Introduction to 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

Contents iv

Preface 1

I Introduction 5

1 Introduction to Media Computation 6
  1.1 What is computer science about? 6
  1.2 What Computers Understand 9
  1.3 Media Computation: Why digitize media? 11
  1.4 Computer Science for Non-Computer Scientists 13
    1.4.1 It’s about communication 13
    1.4.2 It’s about process 13

2 Introduction to Programming 16
  2.1 Programming is about Naming 16
    2.1.1 Files and their Names 18
  2.2 Programming in Java 19
    2.2.1 History of Java 19
    2.2.2 Introduction to Objects and Classes 19
    2.2.3 Introduction to DrJava 20
  2.3 Programming in DrJava 21
  2.4 Media Computation in DrJava 24
    2.4.1 Showing a Picture 27
    2.4.2 Playing a Sound 30
    2.4.3 Naming your Media (and other Values) 31
  2.5 Making a Recipe 35
    2.5.1 Variable Recipes: Real functions that Take Input 40

II Pictures 45

3 Encoding and Manipulating Pictures 46
  3.1 How Pictures are Encoded 46
  3.2 Manipulating Pictures 52
    3.2.1 Exploring pictures 58
  3.3 Changing color values 58
    3.3.1 Using loops in pictures 59
    3.3.2 Increasing/decreasing red (green, blue) 61
    3.3.3 Creating a sunset 70
    3.3.4 Making sense of methods 71
    3.3.5 Lightening and darkening 79
## List of Figures

1.1 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 ................................. 10
1.2 Alan Perlis .................................................................................. 14
2.1 DrJava Preferences Window ........................................................ 21
2.2 DrJava Splash Screen ............................................................... 22
2.3 DrJava (with annotations) ......................................................... 23
2.4 The File Chooser ........................................................................ 26
2.5 File chooser with media types identified ................................. 28
2.6 Picking, making, and showing a picture, using the result of each method in the next method ......................................................... 29
2.7 Picking, making, and showing a picture, when naming the pieces . 31
2.8 Defining and executing `pickAndShow()` ................................. 37
3.1 An example matrix ...................................................................... 47
3.2 Upper left corner of DrJava window with part magnified 600% ... 48
3.3 Image shown in the picture explorer: 100% image on left and 500% on right ................................................................................. 48
3.4 Merging red, green, and blue to make new colors ..................... 49
3.5 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 50
3.6 The Macintosh OS X RGB color picker ................................... 51
3.7 Picking a color using RGB sliders from Java ............................ 51
3.8 RGB triplets in a matrix representation ..................................... 52
3.9 Directly modifying the pixel colors via commands: Note the small black line on the left under the leaf ............................................ 57
3.10 Using the MediaTools image exploration tools ......................... 59
3.11 The original picture (left) and red-reduced version (right) .......... 63
3.12 Using the picture explorer to convince ourselves that the red was decreased ............................................................................... 68
3.13 Overly blue (left) and red increased by 30% (right) .................. 69
3.14 Original (left) and blue erased (right) ........................................ 70
3.15 Original beach scene (left) and at (fake) sunset (right) .......... 71
3.16 Lightening and darkening of original picture ............................ 81
3.17 Negative of the image ............................................................... 82
3.18 Color picture converted to grayscale ........................................ 84
3.19 Once we pick a mirror point, we can just walk x halfway and subtract/add to the mirror point ......................................................... 88
3.20 Original picture (left) and mirrored along the vertical axis (right) . 89
3.21 Santa mirrored horizontally, bottom to top (left) and top to bottom (right) ................................................................. 91
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.22</td>
<td>Temple of Zeus from the ancient agora in Athens, Greece</td>
</tr>
<tr>
<td>3.23</td>
<td>Coordinates where we need to do the mirroring</td>
</tr>
<tr>
<td>3.24</td>
<td>The manipulated temple</td>
</tr>
<tr>
<td>3.25</td>
<td>Copying a picture to a canvas</td>
</tr>
<tr>
<td>3.26</td>
<td>Copying a picture midway into a canvas</td>
</tr>
<tr>
<td>3.27</td>
<td>Copying part of a picture onto a canvas</td>
</tr>
<tr>
<td>3.28</td>
<td>Flowers in the mediasources folder</td>
</tr>
<tr>
<td>3.29</td>
<td>Collage of flowers</td>
</tr>
<tr>
<td>3.30</td>
<td>Blending the picture of Katie and Jenny</td>
</tr>
<tr>
<td>3.31</td>
<td>Copying a picture to a canvas rotated to the left 90 degrees</td>
</tr>
<tr>
<td>3.32</td>
<td>Scaling the picture down</td>
</tr>
<tr>
<td>3.33</td>
<td>Scaling up a picture</td>
</tr>
<tr>
<td>3.34</td>
<td>Increasing reds in the browns</td>
</tr>
<tr>
<td>3.35</td>
<td>Increasing reds in the browns, within a certain range</td>
</tr>
<tr>
<td>3.36</td>
<td>Finding the range of where Jenny’s eyes are red</td>
</tr>
<tr>
<td>3.37</td>
<td>After fixing red-eye</td>
</tr>
<tr>
<td>3.38</td>
<td>Original scene (left) and using our sepia-tone recipe</td>
</tr>
<tr>
<td>3.39</td>
<td>Reducing the colors (right) from the original (left)</td>
</tr>
<tr>
<td>3.40</td>
<td>Pictures posterized to two levels (left) and four levels (right)</td>
</tr>
<tr>
<td>3.41</td>
<td>Making the flower bigger, then blurring to reduce pixellation</td>
</tr>
<tr>
<td>3.42</td>
<td>Merging red, green, and blue to make new colors</td>
</tr>
<tr>
<td>3.43</td>
<td>Color: RGB triplets in a matrix representation</td>
</tr>
<tr>
<td>3.44</td>
<td>Color: The original picture (left) and red-reduced version (right)</td>
</tr>
<tr>
<td>3.45</td>
<td>Color: Overly blue (left) and red increased by 30% (right)</td>
</tr>
<tr>
<td>3.46</td>
<td>Color: Original (left) and blue erased (right)</td>
</tr>
<tr>
<td>3.47</td>
<td>Original beach scene (left) and at (fake) sunset (right)</td>
</tr>
<tr>
<td>3.48</td>
<td>Color: Lightening and darkening the original picture</td>
</tr>
<tr>
<td>3.49</td>
<td>Color: Negative of the image</td>
</tr>
<tr>
<td>3.50</td>
<td>Color: Color picture converted to grayscale</td>
</tr>
<tr>
<td>3.51</td>
<td>Color: Increasing reds in the browns</td>
</tr>
<tr>
<td>3.52</td>
<td>Color: Increasing reds in the browns, within a certain range</td>
</tr>
<tr>
<td>3.53</td>
<td>Finding the range where Jenny’s eyes are red, then changing them to black</td>
</tr>
<tr>
<td>3.54</td>
<td>Frames from the slow sunset movie</td>
</tr>
<tr>
<td>3.55</td>
<td>Frames from the slow fade-out movie</td>
</tr>
<tr>
<td>3.56</td>
<td>Frames from the Mommy watching Katie movie</td>
</tr>
<tr>
<td>3.57</td>
<td>Frames from the original too dark movie</td>
</tr>
<tr>
<td>3.58</td>
<td>Frames from the modified lighter movie</td>
</tr>
<tr>
<td>3.59</td>
<td>Frames from the original movie with kids crawling in front of a blue screen</td>
</tr>
<tr>
<td>3.60</td>
<td>Frames from the kids on the moon movie</td>
</tr>
<tr>
<td>4.1</td>
<td>A picture of a child (Katie), and her background without her</td>
</tr>
<tr>
<td>4.2</td>
<td>A new background, the moon</td>
</tr>
<tr>
<td>4.3</td>
<td>Katie on the moon</td>
</tr>
<tr>
<td>4.4</td>
<td>Two people in front of a wall, and a picture of the wall</td>
</tr>
<tr>
<td>4.5</td>
<td>Swapping a jungle for the wall, using background subtraction, with a threshold of 50</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

