the implicitly passed \texttt{Picture} object, we can use them in new functions like \texttt{makeSunset()}

There is an issue that the new \texttt{makeSunset()} will take twice as long to finish as the original one, since every pixel gets changed twice. We address that issue in a later chapter on speed and complexity. The important issue is still to write the code readably \textit{first}, and worry about efficiency later. However, this could also be handled by a method that changes each color by some passed in amount. This would be a very general and reusable method.

Now, let’s say that we asked you to pick a picture and created the picture in \texttt{makeSunset()} before calling the other methods. The methods \texttt{reduceBlue()} and \texttt{reduceGreen()} are completely flexible and reusable again. But \texttt{makeSunset()} is now less flexible and reusable. Is that a big deal? No, not if you only care about having the ability to give a sunset look to a single picked picture. But what if you later want to build a movie with a few hundred frames of \texttt{Picture} objects, to each of which you want to add a sunset look? Do you really want to pick out each of those few hundred frames? Or would you rather write a method to go through each of the frames (which we’ll learn how to do in a few chapters) and invoke \texttt{makeSunset()} on each \texttt{Picture} object. That’s why we make methods general and reusable—you never know when you’re going to want to use that method again, in a larger context.

\begin{center}
\begin{tcolorbox}[enhanced,arc=0.5mm,boxrule=0.3mm,before skip=1em,after skip=0.5em]
\textbf{Making it Work Tip: Don’t start by trying to write applications}

There’s a tendency for new programmers to want to write complete applications that a non-technical user can use. You might want to write a \texttt{makeSunset()} application that goes out and fetches a picture for a user and generates a sunset for them. Building good user interfaces that anyone can use is hard work. Start out more slowly. It’s hard enough to make a method just operates on a picture. You can work on user interfaces later.

Even larger methods, like \texttt{makeSunset()}, do "one and only one thing." \texttt{makeSunset()} makes a sunset-looking picture. It does that by decreasing green and decreasing blue. It calls two other methods to do that. What we end up with is a \textit{hierarchy} of goals—the "one and only one thing" that is being done. \texttt{makeSunset()} does its one thing, by asking two other methods to do their one thing. We call this \textit{hierarchical decomposition} (breaking down a problem into smaller parts, and then breaking down those smaller parts until you get something that you can easily program), and it’s very powerful for creating complex programs out of pieces that you understand.
\end{tcolorbox}
\end{center}

### 3.3.5 Variable name scope

Names in methods are \textit{completely} separate from names in the interactions pane and also from names in other methods. We say that they have different \textit{scope}. Scope is the area where a name is known by the computer. Variables declared inside of
a method have method scope and only apply inside that method. That is why we can use the same variable names in several methods. Variables declared inside the Interactions Pane are known inside the Interactions Pane until it is reset. This is why you get **Error: Redefinition of 'picture'** when you declare a variable that is already declared in the Interactions Pane.

The only way to get any data (pictures, sounds, filenames, numbers) from the interactions pane into a function is by passing it in as input to the function. Within the function, you can use any names you want—names that you first define within the method (like `pixel` in the last example) or names that you use to stand for the input data (like `fileName`) only exist while the method is running. When the method is done, those variable names literally do not exist anymore.

This is really an advantage. Earlier, we said that naming is very important to computer scientists: We name everything, from data to methods to classes. But if each name could mean one and only one thing ever, we’d run out of names. In natural language, words mean different things in different contexts (e.g., “What do you mean?” and “You are being mean!”). A method is a different context—names can mean something different than they do outside of that method.

Sometimes, you will compute something inside a method that you want to return to the interactions pane or to a calling method. We’ve already seen methods that output a value, like `FileChooser.pickAFile()` which outputs a filename. If you created a `Picture` object using `new Picture(fileName)` inside a method, you should output it so that it can be used. You can do that by using the `return` keyword as we did for `showNamed(fileName)`.

The name that you give to a method’s input can be thought of as a placeholder. Whenever the placeholder appears, imagine the input data appearing instead. So, in a method like:

```java
/**
 * Method to change the red by an amount
 * @param amount the amount to change the red by
 */
public void changeRed(double amount) {
    Pixel[] pixelArray = this.getPixels();
    Pixel pixel = null;
    int value = 0;
    int i = 0;

    // loop through all the pixels
    while (i < pixelArray.length) {
        // get the current pixel
        pixel = pixelArray[i];

        // get the value
```
Section 3.3  Changing color values  101

```java
value = pixel.getRed();

/* set the red value to the original value * times the passed amount */
pixel.setRed((int) (value * amount));

// increment i
i++;
}
```

When you call (invoke) the method `changeRed` with a specific amount such as `picture.changeRed(0.7)`; which would decrease the red by 30%. In the method `changeRed` the input parameter `amount` is set to 0.7. This is similar to declaring a variable inside the method like this `double amount = 0.7;`. Just like any variable declared in the method the parameter amount is known inside the method. It has method scope.

Call `changeRed` with an amount less than 1 to decrease the amount of red in a picture. Call `changeRed` with an amount greater than one to increase the amount of red in a picture. Remember that the amount of red must be between 0 and 255. If you try to set the amount of red less than 0 it will be set to 0. If you try to set the amount of red greater than 255 it will be set to 255.

We’ve now talked about different ways of writing the same method—some better, some worse. There are others that are pretty much equivalent, and others that are much better. Let’s consider a few more ways that we can write methods.

We can pass in more than input at a time. Consider the following:

Program 14: Change all pixel colors by the passed amounts

```java
/**
 * Method to change the color of each pixel in the picture object by passed in amounts.
 * @param redAmount the amount to change the red value
 * @param greenAmount the amount to change the green value
 * @param blueAmount the amount to change the blue value
 */
public void changeColors(double redAmount,
                          double greenAmount,
                          double blueAmount)
{
    Pixel[] pixelArray = this.getPixels();
    Pixel pixel = null;
    int value = 0;
    int i = 0;

    // loop through all the pixels
    while (i < pixelArray.length)
```
Chapter 3  Modifying Pictures using Loops

{  
    // get the current pixel
    pixel = pixelArray[i];

    // change the red value
    value = pixel.getRed();
    pixel.setRed((int)(redAmount * value));

    // change the green value
    value = pixel.getGreen();
    pixel.setGreen((int)(greenAmount * value));

    // change the blue value
    value = pixel.getBlue();
    pixel.setBlue((int)(blueAmount * value));

    // increment i
    i++;
}

We could use this method as shown below:

> String fileName = "C:/intro-prog-java/mediasources/beach-smaller.jpg";
> Picture picture = new Picture(fileName);
> picture.changeColors(1.0, 0.7, 0.7);
> picture.show();

The above code would have the same result as makeSunset(). It keeps the red values the same and decreases the green and blue values 30%. That’s a pretty useful and powerful function.

Recall seeing in Method 10 this code:

    /**
     * Method to clear the blue from the picture (set the blue to 0 for all pixels)
     */
    public void clearBlue()
    {
        Pixel[] pixelArray = this.getPixels();
        Pixel pixel = null;
        int index = 0;

        // loop through all the pixels
        while (index < pixelArray.length)
        {
            // get the current pixel
            pixel = pixelArray[index];

            // set the blue on the pixel to 0
            pixel.setBlue(0);
Section 3.3 Changing color values

We could also write that same algorithm like this:

```java
/**
 * Method to clear the blue from the picture (set the blue to 0 for all pixels)
 */
public void clearBlue2() {
    Pixel[] pixelArray = this.getPixels();
    int i = 0;

    // loop through all the pixels
    while (i < pixelArray.length) {
        pixelArray[i].setBlue(0);
        i++;
    }
}
```

It’s important to note that this function achieves the exact same thing as the earlier method did. Both set the blue channel of all pixels to zero. An advantage of the second method is that it is shorter and doesn’t require a variable declaration for a pixel. However, it may be harder for someone to understand. A shorter method isn’t necessarily better.

3.3.6 Using a for loop

You may have had the problem that you forgot to declare the index variable before you tried to use it in your `while` loop. You may also have had the problem of forgetting to increment the index variable before the end of the loop body. This happens often enough that another kind of loop is usually used when you want to loop a set number of times. It is called a `for` loop.

A `for` loop executes a command or group of commands in a block. A `for` loop allows for declaration and/or initialization of variables before the loop body is first executed. A `for` loop continues executing the loop body while the continuation test is true. After the end of the body of the loop and before the continuation test one or more variables can be changed.

The syntax for a `for` loop is:

```java
for (initialization area; continuation test; change area) {
    /* commands in body of the loop */
}
```

Let’s talk through the pieces here.

- First comes the required Java keyword `for`.
- Next we have a required opening parenthesis
Next is the initialization area. You can declare and initialize variables here. For example, you can have `int i=0` which declares a variable `i` of the primitive type `int` and initializes it to 0. You can initialize more than one variable here by separating the initializations with commas. You are not required to have any initializations here.

Next comes the required semicolon.

Next is the continuation test. This holds an expression that returns true or false. When this expression is true the loop will continue to be executed. When this test is false the loop will finish and the statement following the body of the loop will be executed.

Next comes the required semicolon.

Next is the change area. Here you usually increment or decrement variables, such as `i++` to increment `i`. The statements in the change area actually take place after each execution of the body of the loop.

Next is the required closing parenthesis.

If you just want to execute one statement (command) in the body of the loop it can just follow on the next line. It is normally indented to show that it is part of the `for` loop. If you want to execute more than one statements in the body of the `for` loop you will need to enclose the statements in a block (a set of open and close curly braces).

![Flowchart of a for loop](image)

**FIGURE 3.20: Flowchart of a for loop**
Compare the flowchart (Figure 3.20) for a for loop with the flowchart for a while loop (Figure 3.14). They look the same because for loops and while loops execute in the same way even though the code looks different. Any code can be written using either. The syntax of the for loop just makes it easier to remember to declare a variable for use in the loop and to change it each time through the loop since all of that is written at the same time that you write the test. To change clearBlue() to use a for loop simply move the declaration and initialization of the index variable i to be done in the initialization area and the increment of i to be done in the change area.

Program 15: Another clear blue method

```java
/**
 * Method to clear the blue from the picture (set the blue to 0 for all pixels)
 */
public void clearBlue3()
{
    Pixel[] pixelArray = this.getPixels();

    // loop through all the pixels
    for (int i = 0; i < pixelArray.length; i++)
        pixelArray[i].setBlue(0);
}
```

3.3.7 Lightening and darkening

To lighten or darken a picture is pretty simple. It’s the same pattern as we saw previously, but instead of changing a color component, you change the overall color. Here’s lightening and then darkening as methods. Figure 3.21 (Figure 4.23) shows the lighter and darker versions of the original picture seen earlier.

Program 16: Lighten the picture

```java
/**
 * Method to lighten the colors in the picture
 */
public void lighten()
{
    Pixel[] pixelArray = this.getPixels();
    Color color = null;
    Pixel pixel = null;

    // loop through all the pixels
    for (int i = 0; i < pixelArray.length; i++)
```
Chapter 3  Modifying Pictures using Loops

```java
{  
    // get the current pixel
    pixel = pixelArray[i];

    // get the current color
    color = pixel.getColor();

    // get a lighter color
    color = color.brighter();

    // set the pixel color to the lighter color
    pixel.setColor(color);
}
```

### Program 17: Darken the picture

```java
/**
 * Method to darken the color in the picture
 */
public void darken()
{
    Pixel[] pixelArray = this.getPixels();
    Color color = null;
    Pixel pixel = null;

    // loop through all the pixels
    for (int i = 0; i < pixelArray.length; i++)
    {
        // get the current pixel
        pixel = pixelArray[i];

        // get the current color
        color = pixel.getColor();

        // get a darker color
        color = color.darker();

        // set the pixel color to the darker color
        pixel.setColor(color);
    }
}
```

### 3.3.8 Creating a negative

Creating a negative image of a picture is much easier than you might think at first. Let’s think it through. What we want is the opposite of each of the current values
Section 3.3 Changing color values

for red, green, and blue. It’s easiest to understand at the extremes. If we have a red component of 0, we want 255 instead. If we have 255, we want the negative to have a zero.

Now let’s consider the middle ground. If the red component is slightly red (say, 50), we want something that is almost completely red—where the “almost” is the same amount of redness in the original picture. We want the maximum red (255), but 50 less than that. We want a red component of $255 - 50 = 205$. In general, the negative should be $255 - \text{original}$. We need to compute the negative of each of the red, green, and blue components, then create a new negative color, and set the pixel to the negative color.

Here’s the program that does it, and you can see even from the grayscale image that it really does work (Figure 3.22 and Figure 4.24).

Program 18: Create the negative of the original picture

```java
/** *
  * Method to negate the picture
  */
public void negate()
{
  Pixel[] pixelArray = this.getPixels();
  Pixel pixel = null;
  int redValue, blueValue, greenValue = 0;

  // loop through all the pixels
  for (int i = 0; i < pixelArray.length; i++)
  {
    // get the current pixel
    pixel = pixelArray[i];

    // get the current red, green, and blue values
    redValue = pixel.getRed();
    greenValue = pixel.getGreen();
```

FIGURE 3.21: Original picture, lightened picture, and darkened picture
3.3.9 Converting to grayscale

Converting to grayscale is a fun program. It’s short, not hard to understand, and yet has such a nice visual effect. It’s a really nice example of what one can do easily yet powerfully by manipulating pixel color values.

Recall that the resultant color is gray whenever the red component, green component, and blue component have the same value. That means that our RGB encoding supports 256 levels of gray from, (0,0,0) (black) to (1,1,1) through (100,100,100) and finally (255,255,255). The tricky part is figuring out what the replicated value should be.

What we want is a sense of the intensity of the color. It turns out that it’s pretty easy to compute: We average the three component colors. Since there are three components, the formula for intensity is:

$\frac{\text{red} + \text{green} + \text{blue}}{3}$

This leads us to the following simple program and Figure 3.23 (and Figure 4.25 on page 156).

Program 19: Convert to grayscale

```java
blueValue = pixel.getBlue();

// set the pixel's color to the new color
pixel.setColor(new Color(255 - redValue,
                         255 - greenValue,
                         255 - blueValue));
```

Figure 3.22: Negative of the image
Section 3.3  Changing color values

![Figure 3.23: Color picture converted to grayscale](image)

```java
public void grayscale()
{
    Pixel[] pixelArray = this.getBytes();
    Pixel pixel = null;
    int intensity = 0;

    // loop through all the pixels
    for (int i = 0; i < pixelArray.length; i++)
    {
        // get the current pixel
        pixel = pixelArray[i];

        // compute the intensity of the pixel (average value)
        intensity = (int) ((pixel.getRed() + pixel.getGreen() +
                            pixel.getBlue()) / 3);

        // set the pixel color to the new color
        pixel.setColor(new Color(intensity, intensity, intensity));
    }
}
```

This is an overly simple notion of grayscale. Below is a program that takes into account how the human eye perceives luminance. Remember that we consider blue to be darker than red, even if there's the same amount of light reflected off. So, we weight blue lower, and red more, when computing the average.

```java
/**
 * Method to change the picture to gray scale with luminance
 */
```
public void grayscaleWithLuminance()
{
    Pixel[] pixelArray = this.getPixels();
    Pixel pixel = null;
    int luminance = 0;
    double redValue = 0;
    double greenValue = 0;
    double blueValue = 0;

    // loop through all the pixels
    for (int i = 0; i < pixelArray.length; i++)
    {
        // get the current pixel
        pixel = pixelArray[i];

        // get the corrected red, green, and blue values
        redValue = pixel.getRed() * 0.299;
        greenValue = pixel.getGreen() * 0.587;
        blueValue = pixel.getBlue() * 0.114;

        // compute the intensity of the pixel (average value)
        luminance = (int) (redValue + greenValue + blueValue);

        // set the pixel color to the new color
        pixel.setColor(new Color(luminance, luminance, luminance));
    }
}

OBJECTS AND METHODS SUMMARY

In this chapter, we talk about several kinds of encodings of data (or objects).

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Color</td>
<td>An object that holds red, green, and blue values, each between 0 and 255.</td>
</tr>
<tr>
<td>Picture</td>
<td>Pictures are encodings of images, typically coming from a JPEG file.</td>
</tr>
<tr>
<td>Pixel</td>
<td>A pixel is a dot in a Picture object. It has a color (red, green, and blue)</td>
</tr>
<tr>
<td></td>
<td>and an (x, y) position associated with it. It remembers its own Picture object so that a change to the pixel changes the real dot in the picture.</td>
</tr>
</tbody>
</table>

Picture methods
Section 3.3 Changing color values

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getHeight()</td>
<td>This method returns the height of the Picture object in pixels.</td>
</tr>
<tr>
<td>getPixel(int x, int y)</td>
<td>This method takes an x position and a y position (two numbers), and returns the Pixel object at that location in the Picture object it is invoked on.</td>
</tr>
<tr>
<td>getPixels()</td>
<td>Returns a one-dimensional array of Pixel objects in the Picture object it is invoked on.</td>
</tr>
<tr>
<td>getWidth()</td>
<td>This method the width of the Picture object in pixels.</td>
</tr>
<tr>
<td>writePictureTo(String fileName)</td>
<td>This method takes a file name (a string) as input, then writes the Picture object to the file as a JPEG. (Be sure to end the filename in “.jpg” for the operating system to understand it well.)</td>
</tr>
</tbody>
</table>

**Pixel methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getColor()</td>
<td>Returns the Color object for the Pixel object.</td>
</tr>
<tr>
<td>getRed(), getGreen(), getBlue()</td>
<td>Each method returns the value (between 0 and 255) of the amount of redness, greenness, and blueness (respectively) in the Pixel object.</td>
</tr>
<tr>
<td>getX(), getY()</td>
<td>This method returns the x or y (respectively) position of where that Pixel object is in the picture.</td>
</tr>
<tr>
<td>setColor(Color color)</td>
<td>This method takes a Color object and sets the color for the Pixel object.</td>
</tr>
<tr>
<td>setRed(int value), setGreen(int value), setBlue(int value)</td>
<td>Each method takes a value (between 0 and 255) and sets the redness, greenness, or blueness (respectively) of the Pixel object to the given value.</td>
</tr>
</tbody>
</table>

**Color methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>new Color(int red,int green,int blue)</td>
<td>Takes three inputs: the red, green, and blue values (in that order), then creates and returns a Color object.</td>
</tr>
<tr>
<td>darker(),brighter()</td>
<td>The methods return a slightly darker or lighter (respectively) version of the Color object.</td>
</tr>
</tbody>
</table>

**ColorChooser methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ColorChooser.pickAColor()</td>
<td>Takes no input, but puts up a color picker. Find the color you want, and the function will return the Color object that you picked.</td>
</tr>
</tbody>
</table>
Chapter 3 Modifying Pictures using Loops

There are a bunch of constants that are useful in this chapter. These are variables with pre-defined values. These values are colors: Color.black, Color.white, Color.blue, Color.red, Color.green, Color.gray, Color.darkGray, Color.lightGray, Color.yellow, Color.orange, Color.pink, Color.magenta, Color.cyan. Notice that these are not method calls but are class variables (fields) so they can be accessed using ClassName.fieldName.

PROBLEMS

3.1. Some picture concepts questions:

- Why don’t we see red, green, and blue spots at each position in our picture?
- What is luminance?
- Why is the maximum value of any color channel 255?
- The color encoding we’re using is “RGB”. What does that mean, in terms of the amount of memory required to represent color? Is there a limit to the number of colors that we can represent? Are there enough colors representable in RGB?

3.2. Program 8 (page 81) is obviously too much color reduction. Write a version that only reduces the red by 10%, then one by 20%. Which seems to be more useful? Note that you can always repeatedly reduce the redness in a picture, but you don’t want to have to do it too many times, either.

3.3. Write the blue and green versions of Program 8 (page 81).

3.4. Each of the below is equivalent to Program 9 (page 92). Test them and convince yourself that they are equivalent. Which do you prefer and why?

```java
/**
 * Method to increase the amount of red by 1.3
 */
public void increaseRed2()
{
    Pixel[] pixelArray = this.getPixels();
    int value = 0;

    // loop through all the pixels
    for (int i = 0; i < pixelArray.length; i++)
    {
        // set the red value to 1.3 times what it was
        value = pixelArray[i].getRed();
        pixelArray[i].setRed((int) (value * 1.3));
    }
}

/**
 * Method to increase the amount of red by 1.3
 */
public void increaseRed3()
{
    Pixel[] pixelArray = this.getPixels();
    Pixel pixel = null;
    int red = 0;
```
Section 3.3 Changing color values

```java
int green = 0;
int blue = 0;
int newRed = 0;

// loop through all the pixels
for (int i = 0; i < pixelArray.length; i++)
{
    // get the current pixel
    pixel = pixelArray[i];

    // get the color values
    red = pixel.getRed();
green = pixel.getGreen();
blue = pixel.getBlue();

    // calculate the new red value
    newRed = (int) (red * 1.3);

    // set the pixel color to the new color
    pixel.setColor(new Color(newRed, green, blue));
}
```

3.5. Change any of the methods that used a `while` loop to use a `for` loop. Compile and run the changed method and make sure it still works.

3.6. Change a variable name in any of the given methods. Make sure you change all instances of the original name to the new name. Compile and run the changed method and make sure it still works.

3.7. Write new methods like Program 10 (page 94) to clear red and green. For each of these, which would be the most useful in actual practice? How about combinations of these?

3.8. Write a method to keep just the blue color. This means to set all the green and red values to zero.

3.9. Write a new method to maximize blue (i.e., setting it to 255) instead of clearing it use Program 10 (page 94) as a starting point. Is this useful? Would the red or green versions be useful?

3.10. There is more than one way to compute the right grayscale value for a color value. The simple method that we use in Program 19 (page 108) may not be what your grayscale printer uses when printing a color picture. Compare the color (relatively unconverted by the printer) grayscale image using our simple algorithm in Figure 4.25 with the original color picture that the printer has converted to grayscale (left of Figure 3.15). How do the two pictures differ?

3.11. Think about how the grayscale algorithm works. Basically, if you know the luminance of anything visual (e.g., a small image, a letter), you can replace a pixel with that visual element in a similar way to create a collage image. Try implementing that. You’ll need 256 visual elements of increasing lightness, all of the same size. You’ll create a collage by replacing each pixel in the original image with one of these visual elements.
TO DIG DEEPER

A wonderful new book on how vision works, and how artists have learned to manipulate it, is *Vision and art: The biology of Seeing* by Margaret Livingstone [18].
CHAPTER 4

Modifying Pixels in a Matrix

4.1 COPYING PIXELS
4.2 COPYING AND TRANSFORMING PICTURES
4.3 COLOR FIGURES

Chapter Learning Objectives

The media learning goals for this chapter are:

- To mirror pictures, horizontally or vertically.
- To compose pictures into one another and create collages.
- To blend two pictures.
- To rotate pictures.
- To scale pictures smaller and larger.

The computer science goals for this chapter are:

- To use nested loops for processing elements of a matrix (a two-dimensional array).
- To initialize and change multiple variables in a for loop.
- To develop some debugging strategies, specifically, using System.out.println() statements to explore executing code.
- To break long methods into smaller pieces.
- To introduce method overloading.

4.1 COPYING PIXELS

We can only get so far in our image processing with getPixels() before we need to know where a pixel is. For example, if we want to copy just part of a picture to another picture we will need to know the x and y values to start with and end at.
4.1.1 Looping across the pixels with a nested loop

We can’t use a single for loop if we want to keep track of the x and y values for a pixel. We have to use two for loops—one to move horizontally across the columns, and the other to move vertically to get every pixel. The function getPixels() did this inside itself, to make it easier to write simple picture manipulations. But if you want to access each individual pixel, you’ll need to use two loops, one for each dimension of the picture. The inner loop will be nested inside the outer loop, literally, inside its block.

Your loops will look something like this:

```java
// loop through the columns (x direction)
for (int x = 0; x < getWidth(); x++) {
    // loop through the rows (y direction)
    for (int y = 0; y < getHeight(); y++) {
        // get the current pixel at this x and y position
        pixel = getPixel(x, y);

        // do something to the color
        pixel.setColor(aColor);
    }
}
```

This will process all the y values from top to bottom in the first column and then all the y values in the next column and so on until all the pixels are processed.

You could also process all the x values in the top row and then all the x values in the next row and so on using this:

```java
// loop through the rows (y direction)
for (int y = 0; y < getHeight(); y++) {
    // loop through the columns (x direction)
    for (int x = 0; x < getHeight(); x++) {
        // get the current pixel at this x and y position
        pixel = getPixel(x, y);

        // do something to the color
        pixel.setColor(aColor);
    }
}
```

Does it matter which way you process the pixels? Not if all you are trying to do is process all the pixels. Both of these loops will process all the pixels in a picture.
For example, here’s Program 16 (page 105), but using explicit pixel references.

Program 21: Lighten the picture using nested loops

```java
/**
 * Method to lighten the colors in the picture
 */
public void lighten2 ()
{
    Color color = null;
    Pixel pixel = null;

    // loop through the columns (x direction)
    for (int x = 0; x < getWidth(); x++)
    {
        // loop through the rows (y direction)
        for (int y = 0; y < getHeight(); y++)
        {
            // get pixel at the x and y location
            pixel = getPixel(x,y);

            // get the current color
            color = pixel.getColor();

            // get a lighter color
            color = color.brighter();

            // set the pixel color to the lighter color
            pixel.setColor(color);
        }
    }
}
```

Let’s walk through (trace) how it would work. Imagine that we just executed `picture.lighten2()`.

1. The code `picture.lighten2()` executes the object method in the `Picture` class `public void lighten2()`. The method is implicitly passed the current picture object (you can refer to the current picture object using the keyword `this`).

2. The code `Color color = null;` and `Pixel pixel = null;` declares the variables `color` (an object of the `Color` class) and `pixel` (an object of the `Pixel` class). Both of these are initialized to `null` (not referring to any object yet). These variables will be needed when we are looping through the pixels. We could declare these in the for loop but then they would be redeclared each time through the loop. It is better to declare them once before the loop and change them each time through the loop.