4.6 Mark in front of a blue sheet ................. 148
4.7 Mark on the moon ............................. 149
4.8 Mark in the jungle ............................. 150
4.9 Student in front of a red background, and with flash on .... 151
4.10 Using chromakey recipe with red background ............ 151
4.11 Santa with a grid of drawn lines ................ 153
4.12 A small, drawn picture ........................ 156
4.13 A programmed gray scale effect ............... 158
4.14 Nested filled rectangles image ................ 161
4.15 Nested rectangles image ...................... 162
Preface

This book is based on the proposition that the best way to learn to program is to have something interesting to program. Most educated people want to use a computer, and the task that they most want to do with a computer is communicate. Alan Perlis first made the claim in 1961 that computer science, and programming explicitly, should be part of a liberal education [8]. What we’ve learned since then is that one doesn’t just “learn to program.” One learns to program something [3, 9], and the motivation to do that something can make the difference between learning to program or not [4].

The philosophies which drive the structure of this book include:

- People learn from concrete examples to abstract ideas, driven by need. Teaching structure before content is painful and results in brittle knowledge that can’t be used elsewhere [5]. Certainly, one can introduce structure (and theory and design), but students won’t really understand the structure until they have the content to fill it with – and a reason to need the structure. Thus, this book doesn’t introduce debugging or design (or complexity or most of computer science) until the students are doing complex enough software to make it worthwhile learning.

- Repetition is good. Variety is good. Marvin Minsky once said, “If you know something only one way, you don’t know it at all.” The same ideas come back frequently in this book. The same idea is framed in multiple ways. I will use metaphor, visualizations, mathematics, and even computer science to express ideas in enough different ways that hopefully one of the ways will ring true for the individual student.

- 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.