3. The code `for (int x = 0; x < getWidth(); x++)` declares a variable `x` of
Chapter 4 Modifying Pixels in a Matrix

type int which will be initialized to 0 and then a check will be made to see if \( x \) is less than the width of the current Picture object. If \( x \) is less than the width, then the body of this for loop will be executed. After the body of the loop has been executed one time the value in \( x \) will be incremented and the continuation condition will be tested again.

4. The code `for (int y = 0; y < getHeight(); y++)` declares a variable \( y \) of type int which will be initialized to 0. The test checks that \( y \) is less than the height of the current Picture object. If \( y \) is less than the height then the body of this for loop will be executed. After the body has executed the value in \( y \) will be incremented and the continuation condition will be tested again.

5. The code `pixel = getPixel(x,y);` sets the variable `pixel` to refer to the Pixel object at the given \( x \) and \( y \) location in the picture.

6. The code `color = pixel.getColor();` sets the variable `color` to refer to the Color object at the current pixel.

7. Next comes `color = color.brighter();`. This creates a new lighter (brighter) Color object based on the original Color object and sets the variable `color` to refer to that new Color object.

8. The code `pixel.setColor(color);` sets the current pixel’s color to be the lighter color.

9. Each time we reach the end of the inner for loop the \( y \) value will be incremented by 1 and then the value of \( y \) will be compared to the height of the picture. If the value of \( y \) is less than the height, the statements in the body of the loop will be executed again. If the value of \( y \) is equal or greater than the height, execution will jump to the next statement (the outer loop).

10. Each time we reach the end of the outer for loop the \( x \) value will be incremented by 1 and then the value of \( x \) will be compared to the width of the picture. If the \( x \) value is less than the width of the picture, the commands in the loop body will be executed. If the value of \( x \) is equal or greater than the width of the picture, execution will continue at the statement following the body of the loop.

4.1.2 Mirroring a picture

Let’s start out with an interesting effect that is only occasionally useful, but it is fun. Let’s mirror a picture along its vertical axis. In other words, imagine that you have a mirror, and you place it on a picture so that the left side of the picture shows up in the mirror. That’s the effect that we’re going to implement. We’ll do it in a couple of different ways.

First, let’s think through what we’re going to do. We’ll pick a horizontal mirrorPoint—halfway across the picture, `(int) (picture.getWidth()/2)`. (We want this to be an integer, a whole number, so we’ll cast it using `(int)`.) We’ll have the \( x \) value increment from 1 to the mirrorPoint. At each value of \( x \), we want to copy the color at the pixel \( x \) pixels to the left of the mirrorPoint to the pixel \( x \) pixels to the right of the mirrorPoint. The left would be `mirrorPoint-x` and the right would be `mirrorPoint+x`. Take a look at Figure 4.1 to convince yourself that we’ll actually reach every pixel using this scheme. Here’s the actual program.
Section 4.1  Copying Pixels  119

FIGURE 4.1: Once we pick a mirror point, we can just walk $x$ halfway and subtract/add to the mirror point.

Program 22: Mirror pixels in a picture along a vertical line

```java
/** *
 * Method to mirror around a vertical line in the middle
 * of the picture based on the width
 */
public void mirrorVertical()
{
    int mirrorPoint = (int) (getWidth() / 2);
    Pixel leftPixel = null;
    Pixel rightPixel = null;

    // loop through the rows
    for (int y = 0; y < getHeight(); y++)
    {
        // loop from 1 to just before the mirror point
        for (int x = 1; x < mirrorPoint; x++)
        {
            leftPixel = getPixel((mirrorPoint - x), y);
            rightPixel = getPixel((mirrorPoint + x), y);
            rightPixel.setColor(leftPixel.getColor());
        }
    }
}
```

We’d use it like this, and the result appears in Figure 4.2.

```java
> String fileName = "C:/intro-prog-java/mediasources/caterpillar.jpg";
> System.out.println(fileName);
> Picture picture = new Picture(fileName);
> picture.show();
> picture.mirrorVertical();
> picture.repaint();
```

Another way to code this would be to copy the colors for the pixels starting
with the left-most x (x=0) into the right-most pixel (width - 1). To do this have x range from 0 to less than the mirrorPoint and copy it to (width - 1 - x).

Can we mirror horizontally? Sure!

Program 23: Mirror pixels horizontally, top-to-bottom

```java
/**
 * Method to mirror around a horizontal line in the middle based on the height. It copies the top mirrored to the bottom
 */
public void mirrorHorizontal()
{
    int mirrorPoint = (int) (getHeight() / 2);
    Pixel topPixel = null;
    Pixel bottomPixel = null;

    // loop through the columns
    for (int x=0; x < getWidth(); x++)
    {
        // loop from 1 to just before the mirror point
        for (int y=1; y < mirrorPoint; y++)
        {
            topPixel = getPixel(x, (mirrorPoint - y));
            bottomPixel = getPixel(x, (mirrorPoint + y));
            bottomPixel.setColor(topPixel.getColor());
        }
    }
}
```

We’d use it like this, and the result appears in Figure 4.3.

```java
> String fileName = "C:/intro-prog-java/mediasources/redMotorcycle.jpg";
> System.out.println(fileName);
C:/intro-prog-java/mediasources/redMotorcycle.jpg
> Picture picture = new Picture(fileName);
> picture.show();
> picture.mirrorHorizontal();
```
> picture.repaint();

Now this last method copies from the top of the picture onto the bottom (see Figure 4.3). You can see that we're getting the color from topPixel which is from mirrorPoint - y that will always be above mirrorPoint since smaller values of y are nearer the top of the picture. To copy from the bottom up, simply change the color at the top pixel to the color of the bottom pixel. (Figure 4.3).

**Program 24: Mirror pixels horizontally, bottom-to-top**

```java
/**
 * Method to mirror around a horizontal line in the middle based on the height of the picture. It copies the bottom to the top.
 */
public void mirrorHorizontalBottomToTop()
{
    int mirrorPoint = (int) (getHeight() / 2);
    Pixel topPixel = null;
    Pixel bottomPixel = null;

    // loop through the columns
    for (int x=0; x < getWidth(); x++)
    {
        // loop from 1 to just before the mirror point
        for (int y=1; y < mirrorPoint; y++)
        {
            topPixel = getPixel(x, (mirrorPoint - y));
            bottomPixel = getPixel(x, (mirrorPoint + y));
            topPixel.setColor(bottomPixel.getColor());
        }
    }
}
```

We'd use it like this, and the result appears in Figure 4.3.

> String fileName = "C:/intro-prog-java/mediasources/redMotorcycle.jpg";
> Picture picture = new Picture(fileName);
> picture.show();
> picture.mirrorHorizontalBottomToTop();
> picture.repaint();

Mirroring usefully.

While mirroring is probably mostly used for interesting effects, occasionally it has some more serious (but still fun!) purposes. Mark took a picture of the Temple of Hephaistos which is in the ancient agora in Athens, Greece, when traveling to a conference (Figure 4.4). By sheer luck, Mark got the pediment dead horizontal.
Chapter 4  Modifying Pixels in a Matrix

FIGURE 4.3: A motorcycle mirrored horizontally, top to bottom (left) and bottom to top (right)

The Temple of Hephaistos had its pediment damaged. Mark wondered if he could “fix” it by mirroring the good part onto the broken part.

FIGURE 4.4: Temple of Hephaistos from the ancient agora in Athens, Greece

Mark used the picture explorer to figure out the range of values where he would need to do the mirroring and the point where he should mirror (Figure 4.5). The function he wrote to do the repair is below, and the final picture is in (Figure 4.6)—it worked pretty well! Of course, it is possible to tell that it was digitally manipulated. For example, if you check the shadows, you can see that the sun must have been on the left and the right at the same time.

Program 25: Mirror the Temple

```java
/**
   * Method to mirror the piedmont of the temple
   * @return the corrected picture
   */
public static Picture mirrorTemple()
{
    Picture picture =
        new Picture(FileChooser.getMediaPath("temple.jpg"));
```
Section 4.1 Copying Pixels

FIGURE 4.5: Coordinates where we need to do the mirroring

```java
int mirrorPoint = 276;
int lengthToCopy = mirrorPoint - 13;
Pixel leftPixel = null;
Pixel rightPixel = null;

// loop through the columns
for (int x = 1; x < lengthToCopy; x++)
{
    // loop through the rows
    for (int y = 27; y < 97; y++)
    {
        leftPixel = picture.getPixel(mirrorPoint - x, y);
        rightPixel = picture.getPixel(mirrorPoint + x, y);
        rightPixel.setColor(leftPixel.getColor());
    }
}

// show the picture
picture.show();
return picture;
```

This method is a class method (contains the keyword static in the method declaration. It is a class method because it creates the picture inside of the method. Notice that instead of getPixel, which is really this.getPixel, it uses picture.getPixel to refer to the created Picture object. Class methods are not implicitly passed a Picture object and thus can not use the keyword this. To invoke this method use the following in the interactions pane:

> Picture.mirrorTemple();

In this program, we’re using FileChooser.getMediaPath(fileName). The
function `FileChooser.getMediaPath(fileName)` is a shortcut. If you keep your media in one place, and you’d like to refer to it just by its base name, you can use `FileChooser.getMediaPath(fileName)`, which actually just generates a complete path for you by adding the passed name to the media directory. The default media directory is “c:/intro-prog-java/mediasources”. If you wish to use a different media directory you should use `FileChooser.setMediaPath(directory)` first! `FileChooser.setMediaPath(directory)` lets you specify the place (directory) where you store your media.

The temple example is a good one to ask ourselves about. If you really understand, you can answer questions like “What’s the first pixel to be mirrored in this function?” and “How many pixels get copied anyway?” You should be able to figure these out by thinking through the program—pretend you’re the computer and execute the program in your mind.

If that’s too hard, you can insert `System.out.println()` statements, like this:

```java
/**
 * Method to mirror the piedmont of the temple
 * @return the corrected picture
 */
public static Picture mirrorTemple()
{
    Picture picture = new Picture(getMediaPath("temple.jpg"));
    int mirrorPoint = 276;
    int lengthToCopy = mirrorPoint - 13;
    Pixel leftPixel = null;
    Pixel rightPixel = null;

    // loop through the columns
    for (int x = 1; x < lengthToCopy; x++)
```
{  
  // loop through the rows  
  for (int y = 27; y < 97; y++)  
  {  
    System.out.print("Copying color from " +  
            (mirrorPoint - x) + ", " + y);  
    System.out.println(" to " + (mirrorPoint + x) +  
            ", " + y);  
    leftPixel = picture.getPixel(mirrorPoint - x, y);  
    rightPixel = picture.getPixel(mirrorPoint + x, y);  
    rightPixel.setColor(leftPixel.getColor());  
  }  
}  

// show the picture  
picture.show();  
return picture;  

When we run this version, it takes a long time to finish. Hit RESET after a little bit since we only really care about the first few pixels. Here’s what I got:

> Picture.mirrorTemple();  
Copying color from 275,27 to 277,27  
Copying color from 275,28 to 277,28  
Copying color from 275,29 to 277,29  
Copying color from 275,30 to 277,30  

It copies from just to the left of the mirror point (276), since x is 1 at first, and we copy from mirrorPoint-x to mirrorPoint+x. Thus, we copy down the column before the mirror point to the column of pixels to the right of the mirror point. Then we move back one column to the left, and copy one column further to the right.

How many pixels did we process? We can have the computer figure that one out, too.

/**  
 * Method to mirror the piedmont of the temple  
 * @return the corrected picture  
 */  
public static Picture mirrorTemple()  
{  
P...
4.2 COPYING AND TRANSFORMING PICTURES

We can create wholly new pictures when we copy pixels across pictures. We're going to end up keeping track of a source picture that we take pixels from and a target picture that we're going to set pixels in. Actually, we don't copy the pixels— we simply make the pixels in the target the same color as the pixels in the source. Copying pixels requires us keep track of multiple index variables: The \((x, y)\) positions in the source and the \((x, y)\) in the target.

What's exciting about copying pixels is that making some small changes in how we deal with the index variables leads to not only copying the image but transforming it. In this section, we're going to talk about copying, cropping, rotating, and scaling pictures.

We're going to use `FileChooser.getMediaPath(fileName)` to make our coding of methods with several files easier. We've seen it before. It's particularly helpful when you want to deal with several pieces of media in the same directory but don't want to spell out the whole directory name. You just have to remember to use `FileChooser.setMediaPath(directory)` first! All that this method does is return the media directory followed by the input filename. To set the media directory use: `FileChooser.setMediaPath(directory)`.

```java
> FileChooser.setMediaPath("C:/intro-prog-java/mediasources/");
> FileChooser.getMediaPath("temple.jpg")
```

This one comes back with **We copied 18340 pixels**. Where did that number come from? You can calculate how many times you execute the commands in a for loop with end - start + 1. We copy 70 rows of pixels \((y)\) goes from 27 to 96 (because of the < 97) which is 96 - 27 + 1. We copy 262 columns of pixels \((x)\) goes from 1 to < 263 \((276 - 13)\) which is 262 - 1 + 1 = 262). 70 * 262 is 18,340.
Section 4.2  Copying and transforming pictures

"C:/intro-prog-java/mediasources/temple.jpg"
> Picture temple = new Picture(FileChooser.getMediaPath("temple.jpg"));

Our target will be the paper-sized JPEG file in the mediasources directory, which is 7x9.5 inches, which will fit on a 9x11.5 inch lettersize piece of paper with one inch margins.

> String paperFile = FileChooser.getMediaPath("7inx95in.jpg");
> Picture paperPicture = new Picture(paperFile);
> paperPicture.show();
> System.out.println(paperPicture.getWidth()); 504
> System.out.println(paperPicture.getHeight()); 684

4.2.1 Copying

To copy a picture from one file to another, we simply make sure that we increment sourceX and targetX variables (the source and target index variables for the X axis) together, and the sourceY and targetY variables together. We can initialize more than one variable in the initialization area of a for loop and change more than one variable in the change area.

Here’s a program for copying a picture of Katie to the canvas.

Program 26: Copying a picture to a canvas

```java
/**
 * Method to copy the picture of Katie to the canvas
 * @return the canvas after the picture of Katie has been copied
 */
public static Picture copyKatie()
{
    String sourceFile =
        FileChooser.getMediaPath("KatieFancy.jpg");
    Picture sourcePicture = new Picture(sourceFile);
    String targetFile =
        FileChooser.getMediaPath("7inx95in.jpg");
    Picture targetPicture = new Picture(targetFile);
    Pixel sourcePixel = null;
    Pixel targetPixel = null;

    // loop through the columns
    for (int sourceX = 0, targetX = 0;
         sourceX < sourcePicture.getWidth();
         sourceX++, targetX++)
    {
        // loop through the rows
```
for (int sourceY = 0, targetY = 0;
sourceY < sourcePicture.getHeight();
sourceY++, targetY++)
{
    // set the target pixel color to the source pixel color
    sourcePixel = sourcePicture.getPixel(sourceX, sourceY);
    targetPixel = targetPicture.getPixel(targetX, targetY);
    targetPixel.setColor(sourcePixel.getColor());
}

// show the source and target pictures
sourcePicture.show();
targetPicture.show();
return targetPicture;

FIGURE 4.7: Copying a picture to a canvas

This program copies a picture of Katie to the canvas (blank picture) (Figure 4.7). Here’s how it works:

• The first few lines are just setting up the source (sourcePicture) and target (targetPicture) pictures.

• Next comes the loop for managing the x index variables, sourceX for the source picture and targetX for the target picture. The for loop declares both variables and initializes them to 0. You can have more than one variable declared and initialized in the initialization area of a for loop, just separate
Section 4.2  Copying and transforming pictures  129

them with commas. Next the continuation test checks if the sourceX is less than the width of the source picture. Finally in the change area we increment both the sourceX and targetX variables each time after the statements in the body of the loop have been executed. You can change more than one variable in the change area as long as you separate the changes with commas. The for loop for looping through the columns is:

```
for (int sourceX = 0, targetX = 0;
     sourceX < sourcePicture.getWidth();
     sourceX++, targetX++)
```

- Inside the loop for the X variables is the loop for the Y variables. It has a very similar structure, since it’s goal is to keep targetY and sourceY in synch in exactly the same way.

```
for (int sourceY = 0, targetY = 0;
     sourceY < sourcePicture.getHeight();
     sourceY++, targetY++)
```

It’s inside the Y loop that we actually get the color from the source pixel and set the corresponding pixel in the target to the same color.

FIGURE 4.8: Copying a picture midway into a canvas

Of course, we don’t have to copy from (0,0) in the source to (0,0) in the target. We can easily copy somewhere else in the canvas, too. All we have to do is to change where the target X and Y coordinates start. The rest stays exactly the
same (Figure 4.8).

**Program 27: Copy elsewhere into the canvas**

```java
/**
 * Method to copy the picture of Katie to 100, 100 in the
 * canvas
 * @return the picture of Katie copied to 100,100
 */
public static Picture copyKatieMidway()
{
    String sourceFile =
       FileChooser.getMediaPath("KatieFancy.jpg");
    Picture sourcePicture = new Picture(sourceFile);
    String targetFile =
       FileChooser.getMediaPath("7inx9in.jpg");
    Picture targetPicture = new Picture(targetFile);
    Pixel sourcePixel = null;
    Pixel targetPixel = null;

    // loop through the columns
    for (int sourceX = 0, targetX=100;
        sourceX < sourcePicture.getWidth();
        sourceX++, targetX++)
    {
        // loop through the rows
        for (int sourceY = 0, targetY =100;
            sourceY < sourcePicture.getHeight();
            sourceY++, targetY++)
        {
            // set the target pixel color to the source pixel color
            sourcePixel = sourcePicture.getPixel(sourceX,sourceY);
            targetPixel = targetPicture.getPixel(targetX,targetY);
            targetPixel.setColor(sourcePixel.getColor());
        }
    }

    // show the source and target pictures
    sourcePicture.show();
    targetPicture.show();

    return targetPicture;
}
```

Similarly, we don’t have to copy a whole picture. **Cropping** is taking only part of a picture out of the whole picture. Digitally, that’s just a matter of changing your start and end coordinates. To grab just Katie’s face out of the picture, we only have to figure out the upper left corner of a rectangle enclosing her face and use that as the starting values for `sourceX` and `sourceY`. We also need to determine the
bottom right corner of the rectangle enclosing her face and use that as the stopping x and y values. (Figure 4.9). We can use the picture explorer to determine these values. The upper left corner of the rectangle enclosing the face is at (70, 3) and the bottom right corner is at (135, 80).

![Figure 4.9: Copying part of a picture onto a canvas](image)

**Program 28: Cropping a picture onto a canvas**

```java
/**
 * Method to copy just Katie’s face to the canvas
 * @return the canvas after the copying the face
 */
public static Picture copyKatiesFace()
{
    String sourceFile =
       FileChooser.getMediaPath("KatieFancy.jpg");
    Picture sourcePicture = new Picture(sourceFile);
    String targetFile =
       FileChooser.getMediaPath("7inx95in.jpg");
    Picture targetPicture = new Picture(targetFile);
    Pixel sourcePixel = null;
    Pixel targetPixel = null;

    // loop through the columns
    for (int sourceX = 70, targetX = 100;
        sourceX < 135; sourceX++, targetX++)
    {
```
How does that work?
Let’s look at a small example to see what’s going on in the copying program. We start out with a source and a target, and copy from x=0, y=0 to x=3, y=1.

We then increment both the sourceY and targetY, and copy again.

We continue down the column, incrementing both Y index variables.

When done with that column, we increment the X index variables and move on to the next column, until we copy every pixel.
4.2.2 Creating a Collage

In the mediasources folder are a couple images of flowers (Figure 4.10), each 100 pixels wide. Let’s make a collage of them, by combining several of our effects to create different flowers. We’ll copy them all into the blank image 640x480.jpg. All we really have to do is to copy the pixel colors to the right places.

![Figure 4.10: Flowers in the mediasources folder](image)

**Program 29: Creating a collage**

```java
/**
 * Method to create a collage from the flower pictures. All
 * the flower pictures will be lined up near the bottom of
 * the canvas (5 pixels from the bottom)
 * @return the collage as a picture object
 */
public static Picture createCollage()
{
    // create the three pictures
    Picture flower1Picture =
        new Picture(FileChooser.getMediaPath("flower1.jpg"));
    Picture flower2Picture =
        new Picture(FileChooser.getMediaPath("flower2.jpg"));
    Picture canvasPicture =
        new Picture(FileChooser.getMediaPath("640x480.jpg"));

    // declare the source and target pixel variables
    Pixel sourcePixel = null;
    Pixel targetPixel = null;
```
// save the heights of the two pictures
int flower1Height = flower1Picture.getHeight();
int flower2Height = flower2Picture.getHeight();

// print out the picture information
System.out.println(flower1Picture);
System.out.println(flower2Picture);
System.out.println(canvasPicture);

/* copy the first flower picture to 5 pixels from the bottom left corner of the canvas */
for (int sourceX = 0, targetX = 0;
     sourceX < flower1Picture.getWidth();
     sourceX++, targetX++)
{
    for (int sourceY = 0,
         targetY = canvasPicture.getHeight() -
                    flower1Height - 5;
         sourceY < flower1Picture.getHeight();
         sourceY++, targetY++)
    {
       sourcePixel = flower1Picture.getPixel(sourceX, sourceY);
       targetPixel = canvasPicture.getPixel(targetX, targetY);
       targetPixel.setColor(sourcePixel.getColor());
    }
}

/* copy the flower2 picture starting with x = 100 in the canvas */
for (int sourceX = 0, targetX = 100;
     sourceX < flower2Picture.getWidth();
     sourceX++, targetX++)
{
    for (int sourceY = 0,
         targetY = canvasPicture.getHeight() -
                    flower2Height - 5;
         sourceY < flower2Picture.getHeight();
         sourceY++, targetY++)
    {
       sourcePixel = flower2Picture.getPixel(sourceX, sourceY);
       targetPixel = canvasPicture.getPixel(targetX, targetY);
       targetPixel.setColor(sourcePixel.getColor());
    }
}

// copy the flower1 negated to x = 200 in the canvas
flower1Picture.negate();
for (int sourceX = 0, targetX = 200;
Section 4.2 Copying and transforming pictures

```java
sourceX < flower1Picture.getWidth();
sourceX++, targetX++)
{
    for (int sourceY = 0,
         targetY = canvasPicture.getHeight() -
                    flower1Height - 5;
         sourceY < flower1Picture.getHeight();
         sourceY++, targetY++)
    {
        sourcePixel = flower1Picture.getPixel(sourceX, sourceY);
        targetPixel = canvasPicture.getPixel(targetX, targetY);
        targetPixel.setColor(sourcePixel.getColor());
    }
}

/* clear the blue in flower 2 picture and add at x=300 in
 * the canvas */
flower2Picture.clearBlue();
for (int sourceX = 0, targetX = 300;
    sourceX < flower2Picture.getWidth();
    sourceX++, targetX++)
{
    for (int sourceY = 0,
        targetY = canvasPicture.getHeight() -
                  flower2Height - 5;
        sourceY < flower2Picture.getHeight();
        sourceY++, targetY++)
    {
        sourcePixel = flower2Picture.getPixel(sourceX, sourceY);
        targetPixel = canvasPicture.getPixel(targetX, targetY);
        targetPixel.setColor(sourcePixel.getColor());
    }
}

// copy the negated flower 1 to x=400
for (int sourceX = 0, targetX = 400;
    sourceX < flower1Picture.getWidth();
    sourceX++, targetX++)
{
    for (int sourceY = 0,
        targetY = canvasPicture.getHeight() -
                  flower1Height - 5;
        sourceY < flower1Picture.getHeight();
        sourceY++, targetY++)
    {
        sourcePixel = flower1Picture.getPixel(sourceX, sourceY);
        targetPixel = canvasPicture.getPixel(targetX, targetY);
        targetPixel.setColor(sourcePixel.getColor());
    }
}
```
Here's how we run the collage (Figure 4.11):

```
> Picture flowerCollage = Picture.createCollage();
Picture, filename C:/intro-prog-java/mediasources/flower1.jpg
  height 138 width 100

Picture, filename C:/intro-prog-java/mediasources/flower2.jpg
  height 227 width 100

Picture, filename C:/intro-prog-java/mediasources/640x480.jpg
  height 480 width 640
```

![Collage of flowers](image)

FIGURE 4.11: Collage of flowers

This method is long and repetitive which makes it hard to read. One of the ways to improve it is to pull out pieces of code that are basically the same and make new methods. Each time we add a new picture to our canvas the only things changing are the picture to be added and the `targetX`. The `targetY` is always calculated the same way as the height of the canvas minus the height of the picture being copied minus 5.