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");
    }
}
```
2 LIST OF FIGURES

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.

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

Recipe 1: An Example Recipe

Recipes (programs) appear like this:

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

End of Recipe 1

Computer Science Idea: An Example Idea
Key computer science concepts appear like this.

Common Bug: An Example Common Bug
Common things that can cause your recipe to fail appear like this.

Debugging Tip: An Example Debugging Tip
If there’s a good way to keep those bugs from creeping into your recipes in the first place, they’re highlighted here.
FOR THE TEACHER

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 include the book classes in the classpath.

It’s possible to teach a class where students learn the algorithms from this book, but apply them in another language. The media manipulations described in this book can easily be used in other languages. Examples from this book have been successfully used in classes using Scheme\(^1\), and Squeak\(^2\).

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.
- Jeff Pierce reviewed and advised us on the design of the media language used in the book.
- Picture of Alan Perlis from [http://www.cs.cmu.edu/afs/cs.cmu.edu/Web/csd/perlis.html](http://www.cs.cmu.edu/afs/cs.cmu.edu/Web/csd/perlis.html). Most of the clip art is used with permission from the Art Explosion package by Nova Development.
- 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 my thanks!

---

\(^1\)JScheme, [http://jscheme.sf.net](http://jscheme.sf.net)

\(^2\)http://www.squeak.org
PART ONE

INTRODUCTION

Chapter 1  Introduction to Media Computation
Chapter 2  Introduction to Programming
CHAPTER 1

Introduction to 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
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 recipe 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 position, then being able to write a recipe 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...
Section 1.1 What is computer science about?

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).

- 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 recipe can be changed without changing the result dramatically. You can always increase all the units by a multiplier 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 sauté’ them, the reverse order will probably not work well.

Similarly, for software recipes. 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 recipes with programming languages. Different programming languages are used for different purposes. Some of them are wildly popular, like Java and C++. 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 recipes. 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 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 recipe. When was the last time you told someone how a videogame like Doom or Quake or Super Mario Brothers worked in such minute detail that they could actually replicate the game (say, on paper)? English isn’t good for that kind of task.

There are so many different kinds of programming languages because there are so many different kinds of recipes 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 recipe 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 recipes 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
Section 1.2 What Computers Understand

Computational recipes are written to run on computers. What does a computer know how to do? What can we tell the computer to do in the recipe? 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 (like eight of these wires are called a byte and one of them 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 interpret a pattern of eight 0’s and 1’s, e.g., 01001010. To calculate the decimal number (10 based) that 01001010 represents we add up the values of the digits using the binary number (2 based) system starting from right to left as follows: $2^0 \cdot 0 + 2^1 \cdot 1 + 2^2 \cdot 0 + 2^3 \cdot 0 + 2^4 \cdot 1 + 2^5 \cdot 0 + 2^6 \cdot 1 + 2^7 \cdot 0 = 74$ (Figure 1.1). We can represent the numbers 0 (00000000) to 255 (11111111) using eight wires (a byte). That’s where we come up with the claim that a computer knows about numbers.$^1$

The computer has a memory made up of 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 recipe “Compare these two numbers. If the first one is less than the second one, jump to step 5 in this recipe. Otherwise, continue on to the next step.”

So far, the computer is an incredible calculator, and that’s certainly why it was invented. The first use of the computer was during World War II for calculating

$^1$We’ll talk more about this level of the computer in Chapter ??
Chapter 1  Introduction to Media Computation

FIGURE 1.1: 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.

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 what makes it useful for general recipes 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 sets as Web pages, which can be interpreted to appear as multiple fonts and styles. 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). It could also be the letter A in Unicode which is a more recent standard encoding of number to characters which supports a wide range of languages. 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 recipes 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
Section 1.3  Media Computation: Why digitize media?

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 a lot 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!

Computer Science Idea: Moore’s Law

Gordon Moore, one of the founders of Intel (maker of computer processing chips used by computers running Windows operating systems), made the claim that the number of transistors (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 computing power every 18 months. This Law has continued to hold true for decades.

Computers today can execute literally BILLIONS of recipe steps per second! They can hold in memory literally encyclopediae 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 collection of three bytes, one each for the amount of red, green, and blue at that dot on the screen? And we collect a bunch of these three-byte-sets to determine 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 are thousands or even millions of them in a picture that you might want to work with on your computer or on the Web. 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 *analogue* 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 slower (but easy to understand) languages, even with inefficient (but easy to read and write) recipes, we can learn about computation by manipulating media.
1.4 COMPUTER SCIENCE FOR NON-COMPUTER SCIENTISTS

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 [8]. Perlis is an important figure in the field of computer science (Figure 1.2). 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 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 6), 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.

PROBLEMS

1.1. 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.

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

1.3. 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?)

1.4. 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?

*1.5. How might you represent a floating point number in terms of bytes?

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

1.8. Look up Kurt Gödel 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* [12] describes how ants, termites, and even traffic jams and slime molds can be described pretty accurately with hundreds or thousands of very small programs running and interacting all at once.

*Beyond the Digital Domain* [2] is a wonderful introductory book to computation with lots of good information about digital media.
CHAPTER 2

Introduction to Programming

2.1 PROGRAMMING IS ABOUT NAMING
2.2 PROGRAMMING IN JAVA
2.3 PROGRAMMING IN DRJAVA
2.4 MEDIA COMPUTATION IN DRJAVA
2.5 MAKING A RECIPE

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 naming scheme (the names and what they name) 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 words and the words’ 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, even a way that extends how we think about recipes (processes) entirely.

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. Assigning a variable to a value is one way of defining a name for the computer. Defining a method (function) is giving a name to a recipe. In Java you can also assign a name to a group of related data and methods (functions) when you define a class (type).
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.

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 and build on. 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).

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 recipes 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 recipe than others—but sometimes that “more” leads to a much more (human) readable recipe 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, but 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 recipe can be interpreted is often also named. Remember how we said in Section 1.2 (page 9) that everything is in bytes, but 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
Chapter 2  Introduction to Programming

point number (a 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. Their 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 will only be a floating point number. 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 or use include Windows 95, Windows 98 (Windows ME, NT, XP...), 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 recipes, though, you’ll be explicitly getting file names and asking for their values.

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 11)? 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.

In contrast, your computer’s memory is impermanent (disappears 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 so you can work with it, 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 a dorm room. You can get to things easily in a dorm room—they’re right at hand, easy to reach, easy to use. But you
wouldn’t want to put everything you own (or everything you hope to own) in that one dorm room. All your belongings? Your skis? Your car? Your boat? That’s silly. Instead, you store large things in places designed to store large things. You know how to get them when you need them (and maybe take them back to your dorm room if you need to or can).

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 \( e = Me^2 \) 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 media (as values) in the same way when using them when programming.

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 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 are doing the action in a situation or being acted upon.
Chapter 2  Introduction to Programming

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.

Each object of a class will have the same skills or operations (things it can do) and data (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.

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 recipes within it. DrJava is available for free under the DrJava Open Source License, and it is under active development by the JavaPLT group at Rice University.

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.
Section 2.3 Programming in DrJava

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. Click on the ADD button below the EXTRA CLASSPATH textarea and add the following path: c:/intro-prog-java/bookClasses.

 FIGURE 2.1: DrJava Preferences Window

2.3 PROGRAMMING IN DRJAVA

How you start DrJava depends on your platform. 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. In Windows, you’ll have a DrJava icon that you’ll simply double-click. 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.
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 DrJava 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 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).

- The top right part is the definitions pane. This where you write your classes: The 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.

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 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
Section 2.3  Programming in DrJava  23

A semicolon at the end of a Java statement in the interactions pane it will do the statement but not automatically print the result in the interactions pane.

There are other features of DrJava visible in Figure 2.3. The OPEN button will let you open a file and 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 the HTML documentation from the Javadoc comments in your files (comments that start with '/*' and end with '*/').

FIGURE 2.3: DrJava (with annotations)

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 when you start writing your own programs.
2.4 MEDIA COMPUTATION IN DJAVA

We’re going to start out by simply typing commands in the interactions pane—not defining new names yet, but simply using the names that the computer already knows from within Java (keywords, operators, and classes that come with a release of Java).

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.” 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
```