Since both the target and source `Picture` objects exist when this method is called it can be an object method (no `static` keyword in the method declaration). Which `picture` object should it be called on? Since it is changing the target picture
it makes sense to call it on the target picture and just pass in the source picture and starting x.

```java
/**
 * Method that will copy all of the passed source picture into
 * the current picture object starting with the left corner
 * given by xStart
 * @param sourcePicture the picture object to copy
 * @param xStart the x position to start the copy into
 */
public void copyPictureTo(Picture sourcePicture, int xStart)
{
    Pixel sourcePixel = null;
    Pixel targetPixel = null;

    // loop through the columns
    for (int sourceX = 0, targetX = xStart;
         sourceX < sourcePicture.getWidth();
         sourceX++, targetX++)
    {
        // loop through the rows
        for (int sourceY = 0,
             targetY = this.getHeight() -
             sourcePicture.getHeight() - 5;
             sourceY < sourcePicture.getHeight();
             sourceY++, targetY++)
        {
            sourcePixel = sourcePicture.getPixel(sourceX, sourceY);
            targetPixel = this.getPixel(targetX, targetY);
            targetPixel.setColor(sourcePixel.getColor());
        }
    }
}

/**
 * Method to create a collage of flowers
 * @return the flower collage as a picture object
 */
public static Picture createCollageBetter()
{
    // create the three pictures
    Picture flower1Picture =
        new Picture(FileChooser.getMediaPath("flower1.jpg"));
    Picture flower2Picture =
        new Picture(FileChooser.getMediaPath("flower2.jpg"));
    Picture canvasPicture =
        new Picture(FileChooser.getMediaPath("640x480.jpg"));

    // print out the picture information
System.out.println(flower1Picture);
System.out.println(flower2Picture);
System.out.println(canvasPicture);

// copy the first flower picture to near the
// bottom left corner of the canvas
canvasPicture.copyPictureTo(flower1Picture,0);

/* copy the flower2 picture starting with x = 100 in
 * the canvas */
canvasPicture.copyPictureTo(flower2Picture,100);

// copy the flower1 negated to x = 200 in the canvas
flower1Picture.negate();
canvasPicture.copyPictureTo(flower1Picture,200);

/* clear the blue in flower2 picture and add at x=300 in
 * the canvas */
flower2Picture.clearBlue();
canvasPicture.copyPictureTo(flower2Picture,300);

// copy the negated flower1 to x=400
canvasPicture.copyPictureTo(flower1Picture,400);

// show the resulting picture
canvasPicture.show();

return canvasPicture;
}

The method createCollageBetter is much easier to read and understand now. And, we now have a method copyPictureTo which is easy to reuse.

We can even make a more general copy method which takes both the starting x and starting y values for the target picture and copies the passed source picture into the current picture with the source pictures upper left corner at the passed starting x and y values.

/**
 * Method that will copy all of the passed source picture into
 * the current picture object starting with the left corner
 * given by xStart, yStart
 * @param sourcePicture the picture object to copy
 * @param xStart the x position to start the copy into on the
 * target
 * @param yStart the y position to start the copy into on the
 * target
 */
public void copyPictureTo(Picture sourcePicture,
                          int xStart,
                          int yStart)
Section 4.2  Copying and transforming pictures  139

Notice that you can have two methods with the same names (like `copyPictureTo`) and you don’t have any trouble when you compile. How can that be? Java allows you to have many methods with the same method name as long as the parameters are different. The first `copyPictureTo` method took a `Picture` object and an int. The second `copyPictureTo` method took a `Picture` object, and two int values. So the two methods have a different number of parameters. Having more than one method with the same name but different parameters is called overloading. It doesn’t really matter what you name the parameters. What matters in the types. Two methods with the same name are allowed if the number of parameters is different, or the types of the parameters are different, or the order of the parameter types is different.

4.2.3 Blending Pictures

When we create collages by copying, any overlap typically means that one picture shows over another. The last picture painted on is the one that appears. But it doesn’t have to be that way. We can blend pictures by multiplying their colors and adding them. This gives us the effect of transparency.

We know that 100% of something is the whole thing. 50% of one and 50% of another would also add up to 100%. In the program below, we blend a picture of the two sisters with an overlap of some 50 (the width of Katie minus 150) columns of pixels (Figure 4.12).

```
{  
Pixel sourcePixel = null;
Pixel targetPixel = null;

  // loop through the columns
  for (int sourceX = 0, targetX = xStart;
       sourceX < sourcePicture.getWidth();
       sourceX++, targetX++)
  {
    // loop through the rows
    for (int sourceY = 0,
         targetY = yStart;
         sourceY < sourcePicture.getHeight();
         sourceY++, targetY++)
    {
      sourcePixel = sourcePicture.getPixel(sourceX, sourceY);
      targetPixel = this.getPixel(targetX, targetY);
      targetPixel.setColor(sourcePixel.getColor());
    }
  }
}
```

Program 30: Blending two pictures
/**
 * Method to blend pictures of Katie and Jenny
 * @return the blended picture
 */
public static Picture blendPictures()
{

    // create the three pictures
    Picture katiePicture =
        new Picture(FileChooser.getMediaPath("KatieFancy.jpg"));
    Picture jennyPicture =
        new Picture(FileChooser.getMediaPath("JenParty.jpg"));
    Picture canvasPicture =
        new Picture(FileChooser.getMediaPath("640x480.jpg"));

    // declare the source and target pixel variables
    Pixel katiePixel = null;
    Pixel jennyPixel = null;
    Pixel targetPixel = null;

    // declare the target x and source x since we will need
    // the values after the for loop
    int sourceX = 0;
    int targetX = 0;

    // declare a color variable for the blended color
    Color blendColor = null;

    // copy the first 150 pixels of katie to the canvas
    for (; sourceX < 150; sourceX++, targetX++)
    {
        for (int sourceY = 0, targetY = 0;
             sourceY < katiePicture.getHeight();
             sourceY++, targetY++)
        {
            katiePixel = katiePicture.getPixel(sourceX, sourceY);
            targetPixel = canvasPicture.getPixel(targetX, targetY);
            targetPixel.setColor(katiePixel.getColor());
        }
    }

    // copy 50% of katie and 50% of jenny till the end of
    // katie's width
    for (; sourceX < katiePicture.getWidth();
         sourceX++, targetX++)
    {
        for (int sourceY = 0, targetY = 0;
             sourceY < katiePicture.getHeight();
             sourceY++, targetY++)
        {

        }
    }
}
katiePixel = katiePicture.getPixel(sourceX, sourceY);  
jennyPixel =  
jennyPicture.getPixel(sourceX - 150, sourceY);  
targetPixel = canvasPicture.getPixel(targetX, targetY);  
blendColor =  
    new Color((int) (katiePixel.getRed() * 0.5 +  
                jennyPixel.getRed() * 0.5),  
               (int) (katiePixel.getGreen() * 0.5 +  
                      jennyPixel.getGreen() * 0.5),  
               (int) (katiePixel.getBlue() * 0.5 +  
                      jennyPixel.getBlue() * 0.5))  
targetPixel.setColor(blendColor);  
}  
}  
// copy the rest of Jenny  
sourceX = sourceX - 150;  
for (; sourceX < jennyPicture.getWidth();  
    sourceX++, targetX++)  
{  
    for (int sourceY = 0, targetY = 0;  
        sourceY < jennyPicture.getHeight();  
        sourceY++, targetY++)  
    {  
        jennyPixel = jennyPicture.getPixel(sourceX, sourceY);  
        targetPixel = canvasPicture.getPixel(targetX, targetY);  
        targetPixel.setColor(jennyPixel.getColor());  
    }  
}  
// show the canvas  
canvasPicture.show();  
// return the canvas  
return canvasPicture;
Chapter 4  Modifying Pixels in a Matrix

FIGURE 4.12: Blending the picture of Katie and Jenny

Making it Work Tip: Optional parts of the for loop
Notice that we are missing the initialization area in the for loops in the method blendPictures(). Also notice that we moved the declaration of sourceX and sourceY outside the loops. This is because we want to keep the values around after the first loop ends. The initialization area of a for loop is optional (the ; is not optional). In fact, the initialization area, continuation test, and change area are all optional. You could code a for loop as for (;;;) but that isn’t terribly useful. It would execute the body of the loop forever. This is known as an infinite loop.

4.2.4 Rotation
Transformations to the image occur by using the index variables differently or incrementing them differently, but otherwise keeping the same program. Let’s rotate Katie 90 degrees to the left. What does that mean? Let’s try it with something simple first. You can write some numbers in a table on a piece of paper and then rotate it left and then read the new table to see where the old numbers were moved to (Figure 4.13). Notice that the columns become the rows and the rows the columns but it isn’t as simple as just using the source x for the target y and the source y for the target x.

Value (0,0) in the source moves to (0,2) in the target. Value (0,1) in the source moves to (1,2) in the target. Value (1,0) in the source moves to (0,1) in the target. Value (1,1) in the source moves to (1,1) in the target. Value (2,0) in the
Section 4.2  Copying and transforming pictures  143

source moves to (0,0) in the target. Value (2,1) in the source moves to (1,0) in
the target. So the first column values move into the bottom row and the last column
values move into the top row. Also notice that the target x value is the same as
the source y value.

We will do the rotation by looping through the pixels in the usual way and
getting the source pixel in the usual way but the target pixel’s x value will be the
source y and the target pixel’s y value will be width of the source picture - 1 - the
source x (Figure 4.14).

![Figure 4.13: Rotating some numbers in a table to the left 90 degrees](image)

Program 31: Rotating a picture left 90 degrees

```java
/**
* Method to copy the picture of Katie but rotate her left 90
* degrees on the resulting picture
*/
public static Picture copyKatieLeftRotation ()
{
    String sourceFile =
        FileChooser.getMediaPath("KatieFancy.jpg");
    Picture sourcePicture = new Picture(sourceFile);
    String targetFile =
        FileChooser.getMediaPath("7inx95in.jpg");
    Picture targetPicture = new Picture(targetFile);
    Pixel sourcePixel = null;
    Pixel targetPixel = null;

    // loop through the columns
    for (int sourceX = 0;
        sourceX < sourcePicture.getWidth();
        sourceX++)
    {
        // loop through the rows
        for (int sourceY = 0;
            sourceY < sourcePicture.getHeight();
            sourceY++)
        {
```
Chapter 4  Modifying Pixels in a Matrix

```java
sourceY++)
{
    // set the target pixel color to the source pixel color
    sourcePixel = sourcePicture.getPixel(sourceX, sourceY);
    targetPixel = targetPicture.getPixel(sourceY,
                                           sourcePicture.getWidth() - 1 - sourceX);
    targetPixel.setColor(sourcePixel.getColor());
}

// show the source and target pictures
sourcePicture.show();
targetPicture.show();

return targetPicture;
}
```

FIGURE 4.14: Copying a picture to a canvas rotated to the left 90 degrees

How does that work?

Rotating starts with the same source and target, and even the same variable values, but since we use the target X and Y differently, we get a different effect.

Now, as we increment the Y variables, we’re moving down the source, but across the target from left to right. As we increment the X variables we’re moving across the source but up the target.
Section 4.2   Copying and transforming pictures  

The source $x$ and source $y$ are both 0. The target $x$ is equal to the source $y$ so it is also 0. But, the target $y$ is equal to the width of the source picture minus 1 minus the source $x$. The width of the source picture is 3 so the target $y$ is $3 - 1 - 0$ which is 2. So we copy the color of the source pixel at $(0,0)$ to the target pixel at $(0,2)$.

The source $y$ is incremented by the inner loop to 1 and tested against the height of the source picture (2). Since it is less than the height we do the body of the inner loop. So now the source $x$ is 0 and the source $y$ is 1. The target $x$ is equal to the source $y$ so it is 1. The target $y$ is equal to the width of the source picture minus 1 minus the source $y$. The width of the source picture is 3 so the target $y$ is $3 - 1 - 0$ which is 2. So we copy the color of the source pixel at $(0,1)$ to the target pixel at $(1,2)$. 

The source $x$ is incremented by the inner loop to 1 and tested against the height of the source picture (2). Since it is less than the height we do the body of the inner loop. So now the source $x$ is 1 and the source $y$ is 0. The target $x$ is equal to the source $y$ so it is 0. The target $y$ is equal to the width of the source picture minus 1 minus the source $y$. The width of the source picture is 3 so the target $y$ is $3 - 1 - 1$ which is 1. So we copy the color of the source pixel at $(1,0)$ to the target pixel at $(0,1)$. 

The source $y$ is incremented by the inner loop to 1 and tested against the height of the source picture (2). Since it is less than the height we do the body of the inner loop. So now the source $x$ is 1 and the source $y$ is 1. The target $x$ is equal to the source $y$ so it is 1. The target $y$ is equal to the width of the source picture minus 1 minus the source $y$. The width of the source picture is 3 so the target $y$ is $3 - 1 - 1$ which is 1. So we copy the color of the source pixel at $(1,1)$ to the target pixel at $(1,2)$. 

The source $x$ is incremented by the inner loop to 1 and tested against the height of the source picture (2). Since it is less than the height we do the body of the inner loop. So now the source $x$ is 2 and the source $y$ is 0. The target $x$ is equal to the source $y$ so it is 0. The target $y$ is equal to the width of the source picture minus 1 minus the source $y$. The width of the source picture is 3 so the target $y$ is $3 - 1 - 0$ which is 2. So we copy the color of the source pixel at $(2,0)$ to the target pixel at $(0,2)$. 

The source $x$ is incremented by the inner loop to 2 and tested against the height of the source picture (2). Since it is less than the height we do the body of the inner loop. So now the source $x$ is 2 and the source $y$ is 1. The target $x$ is equal to the source $y$ so it is 1. The target $y$ is equal to the width of the source picture minus 1 minus the source $y$. The width of the source picture is 3 so the target $y$ is $3 - 1 - 1$ which is 1. So we copy the color of the source pixel at $(2,1)$ to the target pixel at $(1,2)$.
Chapter 4  Modifying Pixels in a Matrix

The source y is incremented by the inner loop to 2 and tested against the height of the source picture (2). Since it is not less than the height the inner loop finishes and the source x is incremented to 1 by the outer loop. The inner loop starts and sets the source y to 0. So, the source x is 1 and the source y is 0. The target x is equal to the source y so it is 0. The target y is equal to the width of the source picture minus 1 minus the source y. The width of the source picture is 3 so the target y is 3 - 1 - 1 which is 1. So we copy the color of the source pixel at (1,0) to the target pixel at (0,1).

The inner loop will increment source y and so the next color will be copied from (1,1) to (1,1). Then, the inner loop will stop again and source x will be incremented by 1 to 2. The next color will be copied from (2,0) to (0,0). The inner loop will increment source y so the next color will be copied from (2,1) to (1,0). At this point source x will be incremented to 3 which is not less than the width of the source picture (3) and the nested loop will stop.

4.2.5  Scaling

A very common transformation for pictures is to scale them. Scaling up means to make them larger, and scaling them down makes them smaller. It’s common to scale a 1-megapixel or 3-megapixel picture down to a smaller size to make it easier to use on the Web. Smaller pictures require less disk space, and thus less network bandwidth, and thus are easier and faster to download.

Scaling a picture requires the use of sampling which we’ll also use with sounds later. To scale a picture smaller we are going to take every other pixel when copying from the source to the target. To scale a picture larger we are going to take every pixel twice.

Scaling the picture down is the easier function. We will use the passion-Flower.jpg picture which is 640 (width) by 480 (height). Instead of incrementing the source X and Y variables by 1, we simply increment by 2. We divide the amount of space by 2, since we’ll fill half as much room–our width will be 640/2 and the height will be 480/2. The result is a smaller flower in the canvas (Figure 4.15).

** Program 32: Scaling a picture down (smaller)**

```java
/**
 * Method to copy the flower but smaller (half as big)
 * @return the smaller flower picture
 */
public static Picture copyFlowerSmaller()
{
    Picture flowerPicture = new Picture(FileChooser.getMediaPath("passionFlower.jpg"));
    Picture canvasPicture = new Picture(FileChooser.getMediaPath("640x480.jpg"));
    Pixel sourcePixel = null;
    Pixel targetPixel = null;
```
Section 4.2  Copying and transforming pictures  147

// loop through the columns
for (int sourceX = 0, targetX=0;
     sourceX < flowerPicture.getWidth ();
     sourceX+=2, targetX++)
{
  // loop through the rows
  for (int sourceY=0, targetY=0;
       sourceY < flowerPicture.getHeight ();
       sourceY+=2, targetY++)
  {
    sourcePixel = flowerPicture.getPixel (sourceX, sourceY);
    targetPixel = canvasPicture.getPixel (targetX, targetY);
    targetPixel.setColor (sourcePixel.getColor ());
  }
}

// show the resulting picture
canvasPicture.show ();
return canvasPicture;

FIGURE 4.15: Scaling the picture down

Scaling up the picture (making it larger) is a little trickier. We want to take
every pixel twice. What we’re going to do is to increment the source index variables
by 0.5. Now, we can’t reference pixel 1.5. But if we reference (int) 1.5 we’ll get
1 again, and that’ll work. The sequence of 1, 1.5, 2, 2.5... will become 1,1,2,2...
The result is a larger form of the picture (Figure 4.16). Let’s try this on rose.jpg
which is 320 by 240 so scaling it up will result in a picture that is 640 by 480.

Program 33: Scaling the picture up (larger)

/***/
Chapter 4  Modifying Pixels in a Matrix

* Method to copy a flower but scaled to 2x normal size
* @return the larger flower
*/
public static Picture copyFlowerLarger()
{
    Picture flowerPicture =
        new Picture(FileChooser.getMediaPath("rose.jpg"));
    Picture canvasPicture =
        new Picture(FileChooser.getMediaPath("640x480.jpg"));
    Pixel sourcePixel = null;
    Pixel targetPixel = null;

    // loop through the columns
    for (double sourceX = 0, targetX=0;
        sourceX < flowerPicture.getWidth();
        sourceX = sourceX + 0.5, targetX++)
    {
        // loop through the rows
        for (double sourceY=0, targetY=0;
            sourceY < flowerPicture.getHeight();
            sourceY = sourceY + 0.5, targetY++)
        {
            sourcePixel =
                flowerPicture.getPixel((int)sourceX,(int)sourceY);
            targetPixel =
                canvasPicture.getPixel((int)targetX,(int)targetY);
            targetPixel.setColor(sourcePixel.getColor());
        }
    }

    // show the resulting picture
    canvasPicture.show();

    return canvasPicture;
}

Since this is a static method it can be invoked on the class.

> Picture p = Picture.copyFlowerLarger();
> p.explore();

How did that work?.
We start from the same place as the original code for copying a picture. Say we are copying from the source picture starting at (0,0) and copying to the target picture starting at (3,1). First we will copy the color of the pixel at (0,0) in the source picture to (3,1) in the target picture.
When we increment \( sourceY \) by 0.5, the actual value will be 0.5 but the (int) value is 0 so we end up referring to the same pixel in the source, but the target has moved on to the next pixel. So we will copy the color of the pixel at \((0,0)\) to \((3,2)\).

When we increment \( sourceY \) a second time by 0.5 it will now equal 1.0, so we now move on to the next pixel in the source. So we will copy the color of the pixel at \((0,1)\) to \((3,3)\).

Again when the \( sourceY \) is incremented by 0.5 the actual value will be 1.5 but the (int) of that is 1 so we will copy from \((0,1)\) to \((3,4)\).
And eventually, we cover every pixel. Notice that the end result is degraded—it’s choppier than the original. Each pixel is copied four times: twice in the x direction and twice in the y direction.

You might want to be able to scale a picture to a particular size, instead of always using the canvas pictures. There is a constructor that takes a width and height new Picture(width, height) and creates a blank picture of the passed width and height (both specified in pixels). new Picture(640, 480) would create a picture object that is 640 pixels wide by 480 pixels tall—just like the canvas.

Here is a more general method that will scale up the current picture object by some passed number of times. It uses the constructor that specifies a width and height for the new Picture object.

**Program 34: General scale method**

```java
/**
 * Method to create a new picture that is scaled up by the
 * passed number of times.
 * @return the new scaled up picture
 */
public Picture scaleUp(int numTimes)
{
    Picture targetPicture =
        new Picture(this.getWidth() * numTimes,
                    this.getHeight() * numTimes);
    double factor = 1.0 / numTimes;
    Pixel sourcePixel = null;
    Pixel targetPixel = null;

    // loop through the columns
    for (double sourceX = 0, targetX=0;
         sourceX < this.getWidth();
         sourceX += factor;
    
    return targetPicture;
}
```
Section 4.2 Copying and transforming pictures

sourceX = sourceX + factor, targetX++)
{
    // loop through the rows
    for (double sourceY=0, targetY=0;
        sourceY < this.getHeight();
        sourceY = sourceY + factor, targetY++)
    {
        sourcePixel =
            this.getPixel((int) sourceX,(int) sourceY);
        targetPixel =
            targetPicture.getPixel((int) targetX,(int) targetY);
        targetPixel.setColor(sourcePixel.getColor());
    }
}

// show the resulting picture
targetPicture.show();

return targetPicture;

You can use this to scale any picture.

> Picture p = new Picture(FileChooser.getMediaPath("flower1.jpg"));
> p = p.scaleUp(2);
> p.explore();

OBJECTS AND METHODS SUMMARY

Here are the functions used or introduced in this chapter:

<table>
<thead>
<tr>
<th>new Picture(int width, int height)</th>
<th>Creates a new Picture object with the given width and height. All the pixels are black.</th>
</tr>
</thead>
<tbody>
<tr>
<td>getMediaPath(String fileName)</td>
<td>Returns the full path name with the media directory followed by the passed file name. This is a class method on the FileChooser class. The default media directory is &quot;c:/intro-prog-java/mediasources/&quot;.</td>
</tr>
<tr>
<td>setMediaPath(String directory)</td>
<td>Sets the media directory to use when getting a full path using getMediaPath(String fileName). This is a class method on the FileChooser class.</td>
</tr>
</tbody>
</table>

PROBLEMS

4.1. Write the code to mirror a picture around a horizontal line from (0,height-1) to (width-1,height-1). Be aware that this will double the height of the picture.
4.2. Try to mirror a picture around a diagonal line from (0,0) to (width-1,height-1). Try to mirror a picture around a diagonal line from (0,height-1) to (width-1,0). 4.3. Write the code to rotate a picture to the right by 90 degrees.
4.4. Rewrite any of the methods that are static to be object methods. For example rewrite copyFlowerLarger as copyLarger which copies the current picture onto a new picture created with new Picture(this.getWidth() * 2, this.getHeight())
Chapter 4  Modifying Pixels in a Matrix

4.5. We’ve seen that if you increment the source picture index by 2 while incrementing the target picture index by 1 for each copied pixel, you end up with the source being scaled down onto the target. What happens if you increment the target picture index by 2 as well? What happens if you increment both the source and target by 0.5 and use int to get just the integer part?

4.6. Modify the method `createCollageBetter` to call the more general method `copyPicture`.

4.7. Write a class method named `createCollage` to create a collage of the same image at least four times fit onto the 7x95in.jpg blank JPEG. (You are welcome to add additional images, too.) One of those four copies can be the original picture. The other three should be modified forms. You can do any of scaling, cropping, or rotating the image; creating a negative of the image; shifting or altering colors on the image; and making it darker or lighter.

After composing your image, mirror it. You can do it vertically or horizontally (or otherwise), in any direction just make sure that your four base images are visible still after mirroring.

Your single method should make all of this happen—all of the effects and compositing must occur from the single function `createCollage`. Of course, it is perfectly okay to use other functions, but make it so that a tester of your program need only only to call `setMediaPath()` and put all your input pictures in her mediabackground directory, and then execute `createCollage()`—and will expect to see a collage generated, shown, and returned.

4.8. Think about how the grayscale algorithm works. Basically, if you know the luminance of anything visual (e.g., a small image, a letter), you can replace a pixel with that visual element in a similar way to create a collage image. Try implementing that. You’ll need 256 visual elements of increasing lightness, all of the same size. You’ll create a collage by replacing each pixel in the original image with one of these visual elements.

TO DIG DEEPER

The bible of computer graphics is Introduction to Computer Graphics [10]. It’s highly recommended.
4.3 COLOR FIGURES

FIGURE 4.17: Merging red, green, and blue to make new colors

FIGURE 4.18: Color: RGB triplets in a matrix representation
Chapter 4  Modifying Pixels in a Matrix

FIGURE 4.19: Color: The original picture (left) and red-decreased version (right)

FIGURE 4.20: Color: Overly blue (left) and red increased by 30% (right)

FIGURE 4.21: Color: Original (left) and blue erased (right)
FIGURE 4.22: Original beach scene (left) and at (fake) sunset (right)

FIGURE 4.23: Color: Lightening and darkening the original picture

FIGURE 4.24: Color: Negative of the image
Chapter 4 Modifying Pixels in a Matrix

FIGURE 4.25: Color: Color picture converted to grayscale

FIGURE 4.26: Color: Increasing reds in the browns
FIGURE 4.27: Color: Increasing reds in the browns, within a certain range

FIGURE 4.28: Finding the range where Jenny’s eyes are red, then changing them to black
FIGURE 4.29: Frames from the slow sunset movie

FIGURE 4.30: Frames from the slow fade-out movie
FIGURE 4.31: Frames from the Mommy watching Katie movie

FIGURE 4.32: Frames from the original too dark movie
FIGURE 4.33: Frames from the modified lighter movie

FIGURE 4.34: Frames from the original movie with kids crawling in front of a blue screen
FIGURE 4.35: Frames from the kids on the moon movie
CHAPTER 5

Conditionally Modifying Pixels

5.1 CONDITIONAL PIXEL CHANGES
5.2 SIMPLE EDGE DETECTION: CONDITIONALS WITH TWO OPTIONS
5.3 SEPIA TONED AND POSTERIZED PICTURES: USING MULTIPLE CONDITIONALS TO CHOOSE THE COLOR
5.4 HIGHLIGHTING LIGHTEST AND DARKEST AREAS
5.5 COMBINING PIXELS: BLURRING
5.6 BACKGROUND SUBTRACTION
5.7 CHROMAKEY

Chapter Learning Objectives
The media learning goals for this chapter are:

- To replace one color with another in a picture.
- To do simple edge detection.
- To replace several colors in a picture: sepia-toned.
- To replace a range of colors with one color: posterizing.
- To average nearby pixels when scaling up for a smoother result: blur.
- To replace the background in a picture.

The computer science goals for this chapter are:

- To conditionally execute a statement or block of statements using if.
- To convert a class method into an object method.
- To use a conditional with two possible results: if and else.
- To use a conditional with greater than two possible results: if, else if, and else.
- To use boolean expressions with and and or.
- To define method overloading and give an example.
5.1 CONDITIONAL PIXEL CHANGES

So far we have been processing all of the pixels in the same way. But, what if we want to process the pixels in different ways? For example, we might want to turn someone’s hair a different color, or get rid of ‘red-eye’ in a picture, or even reduce the number of colors in a picture.

We need something that executes a block of code only if some condition (expression) is true. We know that computers can compare values to see if they are equal, less than, or greater than (and combinations of these) and return true or false. We have used this with loops by continuing a loop while the index is less than the length of the array as in: while(index < pixelArray.length). This will execute the loop while the expression (index < pixelArray.length) is true.

One way to conditionally execute code in Java is with an if (expression). The if is a keyword that means that if the expression in the parentheses evaluates to true then execute the following statement or block of statements. If it is false just skip the execution of that statement or block of statements (Figure 5.1) So, we can check if the color at the current pixel is close to a particular color and if so execute a statement or block of statements (in curly braces).