`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
> System.out.println("Hello" + "Mark")
HelloMark
```

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. 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:

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

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

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 calls these methods.

However, you can’t just call a function or method in Java like you can in procedural languages. Every method or function must be defined in a class. There are two types of methods in Java: class methods or object methods. Class methods can be invoked (executed) by using the class name followed by a period and then the method name. By convention class names in Java start with an uppercase letter: like Character.

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 Unicode mapping 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:

```java
> 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 value.

```java
> System.out.println(Math.abs(1))
1
> System.out.println(Math.abs(-1))
1
```
Chapter 2 Introduction to Programming

FIGURE 2.4: The File Chooser

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.

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

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

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

FileChooser.pickAFile() is a class method on the FileChooser class. This is a class that we created to make it easy for you to pick a file name and return a string which represents the full path name of that file. The name of the function (method) 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 2.4.

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.
- Click to select 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
Java keyword that means nothing. Try it: Do System.out.println(FileChooser.pickAFile()) and OPEN a file.

> System.out.println(FileChooser.pickAFile())
C:\intro-prog-java\mediasources\cat.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., “\”). On Linux or MacOS, it will probably look something like the above. 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.
- 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.

Files that have an extension of “.jpg” 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 images. The other kind of media files that we’ll be using frequently are “.wav” files (Figure 2.5). 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.

2.4.1 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 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).

> System.out.println(new Picture(FileChooser.pickAFile()))
Picture, filename
C:\intro-prog-java\mediasources\partFlagSmall.jpg height 217 width 139
Chapter 2  Introduction to Programming

FIGURE 2.5: File chooser with media types identified

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).

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 2.6. This code will first invoke the `pickAFile()` class method of the class `FileChooser` and that will return 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.

- The second way is to name each of the pieces by using =. However, in Java we can't just use new names without saying what type of thing we expect the name to represent. We call this declaring a variable. 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 typically use only int, double, and boolean in this book. The type int is for integer 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 it takes up 1 bit. Java uses primitive types to speed calculations.

A class name 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 (String fileName =) that we get from FileChooser.pickAFile().

FIGURE 2.6: Picking, making, and showing a picture, using the result of each method in the next method.
This says that the name “fileName” will be of type String (will represent 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 name `picture` that will represent an object of the Picture class that we get from creating a new Picture object with the `fileName` 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 2.7.

### 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 a uppercase letter and `picture` is a variable name since it starts with a lowercase letter. 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 be upset with 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 use 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 opening up a picture in a window).

#### 2.4.2 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 now use `new Sound(fileName)` to make a Sound object. `new Sound(fileName)`, as you might imagine, takes a name of a file as input.

- 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(Selector.pickAFile()))
Sound file: croak.wav length: 17616
> new Sound(Selector.pickAFile()).play()
```

We’ll explain what the length of the sound means in the next chapter. 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!

### 2.4.3 Naming your Media (and other Values)

The code `new Sound(Selector.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 functions.

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 name data using =. We can check our namings using `System.out.println()`, just as we have been doing.

```java
> 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 “becomes a name for.” myVariable=12 thus means “myVariable becomes a name for 12.” The reverse (putting the expression on the left and the name on the right) thus makes no sense: 12 = myVariable would then mean “12 becomes a name for myVariable.”

```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

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.

```java
> 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 (or even names and functions) only exist during a session of DrJava.

Remember that data do have encodings or types. How the data act in expressions depends in part of their types. Notice how the *integer*: the primitive type int 12 and the *string*: an object of the String class “12” act differently for addition below. Both are doing something reasonable for their type, but they are very different actions.

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

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 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 ??

```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 assign names to the file name and picture.

```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(FileChooser.pickAFile())\(^1\), 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

```java
new Picture ("C:\intro-prog-java\mediasources\katie.jpg")
```

\(^1\)Assuming, of course, that you picked the same file.
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.

```
> 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);
> 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
```

**Computer Science Idea:** We can substitute names, values, and methods.

We can substitute a value, a name assigned to that value, and the method returning that value *interchangeably*. The computer cares about the values, not if it comes from a string, a name, or a method (function) call.

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 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.

### 2.5 MAKING A RECIPE

We have now used names to stand for values. The values get substituted for the names when the expression is evaluated. We can do the same for recipes. 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 recipe or program 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 recipes.

**Making it Work Tip: Try *every* recipe!**
To really understand what’s going on, type in, compile, and execute *every* recipe 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 recipe 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 recipe as well as what are the steps to take in doing the recipe. 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 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 *public* `Type methodName(parameterList)`. Here the 'Type' is the type of value being returned from the method. Remember that a type can be any of the primitive types (char, byte, short, int, long, float, double, boolean) or a class name.