```
> 0 < 20
true
> 30 < 20
false
> 20 < 20
false
> 20 <= 20
true
```

Try the following in the interactions pane:

```
> int x = 30;
> if (x < 40) System.out.println("x is less than 40");
```
Chapter 5 Conditionally Modifying Pixels

x is less than 40
> if (x > 40) System.out.println("x is greater than 40");
> System.out.println(x);
30

Notice that since x is less than 40 the string saying so was output. However, since x is not greater than 40 the string saying x is greater than 40 was not output. We do see the output from the next statement System.out.println(x) since execution jumps to the statement following an if when the expression is false.

5.1.1 Comparing Colors

What does it mean to compare two colors? How can the computer tell if the color at the current pixel is “red”? The distance between two colors is the Cartesian distance between the colors as points in a three-dimensional space, where red, green, and blue are the three dimensions. Recall that the distance between two points \((x_1, y_1)\) and \((x_2, y_2)\) is:
\[
\sqrt{(x_1-x_2)^2 + (y_1-y_2)^2}
\]
The similar measure for two colors \((\text{red}_1, \text{green}_1, \text{blue}_1)\) and \((\text{red}_2, \text{green}_2, \text{blue}_2)\) is:
\[
\sqrt{(	ext{red}_1-\text{red}_2)^2 + (\text{green}_1-\text{green}_2)^2 + (\text{blue}_1-\text{blue}_2)^2}
\]
However, you won’t have to code this. The Pixel class has an object method colorDistance(color) which returns the distance between the color in the current Pixel object and the passed color. The hard part is determining what “close enough” is for two colors.

5.1.2 Replacing Colors

Here’s a program that tries to replace the brown color in Katie’s hair with red. Mark used the picture explorer to figure out roughly what the RGB values were for Katie’s brown hair, then wrote a program to look for colors close to that, and then increase the redness of those pixels. Mark played a lot with the value that he used for distance (here, 50.0) and the amount of redness increase (here, 100% increase). However, this approach turned part of the couch and carpet red too. (Figure 5.2 and Figure 4.26).

Program 35: Color replacement: Turn Katie into a redhead

```java
/** *
 * Method to turn to turn Katie into a red head *
 */
public static Picture turnKatieRedHead() {
    Color brown = new Color(42,25,15);
    Picture katiePicture = new Picture(Picture.getMediaPath("KatieFancy.jpg"));
    Pixel[] pixels = katiePicture.getPixels();
    Pixel pixel = null;
```
Section 5.1  Conditional Pixel Changes

```java
// loop through the pixels
for (int i = 0; i < pixels.length; i++) {
    // get the current pixel
    pixel = pixels[i];

    // check if in distance to brown and if so double the red
    if (pixel.colorDistance(brown) < 50.0)
        pixel.setColor(new Color((int) (pixel.getRed() * 2.0),
                                 pixel.getGreen(),
                                 pixel.getBlue()));
}

// show the result
katiePicture.show();

return katiePicture;
```

This is a class method (notice the `static` keyword) because there is no `Picture` object to operate on when the method is started. The method creates the `Picture` object and returns it. Remember that you can execute class methods using `className.method()`. Here’s how to execute this method:

> Picture picture = Picture.turnKatieRedHead();

![Figure 5.2: Increasing reds in the browns](image)

“main”
2004/10/21
page 165
Chapter 5  Conditionally Modifying Pixels

Notice that we can use a simple for loop through the one-dimensional array of pixels for this. We don’t care where the pixels are in the two-dimensional array in this method. Of course, we could have used nested for loops instead to loop through all the pixels.

What this method is doing is looping through all the pixels in the picture and for each pixel checking if the distance between the color in the current pixel is less than 50 away from the color brown (defined as red=42, green=25, blue=15). If the distance between the current color and the defined brown is less than 50, the red value at the current pixel is doubled. If the distance is equal or greater than 50, the pixel color is not changed.

With the picture explorer we can also figure out the coordinates just around Katie’s face, and then just do the browns near her face. The effect isn’t too good, though it’s clear that it worked. The line of redness is too sharp and rectangular (Figure 5.3 and Figure 4.27).

Program 36: Color replacement in a range

```java
/**
 * Method to turn to turn Katie into a red head using a range
 */
public static Picture turnKatieRedHeadInRange() {
    Color brown = new Color(42,25,15);
    Picture katiePicture =
        new Picture(Picture.getMediaPath("KatieFancy.jpg");
    Pixel pixel = null;

    // loop through the x values
    for (int x=63; x < 125; x++) {
        for (int y=6; y < 76; y++) {
            // get the current pixel
            pixel = katiePicture.getPixel(x,y);

            /* check if in distance to brown and if so double
             * the red
             */
            if (pixel.colorDistance(brown) < 50.0)
                pixel.setColor(new Color((int) (pixel.getRed() * 2.0),
                                           pixel.getGreen(),
                                           pixel.getBlue()));
        }
    }

    // show the result
    katiePicture.show();

    return katiePicture;
}
```
This is a class method (notice the static keyword) because there is no Picture object to operate on when the method is started. The method creates the Picture object and returns it. Here’s how to invoke this method:

```java
> Picture picture = Picture.turnKatieRedHeadInRange();
```

![Figure 5.3: On left the couch color changes, on right the couch color doesn’t change](image)

### 5.1.3 Turning class methods into object methods

The method `turnKatieRedHeadInRange` is very similar to the method `turnKatieRedHead`. The only difference is that we don’t loop through the entire picture. We loop through the area that we want to change the pixels and then only change the current pixel color if it is less than a distance of 50 to the color brown.

Are the methods `turnKatieRedHead()` and `turnKatieRedHeadInRange()` easy to reuse? Each creates a Picture object from a specific filename and then operates on that picture. Both methods loop through pixels and use the same conditional to check if the color should be changed and each doubles the red. Do these need to be class methods? Well, they are class methods because no Picture object exists when they are called, but you could create the Picture object outside the method and then call the method on that Picture object. Most of the code in the methods works on a Picture object. So these really should be object methods and work on
Conditionally Modifying Pixels

the objects they are invoked on.

**Computer Science Idea: Methods that work on objects should be object methods**

Beginners to object-oriented programming often write class (static) methods to do the work of the program. But, if the method is working on objects it should be an object method. Limit class methods to create objects or for methods that operate on class fields. A main concept behind object-oriented programming is that you create objects and the objects simulate the domain. How is it an object simulation if you ask the classes to do all of the work?

Here is a more general object method that doubles the red in the pixels that are less than the passed distance from the passed color and in a rectangular area defined by the starting and ending x and y values.

**Program 37: General double red if in distance and in a range**

```java
/**
 * Method to double the red in a picture in a rectangular area if it is less than the passed distance to the passed color
 * @param startX the x value at the top left of the rectangular area
 * @param startY the y value at the top left of the rectangular area
 * @param endX the bottom right x value of the rectangular area
 * @param endY the bottom right y value of the rectangular area
 * @param distance the amount that the distance must be less than
 * @param compareColor the color to compare the current pixel color to
 */
public void doubleRedInRange(int startX, int startY, int endX, int endY, double distance, Color compareColor)
{
    Pixel pixel = null;
    // loop through the x values
    for (int x=startX; x<=endX; x++)
    {
        // loop through the y values
```
Section 5.1 Conditional Pixel Changes

for (int y=startY; y <= endY; y++)
{
    // get the current pixel
    pixel = this.getPixel(x,y);

    /* check if the distance is less than the
    * allowed distance and if so double the red
    */
    if (pixel.colorDistance(compareColor) < distance)
        pixel.setColor(new Color((int) (pixel.getRed() * 2.0),
                               pixel.getGreen(),
                               pixel.getBlue()));
}

Now you can use this new more general method to do the same thing as the method turnKatieRedHead by:

> import java.awt.Color;
> Picture p = new Picture(Picture.getMediaPath("katieFancy.jpg"));
> p.show();
> p.doubleRedInRange(0,0,p.getWidth()-1, p.getHeight()-1, 50,
> new Color(42,25,15));
> p.repaint();

And, you can use it to do the same thing as turnKatieRedHeadInRange by:

> import java.awt.Color;
> Picture p = new Picture(Picture.getMediaPath("katieFancy.jpg"));
> p.show();
> p.doubleRedInRange(63, 6, 124, 75, 50,new Color(42,25,15));
> p.repaint();

5.1.4 Reducing red eye

"Red eye" is the effect where the flash from the camera bounces off the back of the subject’s eyes. Reducing red eye is a really simple matter. We find the pixels that are “pretty close” (a distance from red of 165 works well) to red, then change those pixels’ color to a replacement color.

We probably don’t want to change the whole picture. In the Figure 5.4, we can see that Jenny is wearing a red dress—we don’t want to wipe out that red, too. We’ll fix that by only changing the area where Jenny’s eyes are. Using the picture explorer, we find the upper left and lower right corners of her eyes. Those points were (109,91) and (202,107).
Chapter 5  Conditionally Modifying Pixels

FIGURE 5.4: Finding the range of where Jenny’s eyes are red

```java
/** *
 * Method to remove red eye from the current picture object *
 * in the rectangle defined by startX, startY, endX, endY. *
 * The red will be replaced with the passed newColor *
 * @param startX the top left corner x value of a rectangle *
 * @param startY the top left corner y value of a rectangle *
 * @param endX the bottom right corner x value of a rectangle *
 * @param endY the bottom right corner y value of a rectangle *
 * @param newColor the new color to use *
 */
public void removeRedEye(int startX, int startY, int endX, int endY, Color newColor)
{
    Pixel pixel = null;

    /* loop through the pixels in the rectangle defined by the */
    /* startX, startY, and endX and endY */
    for (int x = startX; x < endX; x++)
    {
        for (int y = startY; y < endY; y++)
        {
            // get the current pixel
            pixel = getPixel(x,y);

            // if the color is near red then change it
            if (pixel.colorDistance(Color.red) < 167)
                pixel.setColor(newColor);
        }
    }
}
```
Section 5.2 Simple Edge Detection: conditionals with two options

We call this function with:

```java
> Picture jennyPicture = new Picture("c:/intro-prog-java/mediasources/jenny-red.jpg");
> jennyPicture.removeRedEye(109,91,202,107,java.awt.Color.black);
> jennyPicture.explore();
```

to replace the red with black—certainly other colors could be used for the replacement color. The result was good, and we can check that the eye really does now have all-black pixels (Figure 5.5). (See also Figure 4.28.)

![Figure 5.5: After fixing red-eye.](image)

5.2 SIMPLE EDGE DETECTION: CONDITIONALS WITH TWO OPTIONS

What if we want to look for areas of high contrast between a pixel and the pixel below it? If the contrast is high we can make the pixel black and if the difference is low we can make the pixel white. This is a simple form of edge detection. It results in a picture that looks like a pencil sketch.

What does high contrast mean? It means that the difference between the two colors is high. One way to calculate this is to average the red, green, and blue values in the top pixel and subtract this from the average of the red, green, and blue values in the bottom pixel. If the absolute value of the difference is greater than some amount, then there is high contrast. If there is high contrast set the top pixel color to black and if not set it to white.

You have seen a way to execute a statement or block of statements if some condition is true using an if. But, how do you execute a statement or block of
Chapter 5  Conditionally Modifying Pixels

statements if a condition is false? One way is to use an if but negate the condition using the '!' operator.

> !true
false
> !false
true

So if we wanted to check if a value was less than 20 and if it is print out "Yes, the value is less than 20" and if it isn’t print out "No, this value is not less than 20" we could test for the value being less than 20 and also test for the value being not less than 20 using the '!' operator.

> int x = 30;
> if (x < 20) System.out.println("Yes, the value is less than 20");
> if (! (x < 20)) System.out.println("No, this value is not less than 20");
No, this value is not less than 20

Another way to test that a way isn’t less than 20 is to test if it is greater or equal to 20.

> int x = 30;
> if (x < 20) System.out.println("Yes, the value is less than 20");
> if (x >= 20) System.out.println("No, this value is not less than 20");
No, this value is not less than 20

Yet another way to handle the two possibilities is with an if (expression) and else. As before the if part will execute the statement or block of statements following the if when the expression is true. But if we add an else this will also execute the statement or block of statements following the else when the expression following the if is false (Figure 5.8). To enter the following in DrJava’s interaction pane use SHIFT-ENTER after the if statement to let DrJava know that there will be a following else.

> int x = 30;
> if (x < 20) System.out.println("Yes, the value is less than 20");
else System.out.println("No, this value is not less than 20");

So, to do simple edge detection we will loop through the pixels with the outer loop incrementing y from 0 to less than picture height-1. The inner loop will increment x from 0 to less than the picture width. The top pixel will be set to the pixel at the current x and y location. The bottom pixel will be set to the pixel at the current x but y+1. We will compare the absolute value of the difference between the intensities (average of color values) and if it is less than some passed limit we will change the top pixel color to white and otherwise we will set the top pixel color to black. The Pixel class has an object method getAverage() that returns the average of the three color values.
Method to do a simple edge detection by comparing the absolute value of the difference between the colors intensities (average of the color values) between a pixel and the pixel below it. If the absolute value of the difference between the color intensities is less than a passed amount the top pixel color will be set to white. Otherwise it is set to black.

@param amount if the absolute value of the differences in the color average is less than this set the color to white, else black

```java
public void edgeDetection(double amount) {
    Pixel topPixel = null;
    Pixel bottomPixel = null;
    double topAverage = 0.0;
    double bottomAverage = 0.0;
    int endY = this.getHeight() - 1;

    /* loop through y values from 0 to height - 1 */
    for (int y = 0; y < endY; y++) {
        // loop through the x values from 0 to width
        for (int x = 0; x < this.getWidth(); x++) {
            // get the top and bottom pixels
            topPixel = this.getPixel(x, y);
            bottomPixel = this.getPixel(x, y + 1);

            // get the color averages for the two pixels
            topAverage = topPixel.getAverage();
            bottomAverage = bottomPixel.getAverage();

            // if the expression is true, perform statement
        }
    }
}
```
/* check if the absolute value of the difference * is less than the amount */
if (Math.abs(topAverage - bottomAverage) < amount) {
    topPixel.setColor(Color.WHITE);
    // else set the color to black
} else {
    topPixel.setColor(Color.BLACK);
}

To execute this method use:

> Picture p = new Picture(FileChooser.getMediaPath("butterfly1.jpg"));
> p.explore();
> p.edgeDetection(10);
> p.explore();

FIGURE 5.7: Original picture and after edge detection
5.3 SEPIA TONED AND POSTERIZED PICTURES: USING MULTIPLE CONDITIONALS TO CHOOSE THE COLOR

We handled the case of having two different ways to process the pixels using an if and else. What if we have more than two ways that we want to process some pixels? For example, what if we wanted to do one thing if a value is less than some number, another thing if it is equal and yet a third if it is greater than the number? We could check for each of these conditions with an if as shown below:

```java
> int y = 2;
> if (y < 10) System.out.println("Y is less than 10");
> else if (y == 10) System.out.println("y is equal to 10");
> if (y > 10) System.out.println("y is greater than 10");
```

This works but results in some unnecessary checking. Notice that y was equal to 10 and so that was printed out but it still executed the next statement which checked if y was greater than 10. But, can y be equal to 10 and greater than 10? What would have happened if y was less than 10? It would have printed out a string saying that y is less than 10 and then still checked if y was equal or greater than 10. We need something to say if the previous test was true only execute that and then skip to the end of all the checks. We have seen a way to do this for two possibilities (true or false) using if and else. One way to handle more three or more possibilities is with if, else if and finally else. You can use as many else if statements as needed. You are not required to have a final else.

```java
> int y = 10;
> if (y < 10) System.out.println("y is less than 10");
> if (y == 10) System.out.println("y is equal to 10");
```

Making it Work Tip: Using Curly Braces
You may have noticed that the method `edgeDetection` shows the starting curly braces at the end of the line instead of on a new line as shown in the previous methods. Java doesn’t care if the curly braces are at the end of a line or on a new line. Some programmers prefer one to another. It is often easier to see that you forgot a curly brace if it is on a new line. However, the Java guidelines say to put the opening curly brace at the end of a line and the closing one on a new line. You may also notice that we are using curly braces after the if and else even though there is only one statement to be executed, so they aren’t really needed. It is good practice to have them even if they aren’t needed because the code is easier to read and change.
So far, we’ve done color modification by simply saying “This color replaces that color.” We can be more sophisticated in our color swapping. We can look for a range of colors, by using if, else if, and else, and replace the color with some function of the original color or a specific color. The results are quite interesting.

For example, we might want to generate sepia-toned prints. Older prints sometimes have a yellowish tint to them. We could just do an overall color change, but the end result isn’t aesthetically pleasing. By looking for different kinds of color—highlights, middle ranges, and shadows—and treating them differently, we can get a better effect (Figure 5.9).

The way we do this is to first convert the picture to gray, both because older prints were in shades of gray, and because it makes it a little easier to work with. We then look for high, middle, and low ranges of color, and change them separately. We want to make the shadows (darkest grays) a bit darker. We want to make most of the picture (middle grays) into a brownish color. We want to the highlights (lightest grays) a bit yellow. Recall that yellow is a mixture of red and green so one way to make things yellow is to increase the red and green. Another way is to reduce the amount of blue. The advantage to reducing the blue is that you don’t
Section 5.3 Sepia toned and posterized pictures: Using multiple conditionals to choose the color

FIGURE 5.9: Original scene (left) and using our sepia-tone program

have to worry about increasing a value past 255.

Program 40: Convert a picture to sepia-tones

```java
/**
 * Method to change the current picture to a sepia tint (modify the middle colors to a light brown and the light colors to a light yellow and make the shadows darker
 */
public void sepiaTint() {
    Pixel pixel = null;
    double redValue = 0;
    double greenValue = 0;
    double blueValue = 0;

    // first change the current picture to grayscale
    this.grayscale();

    // loop through the pixels
    for (int x = 0; x < this.getWidth(); x++) {
        for (int y = 0; y < this.getHeight(); y++) {
            // get the current pixel and color values
            pixel = this.getPixel(x, y);
            redValue = pixel.getRed();
            greenValue = pixel.getGreen();
            blueValue = pixel.getBlue();
```
// tint the shadows darker
if (redValue < 60)
{
  redValue = redValue * 0.9;
  greenValue = greenValue * 0.9;
  blueValue = blueValue * 0.9;
}

// tint the midtones a light brown
// by reducing the blue
else if (redValue < 190)
{
  blueValue = blueValue * 0.8;
}

// tint the highlights a light yellow
// by reducing the blue
else
{
  blueValue = blueValue * 0.9;
}

// set the colors
pixel.setRed((int) redValue);
pixel.setGreen((int) greenValue);
pixel.setBlue((int) blueValue);
}
}

Try this method out by:

> Picture picture = new Picture(Picture.getMediaPath("gorge.jpg"));
> picture.show();
> picture.sepiaTint();
> picture.repaint();

Posterizing is a process of converting a picture to a smaller number of colors. We’re going to do that by looking for specific ranges of color, then setting the color to one value in that range. The result is that we reduce the number of colors in the picture (Figure 5.10).

Program 41: Posterizing a picture

/ *
* Method to posterize (reduce the number of colors) in
* the picture. The number of reds, greens, and blues
* will be 4.
*/
Section 5.3  Sepia toned and posterized pictures: Using multiple conditionals to choose the color

FIGURE 5.10: Reducing the colors (right) from the original (left)

```java
public void posterize()
{
    Pixel pixel = null;
    int redValue = 0;
    int greenValue = 0;
    int blueValue = 0;

    // loop through the pixels
    for (int x = 0; x < this.getWidth(); x++) {
        for (int y = 0; y < this.getHeight(); y++) {

            // get the current pixel and colors
            pixel = this.getPixel(x, y);
            redValue = pixel.getRed();
            greenValue = pixel.getGreen();
            blueValue = pixel.getBlue();

            // check for red range and change color
            if (redValue < 64)
                redValue = 31;
            else if (redValue < 128)
                redValue = 95;
            else if (redValue < 192)
                redValue = 159;
            else
                redValue = 223;

            // check for green range
            if (greenValue < 64)
```


greenValue = 31;
else if (greenValue < 128)
greenValue = 95;
else if (greenValue < 192)
greenValue = 159;
else
    greenValue = 223;

// check for blue range
if (blueValue < 64)
    blueValue = 31;
else if (blueValue < 128)
    blueValue = 95;
else if (blueValue < 192)
    blueValue = 159;
else
    blueValue = 223;

// set the colors
pixel.setRed(redValue);
pixel.setGreen(greenValue);
pixel.setBlue(blueValue);
}
}

What’s really going on here, though, is setting up (a) a bunch of levels then (b) setting the value of red, green, or blue to the midpoint of that level. We can do this more generally using mathematics to compute the ranges for a desired number of levels and picking the midpoint. We need to check if the current value is in the range and if so set it to the midpoint of the range.

How do we check if a value is in a range? If we call the bottom of the range bottomValue and the top of the range topValue then we could use this math notation bottomValue <= testValue < topValue. However in Java we need to write it bottomValue <= testValue && testValue < topValue. The two ampersands ('&&') mean 'and’. If I say you have to set the table and sweep the floor, how many jobs do you have to do? The answer is two, or both of them. If I say you can set the table or sweep the floor, how many jobs do you have to do then? The answer is one, or just one of the two. Similarly if in Java you have if (expression && expression) then both expressions must be true for the body of the if to be executed. And, if you have if (expression || expression) then only one of the two expressions must be true for the body of the if to be executed. The || means ‘or’.

Below is the program for a flexible number of levels, and Figure 5.11 shows a couple of examples.
Section 5.3  Sepia toned and posterized pictures: Using multiple conditionals to choose the color

/**
 * Method to posterize (reduce the number of colors) in
 * the picture
 * @param numLevels the number of color levels to use
 */
public void posterize(int numLevels)
{
    Pixel pixel = null;
    int redValue = 0;
    int greenValue = 0;
    int blueValue = 0;
    int increment = (int) (256.0 / numLevels);
    int bottomValue, topValue, middleValue = 0;

    // loop through the pixels
    for (int x = 0; x < this.getWidth(); x++) {
        for (int y = 0; y < this.getHeight(); y++) {

            // get the current pixel and colors
            pixel = this.getPixel(x, y);
            redValue = pixel.getRed();
            greenValue = pixel.getGreen();
            blueValue = pixel.getBlue();

            // loop through the number of levels
            for (int i = 0; i < numLevels; i++) {
                // compute the bottom, top, and middle values
                bottomValue = i * increment;
                topValue = (i + 1) * increment;
                middleValue = (int) ((bottomValue + topValue - 1) / 2.0);

                /* check if current values are in current range and
                * if so set them to the middle value */
                if (bottomValue <= redValue &&
                    redValue < topValue)
                    pixel.setRed(middleValue);
                if (bottomValue <= greenValue &&
                    greenValue < topValue)
                    pixel.setGreen(middleValue);
                if (bottomValue <= blueValue &&
                    blueValue < topValue)
                    pixel.setBlue(middleValue);
            }
        }
    }
}
Chapter 5 Conditionally Modifying Pixels

FIGURE 5.11: Pictures posterized to two levels (left) and four levels (right)

Computer Science Idea: Overloading
Notice that we now have two methods called `posterize`. One method doesn't take any parameters and the other takes the number of levels. In Java you are allowed to have more than one method with the same name as long as the parameters are different. This is called overloading. You can have a different number of parameters as in this case. You can also use different types and/or orders for the parameters. It means that you can have several methods that have the same behavior but operate on different parameters. It certainly is easier than coming up with new names for methods that do that same thing.

5.4 HIGHLIGHTING LIGHTEST AND DARKEST AREAS

What if we want to highlight the lightest and darkest areas of a picture? Would we highlight areas that are less than some amount from white and less than the same amount from black? Is there any color that is both close to white and black? No, we would want to replace the color at all pixels that have a distance from white or a distance from black less than some amount. We used `&&` to mean ‘and’ in the
last program. In this program we will use ‘||’ to mean ‘or’.

Program 43: Highlight extremes

```java
/**
 * Method to replace the pixel colors in the current
 * picture object that have a color distance less than
 * the passed amount to white or black with the passed
 * replacement color
 * @param replacementColor the new color to use
 */
public void highlightLightAndDark(double amount, Color replacementColor) {
    Pixel pixel = null;

    // loop through all the pixels in the x direction
    for (int x = 0; x < getWidth(); x++) {
        // loop through all the pixels in the y direction
        for (int y = 0; y < getHeight(); y++) {
            // get the current pixel
            pixel = getPixel(x, y);

            // if the distance from white or black is less than the
            // passed amount use the replace color instead
            if (pixel.colorDistance(Color.white) < amount ||
                pixel.colorDistance(Color.black) < amount) {
                pixel.setColor(replacementColor);
            }
        }
    }
}
```

You can use this as follows:

```java
> Picture picture = new Picture(Picture.getMediaPath("butterfly1.jpg"));
> picture.explore();
> picture.highlightLightAndDark(50.0, java.awt.Color.yellow);
> picture.explore();
```

5.5 COMBINING PIXELS: BLURRING

When we make pictures larger (scaling them up), we usually get rough edges: Sharp steps to lines, which we call *pixelation*. We can reduce pixelation by *blurring* the image. What we do is set each pixel to an average of pixels around it. In this example, we go through all pixels (note the large loop that surrounds everything) and then in the X and Y dimensions, compute the average of the pixels to either
Conditionally Modifying Pixels

FIGURE 5.12: Original picture (left) and light or dark areas highlighted (right)

side of the pixel. It takes a picture, and a number (size) of pixels to compute the average.

Of course we need to be careful not to try and access pixels beyond the allowed values of the two-dimensional array of pixels. Try this in the interactions pane:

```java
> Picture p = new Picture(FileChooser.getMediaPath("caterpillar.jpg"));
> System.out.println(p.getWidth());
329
> System.out.println(p.getHeight());
150
> p.getPixel(330,160);
java.lang.ArrayIndexOutOfBoundsException: Coordinate out of bounds!
```

The `java.lang.ArrayIndexOutOfBoundsException` tells us that we tried to access an array element that was outside the allowed indices. If this happens when
Section 5.5 Combining pixels: Blurring

our program is executing the program will stop and report the exception.

So how do we check that the index values are acceptable? We know that the x
indices range from 0 to width - 1 and the y indices range from 0 to height - 1. So we
can use \( x >= 0 \&\& x < \text{this.getWidth()} \&\& y >= 0 \&\& y < \text{this.getHeight()} \).

Program 44: A simple blur

```java
/**
   * Method to blur the pixels
   * @param numPixels the number of pixels to average in all
   *   directions so if the numPixels is 2 then we will average
   *   all pixels in the rectangle defined by 2 before the
   *   current pixel to 2 after the current pixel
   */
public void blur(int numPixels)
{
    Pixel pixel = null;
    Pixel samplePixel = null;
    int redValue = 0;
    int greenValue = 0;
    int blueValue = 0;
    int count = 0;

    // loop through the pixels
    for (int x=0; x < this.getWidth(); x++) {
        for (int y=0; y < this.getHeight(); y++) {

            // get the current pixel
            pixel = this.getPixel(x,y);

            // reset the count and red, green, and blue values
            count = 0;
            redValue = greenValue = blueValue = 0;

            /* loop through pixel numPixels before x to
               * numPixels after x */
            for (int xSample = x - numPixels;
                 xSample <= x + numPixels;
                 xSample++) {
                for (int ySample = y - numPixels;
                     ySample <= y + numPixels;
                     ySample++) {

                    /* check that we are in the range of acceptable
                       * pixels */
                    if (xSample >= 0 && xSample < this.getWidth() &&
                        ySample >= 0 && ySample < this.getHeight()) {
```
Chapter 5: Conditionally Modifying Pixels

```java
samplePixel = this.getPixel(xSample, ySample);
redValue = redValue + samplePixel.getRed();
greenValue = greenValue + samplePixel.getGreen();
blueValue = blueValue + samplePixel.getBlue();
count = count + 1;
}
}

// use average color of surrounding pixels
Color newColor = new Color(redValue / count,
greenValue / count,
blueValue / count);
pixel.setColor(newColor);
}
}
```

Here is how to use this method:

```java
> Picture p = new Picture(Chooser.getMediaPath("flower1.jpg");
> p = p.scaleUp(2);
> p.explore();
> p.blur(2);
> p.explore();
```

Figure 5.13 shows the flower from the collage made bigger, then blurred. You can see the pixelation in the bigger version—the sharp, blocky edges. With the blur, some of that pixelation goes away. More careful blurs take into account regions of colors (so that edges between colors are kept sharp), and thus are able to reduce pixelation without removing sharpness.

![Figure 5.13: Making the flower bigger, then blurring to reduce pixellation](image.png)
5.6 BACKGROUND SUBTRACTION

Let’s imagine that you have a picture of someone, and a picture of where they stood without them there (Figure 5.14). Could you subtract the background of the person (i.e., figure out where the colors are close), and then replace another background? Say, of the moon (Figure 5.15)?

![Figure 5.14: A picture of a child (Katie), and her background without her](image)

![Figure 5.15: A new background, the moon](image)

Program 45: Subtract the background and replace it with a new one

```c
/**
 * Method to replace the background in the current picture
 * with the background from another picture
 * @param oldBackground a picture with the old background
 * to replace
```
Chapter 5  Conditionally Modifying Pixels

```java
public void swapBackground(Picture oldBackground, Picture newBackground)
{
    // loop through the columns
    for (int x=0; x<getWidth(); x++)
    {
        // loop through the rows
        for (int y=0; y<getHeight(); y++)
        {
            // get the current pixel and old background pixel
            currPixel = this.getPixel(x,y);
            oldPixel = oldBackground.getPixel(x,y);

            // if the distance between the current pixel color
            // and the old background pixel color is less
            // than the 15 then swap in the new background pixel
            if (currPixel.colorDistance(oldPixel.getColor()) < 15.0)
            {
                newPixel = newBackground.getPixel(x,y);
                currPixel.setColor(newPixel.getColor());
            }
        }
    }
}
```

To test if we can replace an old background with a new background try:

```java
> Picture p = new Picture(FileChooser.getMediaPath("kid-in-frame.jpg");
> Picture oldBg = new Picture(FileChooser.getMediaPath("bgframe.jpg");
> Picture newBg = new Picture(FileChooser.getMediaPath("moon-surface.jpg");
> p.swapBackground(oldBg,newBg);
> p.show();
```

We can, but the effect isn’t as good as we would like (Figure 5.16). Our daughter’s shirt color was too close to the color of the wall. And though the light was dim, the shadow is definitely having an effect here.

Mark tried the same thing with a picture of two students in front of a tiled wall. While Mark did use a tripod (really critical to get the pixels to line up), Mark unfortunately left autofocus on, so the two original pictures (Figure 5.17) weren’t all that comparable. The background swap (again with the jungle scene) hardly
did anything at all! We changed the threshold value to 50, and finally got some swapping (Figure 5.18).

```java
> Picture p = new Picture(FileChooser.getMediaPath("wall-two-people.jpg"));
> Picture oldBg = new Picture(FileChooser.getMediaPath("wall.jpg"));
> Picture newBg = new Picture(FileChooser.getMediaPath("beach.jpg"));
> p.swapBackground(oldBg,newBg);
> p.show();
```

Making it Work Tip: Add an input parameter to generalize a method

Notice that we changed the threshold from 15.0 to 50.0 for the second test of the `swapBackground(oldBG,newBG)` method. A better thing to do would be to change the method to take the threshold distance as another input parameter `swapBackground(oldBG,newBG,threshold)`. This means we won’t have to keep changing the method each time we want to change the threshold, which means the method can be used in more situations.

Program 46: Better Swap Background

```java
/**
 * Method to replace the background in the current picture
 * with the background from another picture
 * @param oldBackground a picture with the old background
 */
```
Chapter 5  Conditionally Modifying Pixels

public void swapBackground(Picture oldBackground, Picture newBackground, double threshold)
{
    Pixel currPixel = null;
    Pixel oldPixel = null;
    Pixel newPixel = null;

    // loop through the columns
    for (int x=0; x<getWidth(); x++)
    {
        // loop through the rows
        for (int y=0; y<getHeight(); y++)
        {
            // get the current pixel and old background pixel
            currPixel = this.getPixel(x,y);
            oldPixel = oldBackground.getPixel(x,y);

            /* if the distance between the current pixel color
             * and the old background pixel color is less than
             * the threshold then swap in the new background
             * pixel */
            if (currPixel.colorDistance(oldPixel.getColor()) < threshold)
            {
                newPixel = newBackground.getPixel(x,y);
                currPixel.setColor(newPixel.getColor());
            }
        }
    }
}

To make this work pass the threshold too when invoking swapBackground:

> Picture p = new Picture(FileChooser.getMediaPath("wall-two-people.jpg"));
> Picture oldBg = new Picture(FileChooser.getMediaPath("wall.jpg"));
> Picture newBg = new Picture(FileChooser.getMediaPath("beach.jpg"));
> p.swapBackground(oldBg,newBg,50);
> p.show();
FIGURE 5.17: Two people in front of a wall, and a picture of the wall

FIGURE 5.18: Swapping a beach for the wall, using background subtraction, with a threshold of 50

5.7 CHROMAKEY

The way that weatherpersons appear to be in front of a weather map that changes, is that they actually stand before a background of a fixed color (usually blue or green), then subtract that color. This is called chromakey. Mark took our son’s blue sheet, attached it to the entertainment center, then took a picture of himself in front of it, using a timer on a camera (Figure 5.19).

Mark tried a new way to test for “blueness”. If the blue value was greater than the sum of the red and green values then it the color was “blue”.

Program 47: Chromakey: Replace all blue with the new background

```java
/**
 * Method to do chromakey using a blue background
 * @param newBg the new background image to use to replace
 * the blue from the current picture
 */
public void chromakey(Picture newBg)
{
```
Chapter 5  Conditionally Modifying Pixels

FIGURE 5.19: Mark in front of a blue sheet

```java
Pixel currPixel = null;
Pixel newPixel = null;

// loop through the columns
for (int x = 0; x < getWidth(); x++)
{
    // loop through the rows
    for (int y = 0; y < getHeight(); y++)
    {
        // get the current pixel
        currPixel = this.getPixel(x, y);

        /* if the color at the current pixel mostly blue
         * (blue value is greater than red and green
         * combined, then use new background color
         */
        if (currPixel.getRed() + currPixel.getGreen() < currPixel.getBlue())
        {
            newPixel = newBg.getPixel(x, y);
            currPixel.setColor(newPixel.getColor());
        }
    }
}
```

The effect is really quite striking (Figure 5.20). Do note the “folds” in the lunar surface, though. The really cool thing is that this program works for any background that’s the same size as the image (Figure 5.21). To put Mark on the
moon and on the beach try this:

```java
> Picture mark = new Picture(FileChooser.getMediaPath("blue-mark.jpg"));
> Picture newBg = new Picture(FileChooser.getMediaPath("moon-surface.jpg"));
> mark.chromakey(newBg);
> mark.explore();
> mark = new Picture(FileChooser.getMediaPath("blue-mark.jpg"));
> newBg = new Picture(FileChooser.getMediaPath("beach.jpg"));
> mark.chromakey(newBg);
> mark.explore();
```

FIGURE 5.20: Mark on the moon

FIGURE 5.21: Mark on the beach
There’s another way of writing this code, which is shorter but does the same thing.

Program 48: Chromakey, shorter

```java
/**
 * Method to do chromakey using a blue background
 * @param newBg the new background image to use to replace
 * the blue from the current picture
 */
public void chromakeyBlue(Picture newBg)
{
    Pixel[] pixelArray = this.getPixels();
    Pixel currPixel = null;
    Pixel newPixel = null;

    // loop through the pixels
    for (int i = 0; i < pixelArray.length; i++)
    {
        // get the current pixel
        currPixel = pixelArray[i];

        /* if the color at the current pixel mostly blue
         * (blue value is greater than green and red
         * combined, then use new background color
         */
        if (currPixel.getRed() + currPixel.getGreen() <
            currPixel.getBlue())
        {
            newPixel = newBg.getPixel(currPixel.getX(),
                                       currPixel.getY());
            currPixel.setColor(newPixel.getColor());
        }
    }
}
```

Making it Work Tip: When do you need a different method name?

Notice that we used a different name for the new shorter method. We couldn’t have used the same name and had both methods in our Picture class since the parameters are the same. Methods can be overloaded (use the same name) as long as the parameters are different (in number, or order, or type).

You don’t really want to do chromakey with a common color, like red—
something that there’s a lot of in your face. Mark tried it with the two pictures in Figure 5.22—one with the flash on, and one with it off. We changed the test to if (currPixel.getRed() > currPixel.getGreen() + currPixel.getBlue()). The one without a flash was terrible—the student’s face was replaced with the background. The one with the flash was better, but the flash is still clear after the swap (Figure 5.23). It’s clear why moviemakers and weather people use blue or green backgrounds.

FIGURE 5.22: Student in front of a red background, and with flash on

FIGURE 5.23: Using chromakey program with a red background

PROBLEMS

5.1. Try doing chromakey in a range—grab something out of its background where the something is only in one part of a picture. For example, put a halo around someone’s head, but don’t mess with the rest of their body.

5.2. Write a method to copy all but the white pixels from one picture to another. Use this to put the robot in robot.jpg on the moon in moon-surface.jpg.

5.3. Start with a picture of someone you know, and make some specific color changes to it:
   • Turn the skin green
   • Turn the eyes red.
   • Turn the hair orange.
Chapter 5  Conditionally Modifying Pixels

Of course, if you’re friend’s skin is already green, or eyes red, or hair orange—choose a different target color.

5.4. Which of the methods below removes all the blue from every pixel of a picture that already has a blue value of more than 100?

1. A only
2. D only
3. B and C
4. C and D
5. None
6. All

What do the other ones do?

A. 

```java
public void blueOneHundred() {
    Pixel p = null;

    for (int x = 0; x < 100; x++) {
        for (int y = 0; y < 100; y++) {
            p = getPixel(x, y);
            p.setBlue(100);
        }
    }
}
```

B. 

```java
public void blueChange() {
    Pixel[] pixelArray = getPixels();
    Pixel pixel = null;

    for (int i = 0; i < pixelArray.length; i++) {
        pixel = pixelArray[i];
        if (pixel.getBlue() > 0) {
            pixel.setBlue(100);
        }
    }
}
```

C. 

```java
public void clearSomeBlue() {
    Pixel[] pixelArray = getPixels();
    Pixel pixel = null;

    for (int i = 0; i < pixelArray.length; i++) {
        pixel = pixelArray[i];
        if (pixel.colorDistance(Color.BLUE) > 100)
            pixel.setBlue(0);
    }
```
Section 5.7 Chromakey

D.

```java
public void setBlue() {
    Pixel[] pixelArray = getPixels();
    Pixel pixel = null;

    for (int i = 0; i < pixelArray.length; i++) {
        pixel = pixelArray[i];
        if (pixel.getBlue() > 100)
            pixel.setBlue(0);
    }
}
```

5.5. Convert the class method `copyFlowerLarger()` to an object method that can scale any picture up. It should return a new picture object created using `new Picture(this.getWidth() * 2, this.getHeight() * 2)`.

5.6. Write the method to turn the lightest areas of a picture gray to simulate a fog.

5.7. Write other edge detection methods. Try comparing the current pixel intensity with the one on the right. Try comparing the current pixel to the average of the pixels to the right and below.

5.8. What would the output from the following be:

```java
int x = 30;
for (int i=x; i < 40; i++) {
    if (i < 35) System.out.println("i is less than 35");
    else if (i == 35) System.out.println("i is 35");
    else System.out.println("i is greater than 35");
}
```

5.9. Write a method to do green or red chromakey.

5.10. Write a method to do chromakey in a specified rectangular region.
PART THREE

SOUNDS

Chapter 6  Modifying all Samples in a Sound
Chapter 7  Modifying Samples using Ranges
CHAPTER 6

Modifying all Samples in a Sound

6.1 HOW SOUND IS ENCODED
6.2 MANIPULATING SOUNDS
6.3 CHANGING THE VOLUME OF SOUNDS
6.4 NORMALIZING SOUNDS

Chapter Learning Objectives

The media learning goals for this chapter are:

- To understand how we digitize sounds, and what the limitations are of human hearing that allow us to digitize sounds.
- To use the Nyquist theorem to determine the sampling rate necessary for digitizing a desired sound.
- To manipulate the volume of a sound.
- To create (and avoid) clipping.

The computer science goals for this chapter are:

- To understand and use a one-dimensional array as a data structure.
- To use the formula that \( n \) bits result in \( 2^n \) possible patterns in order to figure out the number of bits needed to save values.
- To debug sound programs (methods).
- To use iteration (with \texttt{while} and \texttt{for} loops) for manipulating sounds.
- To use conditionals when manipulating sounds.
- To use scope to understand when a variable name is understood.

6.1 HOW SOUND IS ENCODED

There are two parts to understanding how sound is encoded and manipulated.

- First, what are the physics of sound? How is it that we hear a variety of sounds?
- Next, how can we then map these sounds into the numbers of a computer?
6.1.1 The Physics of Sound

Physically, sounds are waves of air pressure. When something makes a sound, it makes ripples in the air just like stones or raindrops dropped into a pond cause ripples in the surface of the water (Figure 6.1). Each drop causes a wave of pressure to pass over the surface of the water, which causes visible rises in the water, and less visible but just as large depressions in the water. The rises are increases in pressure and the lows are decreases in pressure. Some of the ripples we see are actually ones that arise from combinations of ripples—some waves are the sums and interactions from other waves.

FIGURE 6.1: Raindrops causing ripples in the surface of the water, just as sound causes ripples in the air

In the air, we call these increases in pressure compressions and decreases in pressure rarefactions. It’s these compressions and rarefactions that lead to our hearing. The shape of the waves, their frequency, and their amplitude all impact how we perceive the sound.

The simplest sound in the world is a sine wave (Figure 6.2). In a sine wave, the compressions and rarefactions arrive with equal size and regularity. In a sine wave, one compression plus one rarefaction is called a cycle. At some point in the cycle, there has to be a point where there is zero pressure, just between the compression and the rarefaction. The distance from the zero point to the greatest pressure (or least pressure) is called the amplitude.

Formally, amplitude is measured in Newtons per meter-squared ($N/m^2$). That’s a rather hard unit to understand in terms of perception, but you can get a sense of the amazing range of human hearing from this unit. The smallest sound that humans typically hear is $0.0002 N/m^2$, and the point at which we sense the vibrations in our entire body is $200 N/m^2$! In general, amplitude is the most important factor in our perception of volume: If the amplitude rises, we perceive the sound as being louder typically. Other factors like air pressure factor into our perception of increased volume, too. Ever noticed how sounds sound differently on very humid days as compared with very dry days?

When we perceive an increase in volume, we say that we’re perceiving an increase in the intensity of sound. Intensity is measured in watts per meter-squared ($W/m^2$). (Yes, those are watts just like the ones you’re referring to when you get a 60-watt light bulb—it’s a measure of power.) The intensity is proportional to
Figure 6.2: One cycle of the simplest sound, a sine wave

the square of the amplitude. For example, if the amplitude doubles, intensity quadruples.

Human perception of sound is not a direct mapping from the physical reality. The study of the human perception of sound is called psychoacoustics. One of the odd facts about psychoacoustics is that most of our perception of sound is logarithmically related to the actual phenomena. Intensity is an example of this. A change in intensity from 0.1W/m$^2$ to 0.01W/m$^2$ sounds the same to us (as in the same amount of volume change) as a change in intensity of 0.001W/m$^2$ to 0.0001W/m$^2$.

We measure the change in intensity in decibels (dB). That’s probably the unit that you most often associate with volume. A decibel is a logarithmic measure, so it does match the way we perceive volume. It’s always a ratio, a comparison of two values. $10 \times \log_{10}(I_1/I_2)$ is change in intensity in decibels between $I_1$ and $I_2$. If two amplitudes are measured under the same conditions, we can express the same definition as amplitudes: $20 \times \log_{10}(A_1/A_2)$. If $A_2 = 2 \times A_1$ (i.e., the amplitude doubles), the difference is roughly 6dB.

When decibel is used as an absolute measurement, it’s in reference to the threshold of audibility at sound pressure level (SPL): 0 dB SPL. Normal speech has intensity about 60 dB SPL. Shouted speech is about 80 dB SPL.

How often a cycle occurs is called the frequency. If a cycle is short, then there can be lots of them per second. If a cycle is long, then there are fewer of them. As the frequency increases we perceive the pitch to increase. We measure frequency in cycles per second (cps) or Hertz (Hz).

All sounds are periodic: there is always some pattern of rarefaction and compression that leads to cycles. In a sine wave, the notion of a cycle is easy. In natural waves, it’s not so clear where a pattern repeats. Even in the ripples in a pond, the waves aren’t as regular as you might think. The time between peaks in waves isn’t always the same: it varies. This means that a cycle may involve several peaks-and-valleys until it repeats.

Humans hear between 2 Hz and 20,000 Hz (or 20 kilohertz, abbreviated 20
kHz). Again, as with amplitudes, that’s an enormous range! To give you a sense of where music fits into that spectrum, the note A above middle C is 440 Hz in traditional, equal temperament tuning (Figure 6.3).

**FIGURE 6.3:** The note A above middle C is 440 Hz

Like intensity, our perception of pitch is almost exactly proportional to the log of the frequency. We really don’t perceive absolute differences in pitch, but the ratio of the frequencies. If you heard a 100 Hz sound followed by a 200 Hz sound, you’d perceive the same pitch change (or pitch interval) as a shift from 1000 Hz to 2000 Hz. Obviously, a difference of 100 Hz is a lot smaller than a change of 1000 Hz, but we perceive it to be the same.

In standard tuning, the ratio in frequency between the same notes in adjacent octaves is 2 : 1. Frequency doubles each octave. We told you earlier that A above middle C is 440 Hz. You know then that the next A up the scale is 880 Hz.

How we think about music is dependent upon our cultural standards, but there are some universals. Among these universals are the use of pitch intervals (e.g., the ratio between notes C and D remains the same in every octave), the relationship between octaves remains constant, and the existence of four to seven main pitches (not considering sharps and flats here) in an octave.

What makes the experience of one sound different from another? Why is it that a flute playing a note sounds so different than a trumpet or a clarinet playing the same note? We still don’t understand everything about psychoacoustics and what physical properties influence our perception of sound, but here are some of the factors that lead us to perceiving different sounds (especially musical instruments) as distinct.

- Real sounds are almost never single frequency sound waves. Most natural sounds have several frequencies in them, often at different amplitudes. These additional frequencies are sometimes called overtones. When a piano plays the note C, for example, part of the richness of the tone is that the notes E and G are also in the sound, but at lower amplitudes. Different instruments have different overtones in their notes. The central tone, the one we’re trying to play, is called the fundamental.

- Instrument sounds are not continuous with respect to amplitude and frequency. Some come slowly up to the target frequency and amplitude (like
wind instruments), while others hit the frequency and amplitude very quickly and then the volume fades while the frequency remains pretty constant (like a piano).

- Not all sound waves are represented well by sine waves. Real sounds have funny bumps and sharp edges. Our ears can pick these up, at least in the first few waves. We can do a reasonable job synthesizing with sine waves, but synthesizers sometimes also use other kinds of wave forms to get different kinds of sounds (Figure 6.4).

![Figure 6.4: Some synthesizers using triangular (or sawtooth) or square waves.](image)

### 6.1.2 Exploring how sounds look

On your CD, you will find the MediaTools application with documentation for how to get it started. The MediaTools application contains tools for sound, graphics, and video. Using the sound tools, you can actually observe sounds as they’re coming into your computer’s microphone to get a sense of what louder and softer sounds look like, and what higher and lower pitched sounds look like.

The basic sound editor looks like Figure 6.5. You can record sounds, open WAV files on your disk, and view the sounds in a variety of ways. (Of course, assuming that you have a microphone on your computer!)

![Figure 6.5: Sound editor main tool](image)

To view sounds, click the View button, then the Record button. (Hit the Stop button to stop recording.) There are three kinds of views that you can make of the sound.

The first is the signal view (Figure 6.6). In the signal view, you’re looking at the sound raw—each increase in air pressure results in an rise in the graph, and each decrease in sound pressure results in a drop in the graph. Note how rapidly
the wave changes! Try some softer and louder sounds so that you can see how the sounds’ look changes. You can always get back to the signal view from another view by clicking the SIGNAL button.

FIGURE 6.6: Viewing the sound signal as it comes in

The second view is the spectrum view (Figure 6.7). The spectrum view is a completely different perspective on the sound. In the previous section, you read that natural sounds are often actually composed of several different frequencies at once. The spectrum view shows these individual frequencies. This view is also called the frequency domain.

Frequencies increase in the spectrum view from left to right. The height of a column indicates the amount of energy (roughly, the volume) of that frequency in the sound. Natural sounds look like Figure 6.8 with more than one spike (rise in the graph). (The smaller rises around a spike are often seen as noise.)

The technical term for how a spectrum view is generated is called a Fourier transform. A Fourier transform takes the sound from the time domain (rises and falls in the sound over time) into the frequency domain (identifying which frequencies are in a sound, and the energy of those frequencies, over time). The specific technique being used in the MediaTools signal view is a Fast Fourier Transform (or FFT), a very common way to do Fourier transforms quickly on a computer so that we can get a real time view of the changing spectra.

The third view is the sonogram view (Figure 6.9). The sonogram view is very much like the spectrum view in that it’s describing the frequency domain, but it presents these frequencies over time. Each column in the sonogram view, sometimes called a slice or window (of time), represents all the frequencies at a given moment in time. The frequencies increase in the slice from lower (bottom) to higher (top). The darkness of the spot in the column indicates the amount of energy of that frequency in the input sound at the given moment. The sonogram view is great for studying how sounds change over time, e.g., how the sound of a piano key being struck changes as the note fades, or how different instruments differ in their sounds, or in how different vocal sounds differ.
Chapter 6 Modifying all Samples in a Sound

FIGURE 6.7: Viewing the sound in a spectrum view

Making it Work Tip: Explore sounds!
You really should try these different views on real sounds. You’ll get a much better understanding of sound and what the manipulations we’re doing in this chapter are doing to the sounds.

6.1.3 Encoding the Sound

You just read about how sounds work physically and how we perceive them. To manipulate these sounds on a computer and to play them back on a computer, we have to digitize them. To digitize sound means to take this flow of waves and turn it into numbers. We want to be able to capture a sound, perhaps manipulate it, and then play it back (through the computer’s speakers) and hear what we captured—as exactly as possible.

The first part of the process of digitizing a sound is handled by the computer’s hardware—the physical machinery of the computer. If a computer has a microphone and the appropriate sound equipment (like a SoundBlaster sound card on Wintel computers), then it’s possible, at any moment, to measure the amount of air pressure against that microphone as a single number. Positive numbers correspond to rises in pressure, and negative numbers correspond to rarefactions. We call this an analog-to-digital conversion (ADC)—we’ve moved from an analog signal (a continuously changing sound wave) to a digital value. This means that we can get an instantaneous measure of the sound pressure, but it’s only one step along the way. Sound is a continuous changing pressure wave. How do we store that in our computer?

By the way, playback systems on computers work essentially the same in reverse. The sound hardware does a digital-to-analog conversion (DAC), and the analog signal is then sent to the speakers. The DAC process also requires numbers
Section 6.1 How Sound is Encoded

FIGURE 6.8: Viewing a sound in spectrum view with multiple “spikes”

representing pressure.

If you’ve had some calculus, you’ve got some idea of how we might do that. You know that we can get close to measuring the area under a curve with more and more rectangles whose height matches the curve (Figure 6.10). With that idea, it’s pretty clear that if we capture enough of those microphone pressure readings, we capture the wave. We call each of those pressure readings a sample—we are literally “sampling” the sound at that moment. But how many samples do we need? In integral calculus, you compute the area under the curve by (conceptually) having an infinite number of rectangles. While computer memories are growing larger and larger all the time, we still can’t capture an infinite number of samples per sound.

Mathematicians and physicists wondered about these kinds of questions long before there were computers, and the answer to how many samples we need was actually computed long ago. The answer depends on the highest frequency you want to capture. Let’s say that you don’t care about any sounds higher than 8,000 Hz. The Nyquist theorem says that we would need to capture 16,000 samples per second to completely capture and define a wave whose frequency is less than 8,000 cycles per second.

Computer Science Idea: Nyquist Theorem

To capture a sound of at most $n$ cycles per second, you need to capture $2n$ samples per second.

This isn’t just a theoretical result. The Nyquist theorem influences applications in our daily life. It turns out that human voices don’t typically get over 4,000 Hz. That’s why our telephone system is designed around capturing 8,000 samples per second. That’s why playing music through the telephone doesn’t really work...
very well. The limits of (most) human hearing is around 22,000 Hz. If we were to capture 44,000 samples per second, we would be able to capture any sound that we could actually hear. CD’s are created by capturing sound at 44,100 samples per second—just a little bit more than 44 kHz for technical reasons and for a fudge factor.

We call the rate at which samples are collected the **sampling rate**. Most sounds that we hear in daily life are well within the range of the limits of our hearing. You can capture and manipulate sounds in this class at a sampling rate of 22 kHz (22,000 samples per second), and it will sound quite reasonable. If you use too low of a sampling rate to capture a high-pitched sound, you’ll still hear something when you play the sound back, but the pitch will sound strange.

Typically, each of these samples are encoded in two bytes (16 bits). Though there are larger **sample sizes**, 16 bits works perfectly well for most applications. CD-quality sound uses 16 bit samples.
In 16 bits, the numbers that can be encoded range from -32,768 to 32,767. These aren’t magic numbers—they make perfect sense when you understand the encoding. These numbers are encoded in 16 bits using a technique called two’s complement notation, but we can understand it without knowing the details of that technique. We’ve got 16 bits to represent positive and negative numbers. Let’s set aside one of those bits (remember, it’s just 0 or 1) to represent whether we’re talking about a positive (0) or negative (1) number. We call that the sign bit. That leaves 15 bits to represent the actual value. How many different patterns of 15 bits are there? We could start counting:

\[
\begin{align*}
000000000000000 & \\
000000000000001 & \\
000000000000010 & \\
000000000000011 & \\
\vdots & \\
111111111111110 & \\
111111111111111 &
\end{align*}
\]

That looks forbidding. Let’s see if we can figure out a pattern. If we’ve got two bits, there are four patterns: 00, 01, 10, 11. If we’ve got three bits, there are eight patterns: 000, 001, 010, 011, 100, 101, 110, 111. It turns out that \(2^2\) is four, and \(2^3\) is eight. Play with four bits. How many patterns are there? \(2^4 = 16\) It turns out that we can state this as a general principle.

Computer Science Idea: \(2^n\) patterns in \(n\) bits
If you have \(n\) bits, there are \(2^n\) possible patterns in those \(n\) bits.

\(2^{15} = 32,768\). Why is there one more value in the negative range than the positive? Zero is neither negative nor positive, but if we want to represent it as bits, we need to define some pattern as zero. We use one of the positive range values (where the sign bit is zero) to represent zero, so that takes up one of the 32,768 patterns.

The sample size is a limitation on the amplitude of the sound that can be captured. If you have a sound that generates a pressure greater than 32,767 (or a rarefaction greater than -32,768), you’ll only capture up to the limits of the 16 bits. If you were to look at the wave in the signal view, it would look like somebody took some scissors and clipped off the peaks of the waves. We call that effect clipping for that very reason. If you play (or generate) a sound that’s clipped, it sounds bad—it sounds like your speakers are breaking.

There are other ways of digitizing sound, but this is by far the most common. The technical term for this way of encoding sound is pulse coded modulation (PCM). You may encounter that term if you read further in audio or play with audio software.

What this means is that a sound in a computer is a long list of numbers, each of which is a sample in time. There is an ordering in these samples: If you played the samples out of order, you wouldn’t get the same sound at all. The most efficient
way to store an ordered list of data items on a computer is with an array. An array is literally a sequence of bytes right next to one another in memory. We call each value in an array an element.

We can easily store the samples that make up a sound in an array. Think of each two bytes as storing a single sample. The array will be large—for CD-quality sounds, there will be 44,100 elements for every second of recording. A minute long recording will result in an array with 26,460,000 elements.

Each array element has a number associated with it, called its index. The index numbers start at 0 and increase sequentially. The first one is 0, the second one is 1, and so on. It may sound strange to say the the index for the first array element is 0 but this is basically a measure of the distance from the first element in the array. Since the distance from the first element to itself is 0, the index is 0. You can think about an array as a long line of boxes, each one holding a value and each box having an index number on it (Figure 6.11).

FIGURE 6.11: A depiction of the first five elements in a real sound array

Using the MediaTools, you can graph a sound file (Figure 6.12) and get a sense of where the sound is quiet (small amplitudes), and loud (large amplitudes). This is actually important if you want to manipulate the sound. For example, the gaps between recorded words tend to be quiet—at least quieter than the words themselves. You can pick out where words end by looking for these gaps, as in Figure 6.12.

FIGURE 6.12: A sound recording graphed in the MediaTools

You will soon read about how to read a file containing a recording of a sound into a sound object, view the samples in that sound, and change the values of the sound array elements. By changing the values in the array, you change the sound. Manipulating a sound is simply a matter of manipulating elements in an array.

6.2 MANIPULATING SOUNDS

Now that we know how sounds are encoded, we can manipulate sounds using Java programs. Here’s what we’ll need to do.
1. We’ll need to get a filename of a WAV file, and make a Sound object from it. You already saw how to do that in a previous chapter.

2. You will often get the samples of the sound as an array. Sample objects are easy to manipulate, and they know that when you change them, they should automatically change the original sound. You’ll read first about manipulating the samples to start with, then about how to manipulate the sound samples from within the sound itself.

3. Whether you get the sample objects out of a sound, or just deal with the samples in the sound object, you will then want to do something to the value at the sample.

4. You may then want to write the sound back out to a new file, to use elsewhere. (Most sound editing programs know how to deal with audio.)

### 6.2.1 Open sounds and manipulating samples

You have already seen how to pick a file with FileChooser.pickAFile() and then make a Sound object (object of the class Sound) with new Sound(fileName). Here’s an example of doing that in DrJava.

```java
> String fileName = FileChooser.pickAFile();
> Sound sound = new Sound(fileName);
> System.out.println(sound);
Sound file: preamble.wav number of samples: 421110
```

What new Sound(fileName) does is to scoop up all the bytes from the file name provided as input, dump them into memory, and place a big sign on them saying, “This is a sound object (an object of the class Sound)!” When you execute Sound sound = new Sound(fileName), you are saying, “Create an object variable called sound that refers to the Sound object created from the information in the file with a file name given by the variable fileName.” When you use sound as input to functions, you are saying “Use that sound object over there (yeah, the one referred to by the variable sound) as input to this function.”

You can get the samples from a sound using getSamples(). The method getSamples() must be invoked on a Sound object and returns an array of all the samples as SoundSample objects (objects of the class SoundSample). When you execute this function, it may take quite a while before it finishes—longer for longer sounds, shorter for shorter sounds.

```java
> SoundSample[] sampleArray = sound.getSamples();
> System.out.println(sampleArray.length);
421110
```

The method getSamples() is making an array of SoundSample objects out of the samples in the Sound object. An object is more than just a simple value—for example, a SoundSample object knows what Sound object it came from and what its index is. You will read more about objects later, but take it at face value now that getSamples() provides you with a bunch of sample objects that you can manipulate—and in fact, makes manipulation pretty easy. You can get the
value of a SoundSample object by using getValue(), and you set the value of a SoundSample object with setValue(value).

But before we get to the manipulations, let’s look at some other ways to get and set samples. We can ask the sound to give us the value of a specific sample at a specific index, by using the method getSampleValueAt(index) on a Sound object.

> System.out.println(sound.getSampleValueAt(0));
36
> System.out.println(sound.getSampleValueAt(1));
29

What numbers can we use as index values? Anything between 0 and the number of samples minus 1. We can get the number of samples using getLength().

Notice the error that we get below if we try to get a sample past the end of the array.

> System.out.println(sound.getLength());
421110
> sound.getSampleValueAt(500000);
You are trying to access the sample at index: 500000, but the last valid index is at 421109

We can similarly change sample values in a Sound object with setSampleValueAt(index). This method changes the value of the sample at the passed index. We can then check it again with getSampleValueAt().

> System.out.println(sound.getSampleValueAt(0));
36
> sound.setSampleValueAt(0,12);
> System.out.println(sound.getSampleValueAt(0));
12

Common Bug: Mistyping a name
You just saw a whole bunch of method names, and some of them are pretty long. What happens if you type one of them wrong? DrJava will complain that it doesn’t know what you mean, like this:

> sound.setSampleValueAt(0,12);
Error: No 'setSampleValueAt' method in 'Sound'

It’s no big deal. Use the up arrow on the keyboard to bring up the last thing you typed into DrJava and then use the left arrow to get to the place with the error and then fix it. You can use the up arrow to get to any of the commands you have typed in the interactions pane since you started DrJava.

What do you think would happen if we then played this sound? Would it really sound different than it did before, now that we’ve turned the first sample from the
number 36 to the number 12? Not really. To explain why not, let’s find out what the sampling rate is for this sound, by using the method `getSamplingRate()`.

```
> Sound aSound = new Sound(chooser.getMediaPath("preamble.wav"));
> System.out.println(aSound.getSamplingRate());
22050.0
```

The sound that we’re manipulating in this example (a recording of Mark reading part of the U.S. Constitution’s preamble) has a sampling rate of 22,050 samples per second. Changing one sample changes $1/22050$ of the first second of that sound. If you can hear that, you have amazingly good hearing—and I will have my doubts about your truthfulness!

Obviously, to make a significant manipulation to the sound, we have to manipulate hundreds if not thousands of samples. We’re certainly not going to do that by typing thousands of lines like this:

```
> aSound.setSampleValueAt(0, 12);
> aSound.setSampleValueAt(1, 24);
> aSound.setSampleValueAt(2, 100);
> aSound.setSampleValueAt(3, 99);
> aSound.setSampleValueAt(4, -1);
```

We need to take advantage of the computer executing our program, by telling it to go do something hundreds or thousands of times. That’s the topic for the next section.

But we will end this section by talking about how to write your results back out to a file. Once you’ve manipulated your sound and want to save it out to use elsewhere, you use `write(String fileName)` which takes a filename as input. Be sure that your file ends with the extension “.wav” if you’re saving a sound so that your operating system knows what to do with it (what type of data is in it)!

```
> Sound aSound = new Sound(chooser.pickAFile());
> System.out.println(aSound.getFileName());
c:\intro-prog-java\mediasources\preamble.wav
> aSound.write("c:\\intro-prog-java\\mediasources\\preamble.wav");
```
Chapter 6 Modifying all Samples in a Sound

Making it Work Tip: Slashes and Backslashes in Java
Even though the file name is output with backslashes (\) in it, a Java string that wants to have a backslash in it must double it since the backslash character is used in Java strings to indicate special characters like tab. Or, you can use a forward slash (/) instead of double backslashes.

Common Bug: Saving a file quickly—and how to find it again!
What if you don’t know the whole path to a directory of your choosing? You don’t have to specify anything more than the base name.

```
> aSound.write("new-preamble.wav")
```

The problem is finding the file again! In what directory did it get saved? This is a pretty simple bug to resolve. The default directory (the one you get if you don’t specify a path) is wherever DrJava is. You can also use `FileChooser.getMediaPath("new-preamble.wav")` to get the full name of the media directory and base file name.

You’ll probably figure out when playing sounds a lot that if you use `play()` a couple times in quick succession, you’ll mix the sounds. How do you make sure that the computer plays only a single sound and then waits for that sound to end? You use something called `blockingPlay()`. That works the same as `play()`, but it waits for the sound to end so that no other sound can interfere while it’s playing.

6.2.2 Using MediaTools for looking at captured sounds

The MediaTools for manipulating sounds that you read about earlier can also be used to study sound files. Any WAV file on your computer can be opened and studied within the sound tools.

Using the MediaTools application.
From the basic sound editor tool in the MediaTools application, click on File to get the option to open a WAV file (Figure 6.13). The MediaTools’ open file dialog will then appear. Find a WAV file by clicking on the directories on the left until you find one that contains the WAV files you want on the right (Figure 6.14), then click OK.

Your CD contains a `mediasources` directory on it. Most of the examples in the book use the media in this directory. You’ll probably want to drag the `mediasources` folder onto your hard disk so that you can use it there. The default media directory is `c:/intro-prog-java/mediasources` so that would be a good
place to put it. If you put it somewhere else just use 
FileChooser.setMediaPath(directory) to set the directory to use to a different directory.

![Sound Editor Menu](image1)

**FIGURE 6.13:** The sound editor open menu in MediaTools application

![Open File Dialog](image2)

**FIGURE 6.14:** MediaTools application open file dialog

You will then be shown the file in the sound editor view (Figure 6.15). The sound editor lets you explore a sound in many ways (Figure 6.16). As you scroll through the sound and change the sound cursor (the red/blue line in the graph) position, the index changes to show you which sound array element you’re currently looking at, and the value shows you the value at that index. You can also fit the whole sound into the graph to get an overall view (but currently breaks the index/value displays). You can even “play” your recorded sound as if it were an instrument—try pressing the piano keys across the bottom of the editor. You can also set the cursor (via the scrollbar or by dragging in the graph window) and play the sound before the cursor—a good way to hear what part of the sound corresponds to what index positions. Clicking the < > button provides a menu of options which includes getting an FFT view of the sound.

### 6.2.3 Introducing the loop

The problem of wanting to do something similar a great many times is a common one in computing: How do we get the computer to do something over-and-over
again? We need to get the computer to loop or iterate. Java has commands especially for looping (iterating).

Starting with Java version 1.5 there is a new way of looping through all members of an array using a \texttt{for each} loop. The syntax is \texttt{for (Type variableName : array)}. You can read this as, "for each element in the array execute the body of the loop." The first time through the loop the \texttt{variableName} will refer to the first element of the array (the one at index 0). The second time through the loop the \texttt{variableName} will refer to the second element of the array (the one at index 1). The last time through the loop the \texttt{variableName} will refer to the last element of the array (the one at index (length - 1)). The code to loop through all the \texttt{SoundSample} objects in an array of \texttt{SoundSample} objects and set the value of each sample to its original value (no change) is below:

```java
public void doNothing ()
{
    SoundSample [] sampleArray = this.getSamples ();
    int value = 0 ;

    // loop through all the samples in the array
    for (SoundSample sample : sampleArray)
    {
```
Section 6.2  Manipulating sounds  217

value = sample.getValue();
sample.setValue(value);
}
}

As you can see the for each loop makes it easy to loop through all the elements of an array. You can use the for each loop whenever you want to process all the elements of an array and you don’t need to know the current index in the body of the loop.

However, not everyone is using Java 1.5 yet. If you are not using Java 1.5 you may want to start with a while loop. A while loop executes some commands while a test returns true. In order for the loop to stop there but be some way for the test to end up false.

The way that we will manipulate a sound is to change the values in the samples that make up the sound. We want to loop through all the samples in a sound and do something to each value. One way to do that is to loop through all the elements of the array of samples. We are going to use the getSamples() function we saw earlier to provide our array. We will use a while loop that looks like this:

SoundSample[] sampleArray = this.getSamples();
SoundSample sample = null;
int index = 0;

// loop through all the samples in the array
while (index < sampleArray.length)
{
  sample = sampleArray[index];
  index++;
}

Let’s talk through the pieces here.

• First comes the declaration of an array of SoundSample objects called sampleArray. The value of this variable is a reference to the array returned from this.getSamples(). The this could have been left off and the compiler would have added it for you. The this refers to the implicitly passed current object (which is a Sound object.

• Next is the declaration of an object variable sample that will refer to a SoundSample object. Since it currently doesn’t refer to any object the value is set to null.

• A primitive variable index is declared and its value is set to 0.

• The while (index < sampleArray.length) checks if the value stored in the variable index is less than the length of the sampleArray array. If it is the body of the loop is executed. The body of the loop can be a single following statement which is usually indented to show that it is the body of the loop or a block of statements. A block of statements is enclosed in an open curly brace '{' and a closing curly brace '}'.


Chapter 6  Modifying all Samples in a Sound

- In the body of the loop we set the value of the object variable `sample` to refer to the `SoundSample` object at the current value of `index` in the array of sound samples. Since index starts at 0 this will return the first sample the first time through the loop, the second one the second time through the loop, the third one the third time through the loop, and so on.

- The `index++;` means the same as `index = index + 1;` and will add one to the current value of `index`. Of course it doesn’t that the value of index is equal to the value of index plus 1 it means set the value of index to the value of index plus 1.

- Next execution will jump back to the `while (index < sampleArray.length)` and if the `index` is still less than the length of the array the body of the loop will be executed again. Eventually the value of `index` will be equal to the length of the array and execution will continue at the next statement after the body of the loop.

For example, here is the `while` loop that simply sets each sample to its own value (a particularly useless exercise, but it’ll get more interesting in just a couple pages).

```java
public void doNothing() {
    SoundSample [] sampleArray = this.getSamples();
    SoundSample sample = null;
    int index = 0;
    int value = 0;

    // loop through all the samples in the array
    while (index < sampleArray.length) {
        sample = sampleArray[index];
        value = sample.getValue();
        sample.setValue(value);
        index++;
    }
}
```

Let’s talk through this code.

- The first statement gets the array of `SoundSample` objects from the current `Sound` object using the method `getSamples()` and declares a variable `sampleArray` which refers to it.

- The next statement declares a variable `sample` that can refer to a `SoundSample` object but is set to `null` to show that it doesn’t reference any object yet.

- Next a primitive variable `index` is declared and initialized to 0.

- The primitive variable `value` is declared and initialized to 0.
Section 6.2 Manipulating sounds

- The code `while (index < sampleArray.length)` tests if the value of `index` is less than the length of the array of `SoundSample` objects. If it is, the body of the loop will be executed. If not, execution will continue with the first statement following the body of the loop.

- The first statement in the body of the loop sets the variable `sample` to refer to the `SoundSample` object at the value of `index` in the array `soundArray`. Since `index` starts off with a value of 0 this will refer to the first `SoundSample` object in the array the first time through the loop.

- The space reserved for the variable `value` is set to the value of the `SoundSample` object referred to by `sample`.

- Next, the value of the `SoundSample` object referred to by `sample` is set to the contents of `value`. Since the contents of the variable `value` is the value for this `SoundSample` object there will be no change. It just sets the sample value to the original value.

- Finally, the value in `index` in incremented by one. Then execution will jump back to the `while` test again.

Here’s the exact same code (it would work exactly the same), but with different variable names.

```java
SoundSample[] a = this.getSamples();
SoundSample s = null;
int i = 0;
int v = 0;

// loop through all the samples in the array
while (i < a.length)
{
    s = a[i];
    v = s.getValue();
    s.setValue(v);
    i++;
}
```

What’s the difference? These are slightly easier to confuse variable names. `a` and `s` are not as obvious as to what they are naming as `sampleArray` and `sample`. Java doesn’t care which we use, and the single character variable names are clearly easier to type. But the longer variable names make it easier to understand your code. It is best to try to make your code easier for humans to understand.

You may have wondered do we need the variable `v`? We could combine the two statements into one.

```java
SoundSample[] a = this.getSamples();
SoundSample s = null;
int i = 0;

// loop through all the samples in the array
while (i < a.length)
{
    s = a[i];
    s.setValue(s.getValue());
    i++;
}
```
Chapter 6  Modifying all Samples in a Sound

Now that we see how to get the computer to do thousands of commands without writing thousands of individual lines, let’s do something useful with this.

{  
    s = a[i];  
    s.setValue(s.getValue());  
    i++;  
}

Common Bug: Keep sounds short
Longer sounds take up more memory and will process more slowly.

Common Bug: Windows and WAV files
The world of WAV files isn’t as compatible and smooth as one might like. WAV files created with other applications (such as Windows Recorder) may not play in DrJava, and DrJava WAV files may not play in all other applications (e.g., WinAmp 2). Some tools like Apple QuickTime Player Pro (http://www.apple.com/quicktime) are good at reading any WAV file and being able to export a new one that most any other application can read.

6.3  CHANGING THE VOLUME OF SOUNDS

Earlier, we said that the amplitude of a sound is the main factor in the volume. This means that if we increase the amplitude, we increase the volume. Or if we decrease the amplitude, we decrease the volume.

Don’t get confused here—changing the amplitude doesn’t reach out and twist up the volume knob on your speakers. If your speaker’s volume (or computer’s volume) is turned down, the sound will never get very loud. The point is getting the sound itself louder. Have you ever watched a movie on TV where, without changing the volume on the TV, sound becomes so low that you can hardly hear it? (Marlon Brando’s dialogue in the movie The Godfather comes to mind.) That’s what we’re doing here. We can make sounds shout or whisper by tweaking the amplitude.

6.3.1  Increasing volume

Here’s a function that doubles the amplitude of an input sound.

Program 49: Increase an input sound’s volume

/**  
  * Method to double the volume (amplitude) of the sound  
**/
Section 6.3  Changing the volume of sounds  221

```java
/**
 public void increaseVolume()
 {
  SoundSample[] sampleArray = this.getSamples();
  SoundSample sample = null;
  int value = 0;
  int index = 0;

  // loop through all the samples in the array
  while (index < sampleArray.length)
  {
    sample = sampleArray[index];
    value = sample.getValue();
    sample.setValue(value * 2);
    index++;
  }
}
```

Go ahead and type the above into your DrJava definitions pane before the last curly brace in the Sound.java class. Click Compile All to get DrJava to compile it. Follow along the example below to get a better idea of how this all works.

To use this program, you have to create a sound first and invoke this method on it. Don’t forget that you can’t type this code in and have it work as-is: Your path names may be different than what is shown here!

```
> String f = "c:/intro-prog-java/mediasources/gettysburg10.wav";
> Sound s = new Sound(f);
> s.play();
> s.explore();
> s.increaseVolume();
> s.play();
> s.explore();
```

In the interactions pane we create a variable f which refers to a String object that holds the name of a file. We create the variable s which refers to a Sound object created from the file using new Sound(f). We ask this Sound object to play using s.play(). We then open an explorer on the sound to see what it looks like graphically using s.explore(). We next increase its volume using s.increaseVolume(). This implicitly passes the Sound object to the method increaseVolume(). So the code this.getSamples() in the method increaseVolume() means to get them
from the implicitly passed `Sound` object (the one referred to by variable `s`).

**Computer Science Idea: Changing memory doesn’t change the file**

If you create another `Sound` object from the same file will you get the original sound or the sound with volume increased? You will get the original sound. The `Sound` object `s` was created by reading the file data into memory. The change to the `Sound` object was done in memory, but the file wasn’t changed. If you want to save your changes write them out to a file using the method `soundObj.write(String fileName);` where `soundObj` is the name of the `Sound` object and `fileName` is the full path name of the file. So to save the changed `Sound` object above use `s.write("gettyLouder.wav");`.

### 6.3.2 Did that really work?

Now, is it really louder, or does it just seem that way? We can check it in several ways. You could always make the sound even louder by evaluating `increaseVolume` on our sound a few more times—eventually, you’ll be totally convinced that the sound is louder. But there are ways to test even more subtle effects.

If you compare graphs of the two sounds using the sound explorer, you will find that the graph of the sound does have greater amplitude after increasing it using our function. Check it out in Figure 6.17.

![Figure 6.17: Comparing the graphs of the original sound (left) and the louder one (right)](image)

Maybe you’re unsure that you’re really seeing a larger wave in the second picture. You can use a sound explorer to check the individual sample values. You can actually already see that in Figure 6.17—see that the first value (index number 0) is 59 in the original sound and 118 in the second sound. You can also check the value at any index using the sound explorer. Just click on a location and the value will be displayed for that location. To check the same location in the second explorer just type in the desired current index and it will show the value at that index. You’ll see that the louder sound really does have double the value of the...
Section 6.3  Changing the volume of sounds

same sample in the original sound.

Finally, you can always check for yourself from within DrJava. If you’ve been following along with the example\(^1\), then the variable \(s\) is the now louder sound. \(f\) should still be the filename of the original sound. Go ahead and make a new sound object which is the original sound—that is named below as \(s\text{Orig}\) (for sound original). Check any sample that you want—it’s always true that the louder sound has twice the value than the the original sound.

\begin{verbatim}
> System.out.println(s);
Sound file: c:/intro-prog-java/mediasources/gettysburg10-louder.wav number of samples: 220568
> System.out.println(f);
Sound file: c:/intro-prog-java/mediasources/gettysburg10.wav
> Sound sOrig = new Sound(f);
> System.out.println(s.getSampleValueAt(0)); 118
> System.out.println(sOrig.getSampleValueAt(0)); 59
> System.out.println(s.getSampleValueAt(1)); 78
> System.out.println(sOrig.getSampleValueAt(1)); 39
> System.out.println(s.getSampleValueAt(999)); -80
> System.out.println(sOrig.getSampleValueAt(999)); -40
\end{verbatim}

You can see from the last value that even negative values become more negative. That’s what’s meant by “increasing the amplitude.” The amplitude of the wave goes in both directions. We have to make the wave larger in both the positive and negative dimensions.

\(^{1}\text{What? You haven’t? You should! It’ll make much more sense if you try it yourself!}\)
Chapter 6  Modifying all Samples in a Sound

It's important to do what you just read in this chapter: Doubt your programs. Did that really do what I wanted it to do? The way you check is by testing. That's what this section is about. You just saw several ways to test:

- By looking at the result overall (like with the graphs created by the explorer),
- By checking pieces of the results (like with the explorer or MediaTools), and
- By writing additional code statements that check the results of the original program.

**Figuring out how it worked.**
Let's walk through the code, slowly, and consider how this program worked.