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 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 done). 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 your phone number is public (listed) then anyone can look it up and call you. If your phone number is private (unlisted) then only the people that live at the same house will be able to call you.

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 class and 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 FileChooser 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.

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).

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.

We can now define our first recipe! Open Picture.java by clicking on the 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
FIGURE 2.8: Defining and executing pickAndShow()

click the Compile All button near the top of the window (Figure 2.8).

Recipe 2: Pick and show a picture

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

End of Recipe 2

Now you can execute your recipe (Figure ??). 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.

> Picture.pickAndShow()

We can similarly define our second recipe, to pick and play a sound. Open the Sound class definition file and type the following before the last close 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.

Recipe 3: Pick and play a sound

```java
public static void pickAndPlay()
{
    String fileName = FileChooser.pickAFile();
    Sound sound = new Sound(fileName);
    sound.play();
}
```

End of Recipe 3

Making it Work Tip: Name the names you like

You’ll notice that, in the last section, we were using the names myFileName and myPicture. In this recipe, I used fileName and picture. Does it matter? Absolutely not! Well, 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 you show your program to can understand it), and (c) are easy to type. 25-character names, like, myPictureThatIAmGoingToOpenAfterThis are meaningful, easy-to-read, but are a pain to type.

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 recipes, we can define new ones however we like! Let’s define one that will just open a specific picture we want, and another that opens a specific sound we want.

Use FileChooser.pickAFile() to get the file name of the sound or picture that you want. We’re going to need that in defining the recipe 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 quotes directly in the recipe.

Recipe 4: Show a specific picture

Type in the following code before the last ’}’ 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.

```java
public static void showPicture()
{
    String myFile = "FILENAME";
    Picture myPicture = new Picture(myFile);
    myPicture.show();
}
```

End of Recipe 4

Recipe 5: Play a specific sound

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.

```java
public static void playSound()
{
    String myFile = "FILENAME";
    Sound mySound = new Sound(myFile);
    mySound.play();
}
```

End of Recipe 5
2.5.1 Variable Recipes: Real functions that Take Input

How do we create a method (function) with inputs out of our stored recipe, 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 Recipe 4, Picture.showPicture(). 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 recipe that takes input is very easy. It continues to be a matter of substitution and evaluation. We’ll put a name inside those parentheses on the def line. That name is sometimes called the parameter or input variable.

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 input value is assigned to the input variable. We say that the input variable “takes on” the input value. During the execution of the function (recipe), the input value will be substituted for the value.

Here’s what a recipe would look like that takes the file name as an input variable:

---

**Recipe 6: Show the picture file whose file name is input**

```
public static void showNamed(String fileName)
{
    Picture myPicture = new Picture(fileName);
    myPicture.show();
}
```
Section 2.5 Making a Recipe

End of Recipe 6

When I type
Picture.showNamed("C:/intro-prog-java/mediasources/katie.jpg")
and hit return, the variable fileName takes on the value
"C:/intro-prog-java/mediasources/katie.jpg".
myPicture will then be assigned to the picture resulting from reading and interpreting the file at
‘‘C:/intro-prog-java/mediasources/katie.jpg’’
Then the picture is shown.

We can do a sound in the same way.

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

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

End of Recipe 7

We can also write recipes that take pictures or sounds in as the input values. Here’s a recipe that shows a picture but takes the picture object as the input value, instead of the filename.

Recipe 8: Show the picture provided as input

public static void showPicture(Picture myPicture)
{
    myPicture.show();
}

End of Recipe 8

Now, what’s the difference between the method (function) showPicture() and the provided method show()? Nothing at all. We can certainly create a function that provides a new name to another function. If that makes your code easier for you to understand, than it’s a great idea.

What’s the right input value for a function? Is it better to input a filename or a picture? And what does “better” mean here, anyway? You’ll read more about all
of these later, but here’s a short answer: Write the function that is most useful to you. If defining `showPicture()` is more readable for you than `show()`, then that’s useful. If what you really want is a method that takes care of making the picture and showing it to you, then that’s more useful and you might find the `showNamed()` method the most useful.

**OBJECTS AND FUNCTIONS SUMMARY**

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

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integers</td>
<td>Java primitive type int e.g., 3</td>
<td>Numbers without a decimal point—they can’t represent fractions.</td>
</tr>
<tr>
<td>Floating point numbers</td>
<td>Java primitive type double e.g., 5.2, 3.01</td>
<td>Numbers with a fractional piece to them.</td>
</tr>
<tr>
<td>Strings</td>
<td>Java String 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>
<tr>
<td>File name</td>
<td>Java String object</td>
<td>A filename is just a string whose characters represent a path, path separators, and a base file name.</td>
</tr>
<tr>
<td>Pictures</td>
<td>object of our Picture class</td>
<td>Pictures are encodings of images, typically coming from a JPEG file.</td>
</tr>
<tr>
<td>Sounds</td>
<td>object of our Sound class</td>
<td>Sounds are encodings of sounds, typically coming from a WAV file.</td>
</tr>
</tbody>
</table>

Here are the functions introduced in this chapter:
Section 2.5 Making a Recipe

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character.getNumericValue(character)</td>
<td>Returns the equivalent numeric value in Unicode for the input character.</td>
</tr>
<tr>
<td>Math.abs(number)</td>
<td>Takes a number and returns the absolute value of it.</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>new Picture(fileName)</td>
<td>Takes a name of a file as input, reads the file, and creates a picture from it. Returns the created picture object.</td>
</tr>
<tr>
<td>picture.show()</td>
<td>Shows the picture object that it is invoked on. No return value.</td>
</tr>
<tr>
<td>new Sound(fileName)</td>
<td>Takes a filename as input, reads the file, and creates a sound from it. Returns the sound object.</td>
</tr>
<tr>
<td>sound.play()</td>
<td>Plays the sound object that it is invoked on.</td>
</tr>
</tbody>
</table>

PROBLEMS

2.1. Try some other operations with strings in Java. What happens if you add a number to a string, like 3 + "Hello"?

2.2. You can combine the sound playing and picture showing commands in the same recipe. 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?