```java
/**
 * Method to double the volume (amplitude) of the sound
 */
public void increaseVolume()
{
    SoundSample[] sampleArray = this.getSamples();
    SoundSample sample = null;
    int value = 0;
    int index = 0;

    // loop through all the samples in the array
    while (index < sampleArray.length)
    {
        sample = sampleArray[index];
        value = sample.getValue();
        sample.setValue(value * 2);
        index++;
    }
}
```

Recall our picture of the samples in a sound array.

```
<table>
<thead>
<tr>
<th>59</th>
<th>39</th>
<th>16</th>
<th>10</th>
<th>-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

...```

This is what `sound.getSamples()` would return: An array of `SoundSample` objects. The `while` loop allows us to walk through each sample, one at a time. The name (variable) `sample` will refer to each `SoundSample` object in turn.

The variable `index` starts out with a value of 0. This is less than the length of the array `sampleArray` so the body of the loop is executed. The variable `sample` is changed to refer to the first `SoundSample` object (the one at index 0).
Section 6.3  Changing the volume of sounds

The variable \texttt{value} will take on the value of 59 when \texttt{value=sample.getValue()} is executed. The value stored at that \texttt{SoundSample} object will be set to \texttt{value} times 2 ($59 \times 2 = 118$).

The value in variable \texttt{index} will be incremented by 1 ($0 + 1 = 1$). That’s the end of the first pass through the body of the \texttt{while} loop. The loop will then start over. The test that \texttt{index} is less than the length of the array of samples will happen again. Since, it is still less the body of the loop will be executed (statements inside the open and close curly braces). The variable \texttt{sample} will be changed to refer to the second item in the array (the one at index 1).

Again, the variable \texttt{value} is set to the value of the \texttt{SoundSample} object. The value of the \texttt{SoundSample} object is set to twice the amount held in the variable \texttt{value}. 
This is what it will look like after 5 times through the loop.

But really, the while loop keeps going through all the samples—tens of thousands of them! Thank goodness it’s the computer executing this program!

What you have just read in this section is called tracing the program. We slowly went through how each step in the program was executed. We drew pictures to describe the data in the program. We used numbers, arrows, equations, and even plain English to explain what was going on in the program. This is the single most important technique in programming. It’s part of debugging. Your program will not always work. Absolutely, guaranteed, without a shadow of a doubt—you will write code that does not do what you want. But the computer will do SOMETHING. How do you figure out what it is doing? You debug, and the most significant way to do that is by tracing the program.

6.3.3 Decreasing volume

Decreasing volume, then, is the reverse of the previous process.

```
/**
 * Method to halve the volume (amplitude) of the sound.
 */
public void decreaseVolume()
{
```
Section 6.3  Changing the volume of sounds

```java
SoundSample[] sampleArray = this.getSamples();
SoundSample sample = null;
int value = 0;
int index = 0;

// loop through all the samples in the array
while (index < sampleArray.length)
{
    sample = sampleArray[index];
    value = sample.getValue();
    sample.setValue((int)(value * 0.5));
    index++;
}
```

- Our method is called on a Sound object. The Sound object is implicitly passed to the method and is accessed using the keyword this. You can leave off the this on this.getSamples() since it is understood to be invoked on the current object.

- The variable sample will refer to a different SoundSample object each time through the loop.

- Each time sample refers to a new SoundSample object, we will get the value of that SoundSample object. We put that in the variable value.

- We then set the value held by the SoundSample object to 50% of its current value, by multiplying value by 0.5, and setting the sample value to that. However, because the value is an integer and the result of a computation with a floating point value (0.5) is a floating point number we must cast to integer using (int) (value * 0.5) to let the compiler know we realize that we will be throwing away the fractional part.

We can use it like this.

```java
> String f = FileChooser.pickAFile();
> System.out.println(f);
C:\intro-prog-java\mediasources\gettysburg10-louder.wav
> Sound sound = new Sound(f);
> System.out.println(sound);
Sound file:
C:\intro-prog-java\mediasources\gettysburg10-louder.wav number of samples: 220568
> sound.play();
> sound.decreaseVolume();
> sound.play();
```

We can even do it again, and lower the volume even further.

```java
> sound.decreaseVolume();
> sound.play();
```
6.3.4 Using a for loop

Have you ever forgotten to declare the variable `index`? If you did the method wouldn’t compile. Did you ever forget to increment the the variable `index`? If you did the loop would never end until you hit Reset. Because of these problems programmers typically use a for loop instead of a while loop when they want to execute a block of commands a set number of times. A for loop is equivalent to a while loop (means the same thing to the computer). For loops are just less error prone for a programmer (though they are harder for a beginner to understand).

A for loop looks like this: `for (initialization; test; change)`. The initialization area lets you declare and initialize variables for use in the loop, the test is where you test if the loop should continue, and the change area is where you change the value of counters or indices used in the loop. For example, see the following new version of the method `decreaseVolume` which has been modified to use a for loop instead of a while loop.

```
Program 51: Decrease an input sound’s volume using a for loop

/**
 * Method to halve the volume (amplitude) of the sound.
 */
public void decreaseVolume2()
{
    SoundSample[] sampleArray = this.getSamples();
    SoundSample sample = null;
    int value = 0;
    // int index = 0;
    // loop through all the samples in the array
    // while (index < sampleArray.length)
    for (int index = 0; index < sampleArray.length; index++)
    {
        sample = sampleArray[index];
        value = sample.getValue();
        sample.setValue((int)(value * 0.5));
        // index++;
    }
}
```

We have use the to-end-of-line comment ‘//’ to comment out some lines of code to show the difference between the while and for loops. Notice that what is different is that we don’t declare and initialize the index before the loop, it is done in the initialization part of the for statement. We also don’t increment the index as the last statement in the loop. This is moved to the change area in the for statement. So we have replaced three lines of code with one and made it more likely that we will remember to declare variables for use in the loop and change them. However, what really happens during execution is the same thing as what happened during the while loop. The declarations and initializations done in the
initialization part of the \texttt{for} loop will actually take place before the first test. The change of the loop variables will actually take place after each execution of the loop body and before the next test.

### 6.3.5 Making sense of methods, in sounds

The lessons that we learned when writing picture methods (from Section 3.3.4) apply to sound methods as well. For example, we could put all the code to create a sound object (\texttt{FileChooser.pickAFile()} and \texttt{new Sound(file)}) directly into our methods like \texttt{increaseVolume()} and \texttt{decreaseVolume()}, but that means that the methods are doing more than \textit{one and only one thing}. If we had to increase or decrease the volume to a bunch of sounds, we’d find that annoying to have to keep picking files.

We can write methods that take an input value. For example, here’s a program to \texttt{changeVolume}. It accepts a \texttt{factor} that is multiplied by each sample value. This function can be used to increase or decrease the amplitude (and thus, the volume).

```
/**
 * Method to change the volume (amplitude) of the sound by multiplying the current values in the sound by the passed factor.
 * @param factor the factor to multiply by
 */
public void changeVolume(double factor)
{
    SoundSample[] sampleArray = this.getSamples();
    SoundSample sample = null;
    int value = 0;

    // loop through all the samples in the array
    for (int i = 0; i < sampleArray.length; i++)
    {
        sample = sampleArray[i];
        value = sample.getValue();
        sample.setValue((int) (value * factor));
    }
}
```

This program is clearly more flexible than \texttt{increaseVolume()} or \texttt{decreaseVolume()}. Does that make it better? Certainly it is for some purposes (e.g., if you were writing software to do general audio processing), but for other purposes, having separate and clearly named functions for increasing and decreasing volume may be better. Of course, you could modify \texttt{increaseVolume()} and \texttt{decreaseVolume()} to call \texttt{changeVolume()} with the appropriate factor. Remember that software is written for humans—write software that is understandable for the people who will be reading and using your software.
Chapter 6  Modifying all Samples in a Sound

We are reusing the name sample a lot. We have used it in several methods in the Sound class. That’s okay. Names can have different meanings depending on their context. Variables declared in a method have meaning only inside that method. Methods can even use the same variable names as other methods. You can even use the same variable names that you use in your methods in the interactions pane. This is a different context. If you create a variable in a method context (like value in Method 52 above), then that variable won’t exist when you get back out to the interactions pane. We can return values from a method context back out to the interactions pane (or a calling method) by using return, which we’ll talk more about later.

6.4 NORMALIZING SOUNDS

If you think about it some, it seems strange that the last two methods work! We can just multiply these numbers representing a sound—and the sound seems (essentially) the same to our ears just louder? The way we experience a sound depends less on the specific numbers than on the relationship between them. Remember that the overall shape of the sound waveform is dependent on many samples. In general, if we multiply all the samples by the same multiplier, we only effect our sense of volume (intensity), not the sound itself. (We’ll work to change the sound itself in future sections.)

A common operation that people want to do with sounds is to make them as LOUD AS POSSIBLE. That’s called normalizing. It’s not really hard to do, but it takes more lines of code than we’ve used previously and a few more variables, but we can do it. Here’s the algorithm, in English, that we need to tell the computer to do.

• We have to figure out what the largest sample in the sound is. If it’s already at the maximum value (the allowed range is -32768 to 32767 so the maximum allowed positive value is 32767), then we can’t really increase the volume and still get what seems like the same sound. Remember that we have to multiply all the samples by the same multiplier.

It’s an easy (algorithm) to find the largest value—sort of a sub-program within the overall normalizing program. Define a name (say, largest) and assign it a small value (0 works). Now, check all the samples. If you find a sample with an absolute value larger than the largest, save that as the value for largest. Keep checking the samples, comparing to the new largest. Eventually, the very largest value in the array will be in the variable largest.

To do this, we’ll need a way of figuring out the maximum value of two values. We can use an if (value > largest) to check if the current value is greater than the current largest and if so set that value to the largest. We can also save the index of that value so that we can check it with the sound explorer.

• Next, we need to figure out what value to multiply all the samples by. We want the largest value to become 32767. Thus, we want to figure out a multiplier such that

\[(\text{multiplier})(\text{largest}) = 32767.\]
Solve for the multiplier:

\[
\text{multiplier} = \frac{32767}{\text{largest}}.
\]

The multiplier will need to be a floating point number (have a decimal component), so we need to convince Java that not everything here is an integer. Turns out that that’s easy—use 32767.0. Simply stick on “.0”.

- Now, loop through all the samples, as we did for `increaseVolume`, and multiply the sample by the multiplier.

Here’s a program to normalize sounds.

```java
/**
 * Method to normalize (make as loud as possible) a sound.
 */
public void normalize()
{
    int largest = 0;
    int maxIndex = 0;
    SoundSample[] sampleArray = getSamples();
    SoundSample sample = null;
    int value = 0;

    // loop comparing the absolute value of the current value
    // to the current largest
    for (int i = 0; i < sampleArray.length; i++)
    {
        sample = sampleArray[i];
        value = Math.abs(sample.getValue());
        if (value > largest)
        {
            largest = value;
            maxIndex = i;
        }
    }

    // now calculate the multiplier to multiply by
    double multiplier = 32767.0 / largest;

    // print out the largest value and the multiplier
    System.out.println("The largest value was " + largest + " at index " + maxIndex);
    System.out.println("The multiplier is " + multiplier);

    // loop through all the samples and multiply by the
    // multiplier
    for (int i = 0; i < sampleArray.length; i++)
    {
```
There are several notational items to note about this program.

- There is more than one loop in this method. That is okay. I can even use the same variable \( i \) in each loop. Since the variable is declared in the initialization area of the \textit{for} loop it is only known in that loop. So, when we declare \( i \) again in the second loop it is seen as a new variable and not an attempt to declare the same variable more than once.

- There are \texttt{System.out.println()} statements in there! These statements can be \textit{really} useful. First, they give you some feedback that the program is running—a useful thing in long-running programs. Second, they show you what it’s finding, which can be interesting. Third, it’s a terrific testing method and a way to debug your programs. Let’s imagine that the printout showed that the multiplier was less than 1.0. We know that that kind of multiplier \textit{decreases} volume. You should probably suspect that something went wrong.

Here’s how to run this program:

\begin{verbatim}
> Sound s = new Sound(FileChooser.getMediaPath("preamble.wav"));
> s.explore();
> s.normalize();
The largest value was 10216 at index 179377
The multiplier is 3.207419733750979
> s.play();
> s.explore();
\end{verbatim}

Exciting, huh? Obviously, the interesting part is hearing the much louder volume, which is awfully hard to do in a book. So please try it now if you haven’t yet. But, you can see from Figure 6.19 that the values have increased.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure619.png}
\caption{Comparing the original sound with the normalized one}
\end{figure}
6.4.1 Generating clipping

Earlier, we talked about clipping, what happens when the normal curves of the sound are broken by the limitations of the sample size. One way of generating clipping is to keep increasing the volume. Another way is to explicitly force clipping.

What if you only had the largest and smallest possible sample values? What if all positive values (including zero), were the maximum value (32767) and all negative values were the minimum value (-32768)? Try this program, particularly on sounds with words in them.

Program 54: Set all samples to extreme values

```java
/**
 * Method to set all the sample values to the
 * maximum positive value if they were positive
 * (including 0) and the minimum negative
 * value if they were negative.
 */
public void forceToExtremes()
{
  SoundSample[] sampleArray = getSamples();
  SoundSample sample = null;

  // loop through the sample values
  for (int i = 0; i < sampleArray.length; i++)
  {
    // get the current sample
    sample = sampleArray[i];

    // if the value was positive (or zero) set it to the
    // maximum positive value
    if (sample.getValue() >= 0)
      sample.setValue(32767);

    // else force to max negative value
    else
      sample.setValue(-32768);
  }
}
```

Here’s how to run this program:

```java
> Sound s = new Sound(FileChooser.getMediaPath("preamble.wav"));
> s.play();
> s.explore();
> s.forceToExtremes();
> s.play();
> s.explore();
```
Look at Figure 6.20 and see that all the values have been set to extremes. When you play the sound back, you’ll hear a bunch of awful noises. That’s clipping. The really amazing thing is that you can *still* make out the words in sounds that you manipulate with this method. Our ability to decipher words from noise is incredibly powerful.

FIGURE 6.20: Comparing the original sound with one with all values set to extremes.

**OBJECTS AND METHODS SUMMARY**

In this chapter, we talk about several kinds of encodings of data (or objects).

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sound</td>
<td>Sound objects are encodings of sounds, typically coming from a WAV file.</td>
</tr>
<tr>
<td>SoundSample</td>
<td>An object that represents a sample in a sound. The value of a sample is</td>
</tr>
<tr>
<td></td>
<td>between -32768 and 32767 representing the voltage that a microphone would</td>
</tr>
<tr>
<td></td>
<td>generate at a given instant when recording a sound. The length of the</td>
</tr>
<tr>
<td></td>
<td>instant is typically either 1/44, 1000 of a second (for CD-quality sound)</td>
</tr>
<tr>
<td></td>
<td>or 1/22, 050 of a second (for good enough sound on most computers). A Sound-</td>
</tr>
<tr>
<td></td>
<td>Sample object remembers what sound it came from, so if you change its value,</td>
</tr>
<tr>
<td></td>
<td>it knows to go back and change the right sample value in the sound.</td>
</tr>
</tbody>
</table>

Here are the functions used or introduced in this chapter:

**Math methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Math.abs(number)</td>
<td>Takes a number and returns it as a positive number.</td>
</tr>
</tbody>
</table>

**FileChooser methods**
Section 6.4 Normalizing sounds 235

FileChooser.getMediaPath(String fileName) Takes a filename as input, and returns the full path name of the file with the media directory before the filename.

FileChooser.pickAFile() Lets the user pick a file and returns the complete path name as a string.

FileChooser.setMediaPath(String directory) Takes a directory as input and sets the directory name to be what is added to the passed filename using getMediaPath.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileChooser.getMediaPath(String fileName)</td>
<td>Takes a filename as input, and returns the full path name of the file with the media directory before the filename.</td>
</tr>
<tr>
<td>FileChooser.pickAFile()</td>
<td>Lets the user pick a file and returns the complete path name as a string.</td>
</tr>
<tr>
<td>FileChooser.setMediaPath(String directory)</td>
<td>Takes a directory as input and sets the directory name to be what is added to the passed filename using getMediaPath.</td>
</tr>
</tbody>
</table>

Sound methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>blockingPlay()</td>
<td>Plays the Sound object it is invoked on, and makes sure that no other sound plays at the exact same time. (Compare two blockingPlay's with two play's right after each other.)</td>
</tr>
<tr>
<td>getLength()</td>
<td>Returns the number of samples in the Sound object it is invoked on.</td>
</tr>
<tr>
<td>getSamples()</td>
<td>Returns an array of SoundSample objects for the Sound object it is invoked on.</td>
</tr>
<tr>
<td>getSampleValueAt(int index)</td>
<td>Takes an index (an integer value), and returns the value of the sample at that index for the Sound object it is invoked on.</td>
</tr>
<tr>
<td>getSamplingRate()</td>
<td>Returns the number representing the number of samples in each second for the Sound object it is invoked on.</td>
</tr>
<tr>
<td>play()</td>
<td>Plays the Sound object it is invoked on.</td>
</tr>
<tr>
<td>setSampleValueAt(int index, int value)</td>
<td>Takes an index, and a value, and sets the value of the sample at the given index in the Sound object it was invoked on to the given value.</td>
</tr>
<tr>
<td>write(String fileName)</td>
<td>Takes a filename (a string) and writes the sound in the Sound object it is invoked on to that file as a WAV file. (Make sure that the filename ends in &quot;.wav&quot; if you want the operating system to treat it right.)</td>
</tr>
</tbody>
</table>

SoundSample methods
### Chapter 6: Modifying all Samples in a Sound

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getValue()</td>
<td>Returns the value for the <code>SoundSample</code> object it is invoked on.</td>
</tr>
<tr>
<td>setValue(int value)</td>
<td>Sets the value for the <code>SoundSample</code> object it is invoked on to be the passed value.</td>
</tr>
</tbody>
</table>

#### PROBLEMS

6.1. Open up the Sonogram view and say some vowel sounds. Is there a distinctive pattern? Do “Oh’s” always sound the same? Do “Ah’s”? Does it matter if you switch speakers—are the patterns the same?

6.2. Get a couple of different instruments and play the same note on them into MediaTool application’s sound editor with the sonogram view open. Are all “C’s” made equal? Can you see some of why one sound is different than another?

6.3. Try out a variety of WAV files as instruments, using the piano keyboard in the MediaTools application sound editor. What kinds of recordings work best as instruments?

6.4. The increase volume Method 49 (page 220) uses a `while` loop. Convert it to a `for` loop.

6.5. The decrease volume Method 50 (page 226) uses a `while` loop. Convert it to a `for` loop.

6.6. In section 6.3.2, we walked through how Program 49 (page 220) worked. Draw the pictures to show how Program 50 (page 226) works, in the same way.

6.7. What happens if you increase a volume too far? Explore that by creating a `Sound` object, then increase the volume once, and again, and again. Does it always keep getting louder? Or does something else happen? Can you explain why?

6.8. Instead of multiplying samples by a multiplier (like 2 or 0.5), try adding a value to them. What happens to a sound if you add 100 to every sample? 1000?

6.9. Try sprinkling in some specific values into your sounds. What happens if you set the value of a few hundred samples in the middle of a sound to 32767? Or a few hundred -32768? Or a bunch of zeroes? What happens to the sound?

#### TO DIG DEEPER

There are many wonderful books on psychoacoustics and computer music. One of my favorites for understandability is *Computer Music: Synthesis, Composition, and Performance* by Dodge and Jerse [7]. The *bible* of computer music is Curtis Roads’ massive *The Computer Music Tutorial* [21].
CHAPTER 7

Modifying Samples using Ranges

7.1 MANIPULATING DIFFERENT SECTIONS OF THE SOUND DIFFERENTLY
7.2 CREATE A SOUND CLIP
7.3 SPLICING SOUNDS
7.4 REVERSING A SOUND
7.5 MIRRORING A SOUND

Chapter Learning Objectives
The media learning goals for this chapter are:

- To manipulate parts of a sound differently.
- To create a sound clip.
- To splice sounds together to make sound compositions.
- To reverse sounds.
- To mirror sounds.

The computer science goals for this chapter are:

- To use more than one loop in a method.
- To use ranges in iteration.
- To change more than one variable in a loop.
- To identify some algorithms that cross media boundaries.

7.1 MANIPULATING DIFFERENT SECTIONS OF THE SOUND DIFFERENTLY

Manipulating all of the samples in a sound in the same way can be useful, but really interesting effects come from chopping up sounds and manipulating them differentially: Some words this way, other sounds that way. How would you do that? We need to be able to loop through portions of the sample, without walking through the whole thing. This turns out to be an easy thing to do, but we need to manipulate samples somewhat differently (e.g., we have to use our for loop in a slightly different way).

Recall that each sample has an index number, and that we can get each individual sample value with getSampleValue(int index) (with an index number
as input). We can set any sample with `setSampleValue(int index, int value)` (with inputs of an index number, and a new value). That’s how we can manipulate samples without using `getSamples()` and `SoundSample` objects. But we still don’t want to have to write code like:

```java
sound.setSampleValue(0,12);
sound.setSampleValue(1,28);
```

Not for tens of thousands of samples! So, we will continue to use a `for` loop. However, if we are not processing the entire sound in the same way the index value that we start at won’t necessarily be 0 and the last index value won’t necessarily be the length of the sound minus 1.

What if we want to `increase` the sound for the first half of the sound, then `decrease` it in the second half. How could we do that? First we will need to calculate the half-way point. We can determine that by dividing the length of the sound by 2. Since the length and 2 are both integers the result will also be an integer so no casting is needed (any values after the decimal point will be thrown away). We will need two loops. One loop will start at the beginning of sound (0) and loop till the half-way point. The second loop will start at the half-way point and loop to the end of the sound (length - 1).

**Program 55: Increase the volume then decrease**

```java
/**
 * Method to increase the first half of the sound (double it) and then decrease the second half (half it).
 */
public void increaseAndDecrease()
{
    int half = this.getLength() / 2;
    int value = 0;

    // loop through the first half of the sound
    for (int i = 0; i < half; i++)
    {
        // get the current value
        value = this.getSampleValueAt(i);

        // set the value to 2x the original
        this.setSampleValueAt(i, value * 2);
    }

    // loop through the second half of the sound
    for (int i = half; i < this.getLength(); i++)
    {
        // get the current value
        value = this.getSampleValueAt(i);
```
Section 7.2  Create a Sound Clip  239

// set the value to half the original
this.setSampleValueAt(i,(int) ( value * 0.5));
}
}

There are two loops in increaseAndDecrease(), each of which deals with one half of the sound.

- The first loop deals with the samples from 0 to halfway through the sound. Those samples all get multiplied by 2, to double their amplitude.
- The second loop goes from halfway through to the end of the sound. Here, we multiply each sample by 0.5 thus decreasing the sound by 50%.

Another way of writing array references.

It’s worth pointing out that, in many languages, square brackets ([ ]) are standard notation for manipulating arrays. It works that way in Java. For any array, array[index] returns the index-th element in the array. The number inside the square brackets is always an index variable, but it’s sometimes referred to as a subscript, because of the way that mathematicians refer to the i-th element of a, e.g., a_i.

Let’s demonstrate how to get the elements at various index values.

> Sound sound = new Sound(FileChooser.getMediaPath("croak.wav"));
> SoundSample[] sampleArray = sound.getSamples();
> System.out.println(sampleArray[0]);
Sample at index 0 has value 0
> System.out.println(sampleArray[1]);
Sample at index 1 has value 0
> System.out.println(sampleArray[8000]);
Sample at index 8000 has value 512

7.2 CREATE A SOUND CLIP

Sometimes a sound is too long and you want just part of it. This can happen when you record a sound. There may be silence before and after the actual sound. Or, you may want to pull one word out of a sound. You can “clip” a sound just as we clipped a picture by copying just part of the sound. What if we want to pull the “This” out of the sound thisisatest.wav? How can we tell where it ends? Open an explorer on the sound using sound.explore();. Click at the first flat area after the first non-flat area (silence should have values near 0). Click the PLAY BEFORE button to play the part of the sound before the current index. You can use the arrow buttons to change the current index as well. Using the explorer we see that the word “this” ends at about 8500 (Figure 7.1).

So to copy just part of a sound into another sound we will need to create a new Sound object. One of the ways to make a new Sound object is to tell it how long it needs to be as a number of seconds new Sound(int numSecs). How
FIGURE 7.1: Exploring the "This is a test" to find the end of the first word

do we calculate the number of seconds that we will need? Well if we subtract the
start position from the end position and add 1 to that we will have the number of
samples. If we divide this by the number of samples per second in the sound that
will give us the number of seconds that we will need.

We can then create a new sound and loop copying from the start to the end
from the source Sound object into the target Sound object starting at the beginning
of the target sound. We have to make sure to increment both the index in the source
and the index in the target. If we forget to increment the source index we will copy
the same source sample over and over and if we fail to increment the target index
we will copy to the same place in the target over and over.