2.3. We evaluated the expression FileChooser.pickAFile() when we wanted to execute the function named pickAFile(). But what is the name pickAFile() anyway? You could open the FileChooser class and find that method declaration.

2.4. 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 [1], 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 [6].

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* [10] 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](http://java.sun.com) including tutorials, papers, and APIs. To learn more about DrJava see the web site [http://www.drjava.org/](http://www.drjava.org/). *Thinking in Java* by Bruce Eckel is a great book for those who have some coding experience and like to understand a language deeply. Beginners might want to start with *Beginning Java* by Ivor Horton, or *Headfirst Java* by Kathy Sierra and Bert Bates.
PART TWO

PICTURES

Chapter 3  Encoding and Manipulating Pictures
Chapter 4  Advanced Pictures
Chapter 5  Advanced Sounds
Chapter 6  Design and Debugging
CHAPTER 3

Encoding and Manipulating Pictures

3.1 HOW PICTURES ARE ENCODED

Pictures are two-dimensional arrays of pixels. In this section, each of those terms will be described.

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.

You can access elements of an array in Java using arrayName[index]. For example, to access the first element in an array named pixels use pixels[0]. You can get the number of items in an array using arrayName.length.
Section 3.1 How Pictures are Encoded

Making it Work Tip: Using dot notation for public fields

Notice that there are no parenthesis 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 3.1, 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)).

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>15</td>
<td>12</td>
<td>13</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>9</td>
<td>7</td>
<td>43</td>
<td>23</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>13</td>
<td>15</td>
<td>16</td>
</tr>
</tbody>
</table>

**FIGURE 3.1:** An example matrix

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 3.2 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.

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% where each individual pixel is visible (Figure 3.3).

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
Chapter 3  Encoding and Manipulating Pictures

FIGURE 3.2: Upper left corner of DrJava window with part magnified 600%

FIGURE 3.3: Image shown in the picture explorer: 100% image on left and 500% on right

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 your 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 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. A picture has two dimensions, a width and a height.

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 will 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 3.4) (replicated at Figure 3.42 (page 134). Combining all three gives us pure white. Turning off all three gives us black. We call this the **RGB model**.

![FIGURE 3.4: 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. 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). 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 probably the most popular model on computers.

Each color component (sometimes called a *channel*) in a pixel is typically represented with a single byte, eight bits. Eight bits can represent 256 values ($2^8$), which we typically use to represent the values 0 to 255. Each pixel, then, uses 24 bits to represent colors. Using the same formula ($2^{24}$), we know that the standard encoding for color using the RGB model can represent 16,777,216 colors. There are certainly more than 16 million colors in all of creation, but it turns out that it just doesn’t matter—humans have no technology that comes even close to being able
to replicate 16 million distinct colors. Printing technology, color pictures, color monitors—none of these technologies can represent anything close to 16 million colors. So, the 24 bit RGB model is just fine for most purposes.

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, and while they do capture (encode) more color information, we don’t usually need that much information.

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 3.5 highlights this. 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.

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 3.6). The color chooser in Java offers a similar set of sliders (Figure 3.7).

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 3.8 (replicated at Figure 3.43 (page 134) in the color pages at the end of this chapter) 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.

FIGURE 3.5: 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
Section 3.1  How Pictures are Encoded

Figure 3.6: The Macintosh OS X RGB color picker

Figure 3.7: Picking a color using RGB sliders from Java

((150, 255, 150)), so it’s a fairly light green.

Images on disk and even in computer memory are usually stored in some kind 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 nearly 2 million bits. A computer monitor with 1024 pixels across and 768 pixels vertically with 32-bits per pixel takes up a whopping 25 million bits of data or over 3 million bytes of data (A byte is 8 bits).

Table 3.1: Number of bits needed to store pixels at various sizes and formats

<table>
<thead>
<tr>
<th></th>
<th>320x240 image</th>
<th>640x480</th>
<th>1024x768</th>
</tr>
</thead>
<tbody>
<tr>
<td>24-bit color</td>
<td>1,843,200 bits</td>
<td>7,372,800 bits</td>
<td>18,874,368 bits</td>
</tr>
<tr>
<td>32-bit color</td>
<td>2,457,600 bits</td>
<td>9,830,400 bits</td>
<td>25,165,824 bits</td>
</tr>
</tbody>
</table>
3.2 MANIPULATING PICTURES

We manipulate pictures 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 pictures using `new Picture(fileName)`. We make the picture appear with the Picture method `show()`. We can also explore a picture with the method `explore()`.

```java
> String fileName = FileChooser.pickAFile();
> System.out.println(fileName);
C:\intro-prog-java\mediasources\catapillarClipart.jpg
> Picture picture = new Picture(fileName);
> picture.show();
> System.out.println(picture);
Picture, filename
C:\intro-prog-java\mediasources\catapillarClipart.jpg height 181
width 360
```

What `new Picture(fileName)` does is to scoop up all the bytes at 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 picture = new Picture(fileName)`, you are saying “The name picture 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 methods `getWidth()` and `getHeight()`.

```java
> System.out.println(picture.getWidth());
360
> System.out.println(picture.getHeight());
181
```
Section 3.2 Manipulating Pictures

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 increases 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()`.

```java
> Pixel pixel = picture.getPixel(0,0);
> System.out.println(pixel);
Pixel red=255 green=255 blue=255
> Pixel[] pixels=picture.getPixels();
> System.out.println(pixels[0]);
Pixel red=255 green=255 blue=255
```

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

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

Each pixel knows how to `getRed()` and `setRed(redValue)` (Green and blue work similarly.)

```java
> System.out.println(pixel.getRed());
255
> pixel.setRed(0);
> System.out.println(pixel.getRed());
0
```

You can ask a pixel for its color with `getColor()`, and you can ask the pixel to set the color with `setColor(color)`. Color objects know their red, green, and blue components. You can also create new color objects with `new Color(redValue, greenValue, blueValue)`. 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`, for yellow use `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`.