Program 56: Create a sound clip

```java
/**
 * Method to create a new sound by copying just part of the
 * current sound to a new sound
 * @param start the index to start the copy at (inclusive)
 * @param end the index to stop the copy at (inclusive)
 * @return a new sound with just the samples from start to
 * end in it
 */
public Sound clip(int start, int end)
{
    int lengthInSamples = end - start + 1; // num samples
    int lengthInSecs = (int) Math.ceil(lengthInSamples /
        this.getSamplingRate());
```
Section 7.2  Create a Sound Clip 241

Sound target = new Sound(lengthInSecs); // the target sound

int value = 0; // holds the current sample value
int targetIndex = 0; // index in target sound

// copy from start to end from source into target
for (int i = start; i <= end; i++, targetIndex++) {
    value = this.getSampleValueAt(i);
    target.setSampleValueAt(targetIndex, value);
}

return target;

To test this try:

> Sound test = new
  Sound(FileChooser.getMediaPath("thisisatest.wav"));
> test.explore();
> Sound clip = test.clip(0,8500);
> clip.play();
> clip.explore();

Use the explorer on the original sound and the clip. Change the number of
samples between pixels to be 100 in the clip. Then compare the sample values.
Convince yourself that the clip does have the same values as the original. (Fig-
ure 7.2)

![Exploring the sound clip](image)

FIGURE 7.2: Exploring the sound clip
7.3 SPlicing Sounds

Splicing sounds is a term that dates back to when sounds were recorded on tape, so juggling the order of things on the tape involved literally cutting the tape into segments and then gluing it back together in the right order. That’s “splicing”. When everything is digital, it’s much easier.

To splice sounds, we simply have to copy elements around in the array. It’s easiest to do this with two (or more) arrays, rather than copying within the same array. Splicing lets you create all kinds of sounds, speeches, non-sense, and art.

The easiest kind of splice to do is when the sounds are in separate files. All that you need to do is to copy each sound, in order, into a target sound. You need to keep track of the next index in the target sound. Here’s a recipe that creates the start of a sentence “Guzdial is...”. (Readers are welcome to complete the sentence.)

Program 57: Splice words into a single sentence

```java
/**
 * Method to splice two sounds together with some silence between them
 * @return the resulting sound
 */
public static Sound splice() {
    Sound sound1 = new Sound(FileChooser.getMediaPath("guzdial.wav"));
    Sound sound2 = new Sound(FileChooser.getMediaPath("is.wav"));
    Sound target = new Sound(FileChooser.getMediaPath("sec3silence.wav"));
    int targetIndex = 0; // the starting place on the target
    int value = 0;

    // copy all of sound 1 into the target
    for (int i = 0;
        i < sound1.getLength();
        i++, targetIndex++) {
        value = sound1.getSampleValueAt(i);
        target.setSampleValueAt(targetIndex, value);
    }

    // create silence between words by setting values to 0
    for (int i = 0;
        i < (int) (target.getSamplingRate() * 0.1);
        i++, targetIndex++) {
        target.setSampleValueAt(targetIndex, 0);
    }

    // copy all of sound 2 into the target
    for (int i = 0;
```
Section 7.3 Splicing sounds

```java
i < sound2.getLength();
i++, targetIndex++) {
    value = sound2.getSampleValueAt(i);
target.setSampleValueAt(targetIndex, value);
}

// play the resulting sound
target.play();
return target;
}
```

To test this try:

```java
> Sound s = Sound.splice();
> s.explore();
```

There are three loops in this method `splice`, each of which copies one segment into the target sound—a segment being either a word or a silence between words.

- The function starts by creating sound objects for the word “Guzdial” (s1), the word “is” (s2), and the target sound (target).

- Notice that we set `targetIndex` (the index for the target sound) equal to 0 before the first loop. We then increment it in every loop, but we never again set it to a specific value. That’s because `targetIndex` is always the index for the next empty sample in the target sound. Because each loop follows the previous, we just keep tacking samples onto the end of the target.

- In the first loop, we copy each and every sample from `s1` into the target. We have the index `i` go from 0 to one before the length of `s1`. We get the sample value at index `i` from `s1`, then set the sample value at `targetIndex` in the target sound to that value. We then increment both `i` and `targetIndex`.

- In the second loop, we create 0.1 seconds of silence. Since `target.getSamplingRate()` gives us the number of samples in one second of target, 0.1 times that tells us the number of samples in 0.1 seconds. We don’t get any source value here—we simply set the `targetIndex`-th sample to 0 (for silence), then increment the `targetIndex`.

- Finally, we copy in all the samples from `s2`, just like the first loop where we copied in `s1`.

- Next we play and return the target sound. We return the sound because we created the sound in the method. If we didn’t return the sound we created in `splice`, it would disappear with the end of the `splice` method. By returning it, we allow the sound to still be used even after the method has finished executing.

To use `new Sound(int numSeconds)` in this example, simply replace the creation of the target sound with something like:
Chapter 7  Modifying Samples using Ranges

Sound target = new Sound(3);

The more common kind of splicing is when the words are in the middle of an existing sound, and you need to pull them out from there. The first thing to do in splicing like that is to figure out the index numbers that delimit the pieces you’re interested in. Using the explorer, that’s pretty easy to do.

- Open an explorer using `sound.explore()`.
- Click the mouse button to choose a current position and then play the sound before or after that current position.
- Or, select part of the sound by clicking the mouse when the cursor points to where a place to start and dragging the mouse to the end of an area of interest. This selection will highlight and you can play the selection.

Using exactly this process, Mark found the ending points of the first few words in `preamble10.wav`. (He assumed that the first word starts at the index 0, though that might not always be true for every sound.)

<table>
<thead>
<tr>
<th>Word</th>
<th>Ending index</th>
</tr>
</thead>
<tbody>
<tr>
<td>We</td>
<td>15730</td>
</tr>
<tr>
<td>the</td>
<td>17407</td>
</tr>
<tr>
<td>People</td>
<td>26726</td>
</tr>
<tr>
<td>of</td>
<td>32131</td>
</tr>
<tr>
<td>the</td>
<td>33413</td>
</tr>
<tr>
<td>United</td>
<td>40052</td>
</tr>
<tr>
<td>States</td>
<td>55510</td>
</tr>
</tbody>
</table>

Writing a loop that copies things from one array to another requires a little bit of juggling. You need to think about keeping track of two indices: Where you are in the array that you’re copying from, and where you are in the array that you’re copying to. These are two different variables, tracking two different indexes. But they both increment in the same way.

Below is the recipe that changes the preamble from “We the people of the United States” to “We the UNITED people of the United States.”

Program 58: Splice the preamble to have united people

Be sure to change the `file` variable before trying this on your computer.

```java
/**
 * Method to splice ”United” into ”We the people of the United States” by adding it after the ”the”
 * @return the changed sound
 */
public static Sound splicePreamble()
{
    String file = FileChooser.getMediaPath("preamble10.wav");
    Sound source = new Sound(file);
    Sound target = new Sound(file);
```
int targetIndex = 17408; // declare here to keep it

// loop copying the "united" after "the"
for (int sourceIndex = 33414;
    sourceIndex < 40052;
    sourceIndex++, targetIndex++)
target.setSampleValueAt(targetIndex,
    source.getSampleValueAt(sourceIndex));

// copy after "united" the rest of the original
for (int sourceIndex = 17408;
    sourceIndex < source.getLength() &&
    targetIndex < target.getLength();
    sourceIndex++, targetIndex++)
target.setSampleValueAt(targetIndex,
    source.getSampleValueAt(sourceIndex));

// play the resulting sound
target.play();

// return the resulting sound
return target;

We’d use it as simply as saying:

> Sound s = Sound.splicePreamble();
> s.explore();

The first loop copies the word “united” into place in the target. The second one copies the rest of the original sound, making sure that we don’t go past the length of the target. Why do we need to copy the rest of the original sound? The target sound was created from the same file as the original sound so it had all the words in it originally. But, when we copied “united” after “the” we copied over part of the word “people”.

Here’s the simpler form. Try it and listen to the result.

Program 59: Splice united in a simpler form

/**
 * Method to splice "United" into "We the people of the
 * United States" by adding it after the "the"
 * @return the changed sound
 */
public static Sound spliceSimpler()
{
    String file = FileChooser.getMediaPath("preamble10.wav");
    Sound source = new Sound(file);
    Sound target = new Sound(file);
int targetIndex = 17408; // declare here to keep it

// loop copying the "united" after "the"
for (int sourceIndex = 33414;
     sourceIndex < 40052;
     sourceIndex++, targetIndex++)
target.setSampleValueAt(targetIndex, source.getSampleValueAt(sourceIndex));

// play the resulting sound
target.play();

// return the resulting sound
return target;
}

Let’s see if we can figure out what’s going on mathematically. Recall the table back on page 244. We’re going to start inserting samples at sample index 17408. The word “United” has \((40052 - 33414) = 6638\) samples. (Exercise for the reader: How long is that in seconds?) That means that we’ll be writing into the target from 17408 to \((17408 + 6638) = 24046\) sample index. We know from the table that the word “People” ends at index 26726. If the word “People” is more than \((26726 - 24046) = 2,680\) samples, then it will start earlier than 24046, and our insertion of “United” is going to trample on part of it. If the word “United” is over 6000 samples, I doubt that the word “People” is less than 2000. That’s why it sounds crunched.

Figure 7.3 shows the original preamble10.wav file in the left sound explorer, and the new spliced one (saved with write(String fileName)) on the right.

![Figure 7.3: Comparing the original sound (left) to the spliced sound (right)](image)

The method splicePreamble() takes no parameters and is a class method because of the static keyword. It is a class method because it creates the picture objects inside of it and returns it. But most of the code in the method works on a sound object. It would be better to write it as a general object method in the same way as we’ve done the general increase volume and normalize methods. To make a general splice method we will pass in the Sound object to copy from, the starting index to use in that passed sound, the index to stop before in the passed
Section 7.4 Reversing a sound

In the splicing example, we copied the samples from the words just as they were in the original sound. We don’t have to always go in the same order. We can reverse sound, and the place to copy it to in the current sound.

Program 60: General splice method

```java
/**
 * Method to copy part of the passed sound into this sound at
 * the given start index
 * @param source the source sound to copy from
 * @param sourceStart the starting index to copy from in the
 * source (the copy will include this)
 * @param sourceStop the ending index (the copy won’t include
 * this)
 * @param targetStart the index to start copying into
 */
public void splice(Sound source,
                   int sourceStart,
                   int sourceStop,
                   int targetStart)
{
    // loop copying from source to target
    for (int sourceIndex = sourceStart,
         targetIndex = targetStart;
         sourceIndex < sourceStop &&
         targetIndex < this.getLength();
         sourceIndex++, targetIndex++)
        this.setSampleValueAt(targetIndex,
                              source.getSampleValueAt(sourceIndex));
}
```

This new object method can be used to splice “united” in the phrase “We the people of the United States” as shown below:

```java
> Sound s1 = new Sound(FileChooser.getMediaPath("preamble10.wav"));
> Sound s2 = new Sound(FileChooser.getMediaPath("preamble10.wav"));
> s1.splice(s2,33414,40052,17408);
> s1.splice(s2,17408,s2.getLength(), 17408 + (40052 - 33414));
> s1.play();
> s1.explore();
```

This is a more general method that can be used to splice any two sounds together. This is also an object method (no static keyword) and must be called on a target Sound object. The part of the source Sound object from sourceStart to (sourceEnd - 1) will be copied to the Sound object this method is invoked on starting at targetStart.

7.4 REVERSING A SOUND

This new object method can be used to splice “united” in the phrase “We the people of the United States” as shown below:
the words—or make them faster, slower, louder, or softer. For an example, here’s a recipe that reverses a sound so that you can play it backwards.

**Program 61: Reverse a sound**

```java
/**
 * Method to reverse the sound with the passed file name
 * @param file the name of the file
 * @return the sound reversed
 */
public static Sound reverse(String file) {
    Sound source = new Sound(file);
    Sound target = new Sound(file);
    int length = source.getLength();

    // loop through the samples
    for (int targetIndex = 0, sourceIndex = length - 1;
         targetIndex < length;
         targetIndex++, sourceIndex --) {
        target.setSampleValueAt(targetIndex,
                                source.getSampleValueAt(sourceIndex));
    }
    return target;
}
```

This method initializes the value of `targetIndex` to 0, loops while `targetIndex` is less than the length of the sound, and increments `targetIndex` by 1 after the body of the loop. But, the value of `sourceIndex` is initialized to the length of the source minus 1 and decrements by 1 each time. Remember that the last valid index is at the length minus 1, which is why the `sourceIndex` starts with this value. So we copy from the end of the source sound (length - 1) to the beginning of the target sound (0) during the first execution of the loop. The second time through the loop we copy from the next to last sound sample in the source (length - 2) to the second position in the target (1). We will keep looping until the `targetIndex` equals the length of the sound.

You can use this method to reverse a sound (Figure ??).

```java
> Sound s = new Sound(FileChooser.getMediaPath("croak.wav"));
> s.explore();
> Sound s2 = Sound.reverse(FileChooser.getMediaPath("croak.wav"));
> s2.explore();
```

### 7.5 MIRRORING A SOUND

Once we know how to play sounds forwards and backwards, mirroring the sound is the exact same process as mirroring pictures! Compare this to Program 22,
Section 7.5 Mirroring a sound

FIGURE 7.4: Comparing the original sound (left) to the reversed sound (right) (page 119). Do you see that this is the same algorithm, though we’re dealing with a different medium?

Program 62: Mirror a sound, front to back

```java
/**
 * Method to mirror a sound front to back
 */
public void mirrorFrontToBack()
{
    int length = getLength(); // save the length
    int lastIndex = length - 1; // last valid index
    int mirrorPoint = length / 2; // mirror around this
    int value = 0; // hold the current value

    // loop from 0 to mirrorPoint
    for (int i = 0; i < mirrorPoint; i++) {
        value = this.getSampleValueAt(mirrorPoint-i);
        this.setSampleValueAt(mirrorPoint+i);
    }
}
```

To use this recipe try:

```java
> Sound s = new Sound(FileChooser.getMediaPath("croak.wav"));
> s.explore();
> s.mirrorFrontToBack();
> s.explore();
```

The sound in the file “croak.wav” is 8808, so the mirror point is at 4404. Use the explorer to check the values on either side of the mirror point.

PROBLEMS

7.1. Rewrite Program 55 (page 238) so that two input values are provided to the function: The sound, and a percentage of how far into the sound to go before dropping the volume.
7.2. Rewrite Program 55 (page 238) so that you normalize the first second of a sound, then slowly decrease the sound in steps of $1/5$ for each following second. (How many samples are in a second? `getSamplingRate()` is the number of samples per second for the given sound.)

7.3. Try rewriting Program 55 (page 238) so that you have a linear increase in volume to halfway through the sound, then linearly decrease the volume down to zero in the second half.

7.4. I think that if we’re going to say “We the UNITED people” in the splice (Program 58 (page 244)), the “UNITED” should be really emphasized—really loud. Change the recipe so that the word “united” is maximally loud (normalized) in the phrase “united people.”

7.5. Try using a stopwatch to time the execution of the recipes in this chapter. Time from hitting return on the command, until the next prompt appears. What is the relationship between execution time and the length of the sound? Is it a linear relationship, i.e., longer sounds take longer to process and shorter sounds take less time to process? Or is it something else? Compare the individual recipes. Does normalizing a sound take longer than raising (or lowering) the amplitude a constant amount? How much longer? Does it matter if the sound is longer or shorter?

7.6. Make an audio collage. Make it at least five seconds long, and include at least two different sounds (e.g., come from different files). Make a copy of one of those different sounds and modify it using any of the techniques described in this chapter (e.g., mirroring, splicing, and volume manipulations). Splice together the original two sounds and the modified sound to make the complete collage.

7.7. Compose a sentence that no one ever said, by combining words from other sounds into a grammatically correct new sound. Write a function named `audioSentence` to generate a sentence out of individual words. Use at least three words in your sentence! You can use the words in the `Speech` folder on your CD or record your own words. Be sure to include a tenth (1/10) of a second pause between the words. (Hint 1: Remember that zeroes for the sample values generate silence or pause.) (Hint 2: Remember that the sampling rate is the number of samples per second. From there, you should be able to figure out how many samples need to be made zero to generate a 1/10 of a second pause.) Be sure to access your sounds in your Media Folder using `getMediaPath` so that it will work for users of your program will work as long as they first execute `setMediaPath`.

7.8. Write a method called `erasePart` to set all the samples in the 2nd second of
Section 7.5 Mirroring a sound

“thisisatest.wav” to 0’s—essentially, making the 2nd second go silent. (Hint: Remember that `getSamplingRate()` tells you the number of samples in a single second in a sound.) Play and return the partially-erased sound.

7.9. We’ve seen a function that reverses a sound and a function that can process samples by index number.

Write a function called `reverseLastHalf` that takes a filename as input and then reverses just the second half of the sound and returns the result. For example, if the sound said “MarkBark” the returned sound should say “MarkkraB.”

7.10. Write a method similar to Program 62 (page 249) that mirrors from back to front.

7.11. Rewrite Program 61 (page 248) to be an object method.

TO DIG DEEPER

When you are using the MediaTools application, you are actually using a programming language called Squeak, developed initially and primarily by Alan Kay, Dan Ingalls, Ted Kaehler, John Maloney, and Scott Wallace [16]. Squeak is now open-source\(^1\), and is an excellent cross-platform multimedia tool. There is a book that introduces Squeak including its sound capabilities [12], and another book on Squeak [13] that includes a chapter on Siren, a variation of Squeak by Stephen Pope especially designed for computer music exploration and composition.

\(^1\)http://www.squeak.org
Chapter 7 Modifying Samples using Ranges
Bibliography


BIBLIOGRAPHY


Index

abstraction, 55
ACM Turing Award, 17
acuity, 65
algebra, 22
algorithm, 9, 12, 230, 249
  definition, 12
  mirroring, 249
alpha channel, 67
American Standard Code for Information Interchange (ASCII), 14
amplitude, 201
analog, 16
analog-to-digital conversion, 206
analog-to-digital conversion (ADC), 206
applets, 22
argument, 55
array, 21, 210
  access elements, 63
  copying, 244
  definition, 63
  element, 210
  index, 210
  notation, 239
artificial intelligence, 9
ASCII, 14
assignment, 43

background subtraction, 187
base file name, 37
binary, 13
binary number system, 13
binding, 43, 45
bit, 13
bitmap, 62
blend, 139
blending
  pictures, 139
block, 50
blue, 15
blurring, 183
body of method, 50
Brando, Marlon, 220
byte, 13, 21
Index

definition, 21

C, 11
calculus, 17
capitalization, 36
casting, 29, 88
changeVolume, 229
channel, 67
chromakey, 191
class, 20, 23
  fully qualified name, 72
class fields
  accessing, 72
class method
  declaration, 49
  declaration example, 50
class methods, 32
  definition, 32
class variables
  accessing, 72
cropped, 209
cropping, 209, 233
CMYK color model, 67
code, 20
collage, 133
color
  sepia, 176
color replacement, 164
ColorChooser.pickAColor(), 73
colorObj.brighter(), 73
colorObj.darker(), 73
commands, 48
compile, 26, 82
compiling
  definition, 26
compressed, 22, 63, 69
compression, 63
compressions, 201
computational recipe, 8
computer, 13
computer hardware, 206
computer music, 10
constants, 10, 112
convention, 27
coordinates, 64
copying, 126
cropping, 130
cycle, 201
cycles per second, 202
data, 21
data representation, 15
data structures, 9, 15
databases, 9
debugging
    using print statements, 124
debugging, 226, 232
    using print statements, 124
decibels, 202
decimal number, 13
definitions pane, 26
Digital, 16
Digital media, 16
digital-to-analog conversion, 207
digital-to-analog conversion (DAC), 206
digitization, 16
digitizing media
    sounds, 206
digitizing media
    pictures, 65
    why?, 16
directory, 21, 37
disk, 21
double quote, 21, 37
DrJava, 24
    definitions pane, 26
    interactions pane, 26
    files pane, 25
    installing, 24
    running slowly, 25
    starting, 24
Dynabook, 18
edge detection, 171
editor, 24
emergent properties, 10
encoding, 8, 13, 37
equal temperament, 203
error
    illegal start of expression, 50
    semicolon expected, 50
evaluation, 47, 55
explore(), 70
expression, 28, 44
Index

Fast Fourier Transform, 205
FFT, 205
file, 21
file extension, 37
FileChooser.getMediaPath(fileName), 123, 126
files pane, 25
floating point
typing, 29
floating point number, 18, 21
for loop, 103
Fourier transform, 205
frequency, 201, 202, 207
frequency domain, 205
full file name, 37
fully qualified name, 72
function
arguments, 55
input parameters make functions more general, 56
input values, 55
input variables, 55
parameters, 55
function calls, 47
functions, 31
fundamental, 203
garbage collection, 35
general, 97
getColor(), 71
getHeight(), 71
getPixel(x,y), 71
getPixels(), 71
getRed(), 71
goldWidth(), 71
g getX(), 71
g getY(), 71
going digital, 16
graphics, 10
greyscale, 108
green, 15

hard disk, 21
hardware, 206
Hertz, 202
hierarchical decomposition, 99
hierarchy, 99
HSB, 66
HSV color model, 66
HTML, 15
human-computer interface, 9
immutable, 34
import, 72
  how to, 72
  why, 72
infinite loop, 80, 81, 142
  stopping one, 80
Ingalls, Dan, 251
inherit, 34
inheritance, 34
input, 33
input value, 55
input variables, 55
integer, 21
typing, 29
Intel, 15
intelligence, 9
intelligent systems, 9
intensity, 108, 201
interactions pane, 26
interface, 9
iterate, 216
Java, 11, 22
  capitalization, 36
  installing, 24
Javadoc comments, 26
JPEG, 37, 63
Kaehler, Ted, 251
Kay, Alan, 251
kilobytes, 69
knots, 9
liberal education, 17
Lisp, 11
loop, 216
  nested, 116
lossy compression, 63
luminance, 65, 67, 109
makeSound
  what it does, 211
Maloney, John, 251
matrix, 64
  definition, 64
max, 230
maximum, finding, 230
media computation, 15
MediaTools, 75, 204
  application, 214
  picture tools, 75
  sound tools, 204
megabyte, 69
memory, 13, 25
memory leak, 36
method, 20
  definition, 32
method body, 50
methods, 32
methods vs. recipes, 98
mirroring, 248
modulo, 29
Moore’s Law, 15
Moore, Gordon, 15

negative image, 106
nested, 116
nested loops, 116
networking, 10
new Color(redValue,greenValue,blueValue), 71
new Picture(fileName), 70
new Picture(width,height), 150
new Sound(fileName), 42
new Sound(int numSeconds), 243
noise, 205
normalizing, 230
Nyquist theorem, 207
  applications, 207

object, 211
  definition, 23
object method
  declaration, 49
  declaration example, 49
object methods, 32
  definition, 32
object reference, 41
object variable, 41
object-oriented, 22
objects, 211
operating system, 21
operators, 28
overloading, 139, 182
overtones, 203

package
  definition, 72
parameters, 55
pass by value, 55
path, 37
path separator, 37
PCM, 209
percentage, 77
Perlis, Alan, 17
physics
  color, 15
pickAFile(), 36
picture
  show(), 38
picture element, 15
picture objects, 62
picture tool, 75
pictureObject.repaint(), 73
pitch, 202
pitch interval, 203
pixel, 15, 62, 64
  definition, 64
pixelation, 183
pixelization, 66
pixels, 62
placeholder, 100
play(), 42
posterizing, 178
primitive, 40, 41
primitive type, 40
private visibility
  definition, 49
process, 17
program, 9, 12, 20
  defined, 20
  definition, 12
programming language, 20
programming languages, 10
psychoacoustics, 202
public visibility
  definition, 49
pulse coded modulation (PCM), 209
ranges, 10
rarefactions, 201
recipe, 8
recipes vs. methods, 98
red, 15
reference, 41
return, 100, 230
reusable, 97
RGB color model, 66
Roads, Curtis, 236
rounding errors, 73
sample, 16, 207
sample objects, 211
sample size, 233
sample sizes, 208
sampling, 146
sampling rate, 213
sampling rate, 208
sawtooth, 204
scope, 87, 99
definition, 99
sepia-toned, 176
setColor(color), 71
setRed(redValue), 71
show(), 38, 70
sign bit, 209
signal view, 204
sine wave, 201
Siren, 251
slice, 205
software engineering, 9, 17
sonogram view, 205
sound
amplitude, 201
blockingPlay, 214
cracking, 209
decibels, 202
decreasing volume, 226
definition, 201
decreasing volume, 211
decreasing volume, 205
frequency domain, 205
fundamental, 203
getLength(), 212
getSamples, 211
getSampleValueAt, 212
getSamplingRate, 213
getValue, 212
how to manipulate, 210
increasing volume, 220
intensity, 201
new Sound(fileName), 42
overtones, 203
pitch, 202
play(), 42
pulse coded modulation, 209
sampling rate, 208
setSampleValueAt, 212
setValue, 212
sonogram view, 205
splicing, 242
volume, 201, 220, 226
sound cursor, 215
sound editor, 204
sound object, 210
sound pressure level, 202
source, 126
spectrum view, 205
spike, 205
splicing, 242
square brackets, 239
Squeak, 251
static, 49
stepping, 84
String, 21, 47
strings, 29
strongly typed, 21
sub-program, 230
subscript, 239
substitution, 47
substitution, 55
Sun, 24
symbols, 20
syntax, 49
systems, 9
target, 126
testing, 224
  using print statements, 124
testing methods, 224, 232
The Godfather, 220
time domain, 205
trace, 84
tracing, 226
using print statements, 124
transforming, 126
transistor, 15
transparency, 67, 139
two's complement notation, 209
type, 20
types
byte, 21
double, 21
float, 21
int, 21
integer array, 21
String, 21

Unicode, 18

variable, 39
declaration, 30, 39
object, 41
primitive, 41
variables, 22, 43
declaring, 43
reusing, 44
visibility, 49
definition, 49
private, 49
public, 49
volume, 201

walking through, 84
Wallace, Scott, 251
WAV, 37
while, 77
while loop, 77, 217
window (of time), 205
wrapper, 32
wrapper classes
definition, 32
write(fileName), 73