Chapter 3  Encoding and Manipulating Pictures

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 by using 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;

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 color=pixel.getColor();
> System.out.println(color);
java.awt.Color[r=255,g=255,b=255]
> Color newColor=new Color(0,100,0);
> System.out.println(newColor);
java.awt.Color[r=0,g=100,b=0]
> pixel.setColor(newColor);
> System.out.println(pixel.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.

```java
> System.out.println(picture.getPixel(0,0));
```

Pixel red=0 green=100 blue=0
Section 3.2 Manipulating Pictures

One of the important things that you can do with colors is to compare them. Some recipes for manipulating pictures will do different things with pixels depending on the color of the pixel. There are several ways of comparing pictures.

One way of comparing colors is the same way that one would compare numbers. We can subtract the red, green, and blue values of two colors. If we do that, we can create a new color whose red, green, and blue components are the amount of difference between the colors for those components. We can also use `color1.equals(color2)` to compare colors to see if they have the same red, green, and blue values. We would use the operator `==` to test if they are the same color object.

Another method of comparing pictures is with a notion of color distance.

```java
> Color color1 = new Color(10,10,10);
> System.out.println(color1);
java.awt.Color[r=10,g=10,b=10]
> Color color2 = new Color(20,20,20);
> System.out.println(color2);
java.awt.Color[r=20,g=20,b=20]
> int diffRed = Math.abs(color1.getRed() - color2.getRed());
> int diffGreen = Math.abs(color1.getGreen() - color2.getGreen());
> int diffBlue = Math.abs(color1.getBlue() - color2.getBlue());
> Color colorDiff = new Color(diffRed,diffGreen,diffBlue);
> System.out.println(colorDiff);
java.awt.Color[r=10,g=10,b=10]
> System.out.println(color1.equals(color2));
false
> System.out.println(color1.equals(colorDiff));
true
> System.out.println(color1 == colorDiff);
false
```

Another method of comparing pictures is with a notion of color distance.
You often won’t care about an exact match of colors—two shades of blue might be close enough for your purposes.

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:

\[
\sqrt{(\text{red}_1 - \text{red}_2)^2 + (\text{green}_1 - \text{green}_2)^2 + (\text{blue}_1 - \text{blue}_2)^2}
\]

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

```java
> Color testColor = new Color(168,131,105);
> System.out.println(testColor);
java.awt.Color[r=168,g=131,b=105]
> testColor = testColor.darker();
> System.out.println(testColor);
java.awt.Color[r=117,g=91,b=73]
> testColor = testColor.brighter();
> System.out.println(testColor);
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 JColorChooser.

```java
> import java.awt.Color;
> Color pickedColor = ColorChooser.pickAColor();
> System.out.println(pickedColor);
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 `picture.write(fileName)`.

```java
> picture.write("newPicture.jpg");
```

**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 functions just described.

```java
> String fileName = "C:/intro-prog-java/mediasources/catapillarClipart.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();
```

The result showing a small black line on the left side appears in Figure 3.9. The black line is 100 pixels down, and the pixels 10 though 19 from the left edge have been turned black.

![Figure 3.9: Directly modifying the pixel colors via commands: Note the small black line on the left under the leaf](image-url)
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 saw the picture explorer in Figure 3.3 and the MediaTools application appears in Figure 3.10. 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.

The picture explorer can be opened on a picture object. Picture p = new Picture(FileChooser.pickAFile()); will allow you to define a picture 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.

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.)

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 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,
Section 3.3 Changing color values

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 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/catapillarClipart.jpg";
> Picture pict = new Picture(fileName);
> pict.show();
> Pixel currPixel = pict.getPixel(11,100);
> int redValue = currPixel.getRed();
> currPixel.setRed((int) (redValue * 0.5));
> currPixel = pict.getPixel(11,101);
> redValue = currPixel.getRed();
> currPixel.setRed((int) (redValue * 0.5));
> currPixel = pict.getPixel(11,102);
> redValue = currPixel.getRed();
> currPixel.setRed((int) (redValue * 0.5));
> pict.repaint();
```

That’s 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 `for` loop. A `for` loop executes a command or group of commands in a block. A `for` loop continues executing until a continuation test
is false. Each time through the loop one or more variables can be changed.

The syntax for a for loop is:

\[ \text{for (initialization area; continuation test; change area)} \]

Let’s talk through the pieces here.

• First comes the required Java keyword \texttt{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 \texttt{int i=0} which declares a variable \texttt{i} of the primitive type \texttt{int} and initializes it to \texttt{0}. You can initialize more than one variable here by separating the initializations with commas.

• 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 loop will be executed.

• Next comes the required semicolon.

• Next is the change area. Here you usually increment or decrement variables, such as \texttt{i++} to increment \texttt{i}. \texttt{i++} is a shorthand way of saying \texttt{i = i + 1}. You can use either \texttt{i++} or \texttt{i=i+1}, they both will increment the value in \texttt{i} by \texttt{1}. 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).

For example, here is the \texttt{for} loop that simply sets each pixel’s color to black in a picture.

\begin{verbatim}
> import java.awt.Color;
> Pixel[] pixels = pict.getPixels();
> for (int i=0; i<pixels.length; i++)
> pixels[i].setColor(Color.black);
> pict.repaint();
\end{verbatim}

Let’s talk through this code.

• We are using the Color class so we need to either use the fully qualified name (java.awt.Color) or import the Color class using \texttt{import java.awt.Color;}. 

Section 3.3 Changing color values

- Next we declare a variable `pixels` 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.

- Next we have a `for` loop. In the `for` loop initialization area a variable `i` is declared of type int and the value of it is set to 0. The continuation test will be true as long as variable `i` is less than the length of the pixels array. Since arrays are indexed starting at 0 the last element that we want to process is at index (length - 1). The change area will increment the value of `i` each time we finish executing the body of the loop.

- The body of the `for` loop simply tells the current pixel (element of the pixel array at index `i`) to set its color to the color black.

- The statement after the body of the `for` loop will ask the picture object to repaint so that we can see the color change.

```
Debugging Tip: Loops and Variable Declarations
Remember that you can’t declare a variable more than once so you shouldn’t put variable declarations inside loops. Declare any variables that you will need before you start the loop or in the initialization area of the for loop.
```

You don’t actually have to put loops into functions to use them. You can type them into the command area of DrJava. However, you will need to put the whole loop on one line.

```
> for (int i=0; i<pixels.length; i++) pixels[i].setColor(Color.black);
```

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 below recipe reduces the amount of red 50% in the current picture. It uses the variable `p` to stand for the pixel (where we used the variable `pixel` before). It doesn’t matter—the names are merely our choices.

```
Recipe 9: Reduce the amount of red in a picture by 50%

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

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

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

        // set the red value to half what it was
        p.setRed((int) (value * 0.5));
    }
}

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.

Making it Work Tip: Comments in Java

You may notice that there are some interesting characters in the reduceRed recipe. 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 recipe works on a picture object—the one that we’ll use to get the pixels from. To get a picture, we need a filename then we need to make a picture from it. After we ask the picture to decreaseRed(), we’ll want to repaint the picture to
Section 3.3 Changing color values

FIGURE 3.11: The original picture (left) and red-reduced version (right)

see the effect. Therefore, the decreaseRed recipe can be used like this:

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

Common Bug: Patience: for 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-reduced version appear in Figure 3.11 (and at Figure 3.44 on page 135). 50% is obviously a lot of red to reduce! The picture looks like it was taken through a blue filter.

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 reduce red and see how it worked. We want to break in at the point where we just called decreaseRed()

> String fileName = "C:/intro-prog-java/mediasources/catapillarClipart.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`.

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

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

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

        // set the red value to half what it was
        p.setRed((int) (value * 0.5));
    }
}
```

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

- The `Pixel[] pixels =` is a declaration of a variable 'pixels' that references an array of `Pixel` objects. The '=' means that the variable `pixels` will be initialized to the result of the right side expression which is `this.getPixels()`.

- 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.
Section 3.3 Changing color values

So at the end of the first line we have a variable pixels that is referring 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 reduceRed().

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 Pixel p = null; We start it off referring to nothing by using the keyword 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. 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 p and value are completely different than the names in the Command Area or any other method. We say that they have a different scope. The scope of a variable is the area of a program 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 initialization area of a for loop are only known until the end of the body of the for loop.

Next comes the for loop for (int i = 0; i < pixels.length; i++). This declares a variable i of the type int and initializes it to 0. It tests that the value of i is less than the length of the pixels array and if so will execute the statements in the body of the loop.

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

```
pixels p, value = 0, i = 0
```

Next in the body of the loop is \( value = p\.getRed() \). This sets the variable value to the amount of red in the current pixel.

```
pixels p, value = 255, i = 0
```

Next in the body of the loop is \( p\.setRed((\text{int}) (value * 0.5)) \). This changes the amount of red in the pixel to half of its original value. The \((\text{int}) (value * 0.5)\) is needed because the variable value is declared of type int and yet the calculation of \((value * 0.5)\) 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 int. So, the compiler won’t let us do this without telling it that we really want it to by including the \((\text{int})\). This is called casting and is required whenever a larger value is being placed into a smaller variable. So if the result of the multiplication has a fractional part that fractional part will just be thrown away so that the result can fit in an int.

```
pixels p, value = 255, i = 0
```

After the statements in the body of the loop are executed the \( i++ \) will be executed which will add one to the current value of i. The value of i will change to 1.

```
pixels p, value = 255, i = 1
```

What happens next is very important. The loop starts over again. The continuation test will check that the value in i is less than the length of the pixels array and since the value of i is less the statements in the body of the loop will be executed again. The variable p will be set to the pixel object in the pixels array at index 1.
Section 3.3 Changing color values

The value variable is updated to get the red value for the pixel pointed to by p (this happens to have the same value as the previous pixel but it could have changed).

At the end of each loop body the commands in the change area will be executed again (so the value in i will increment). Next, the continuation 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 3.11 (and at Figure 3.44 on page 135). We keep going through all the pixels in the sequence and changing all the red values.

Testing the program: Did that really work?.

How do we know that that really worked? Sure, something happened to the 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 tell them. 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(FileChooser.pickAFile()); and Picture
Chapter 3  Encoding and Manipulating Pictures

FIGURE 3.12: Using the picture explorer to convince ourselves that the red was decreased

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 Command Area to check the red values of individual pixels.

> String fileName = "C:/intro-prog-java/mediasources/catapillarClipart.jpg";
> Picture pict = new Picture(fileName);
> Pixel pixel = pict.getPixel(0,0);
> System.out.println(pixel);
Pixel red=255 green=255 blue=255
> pict.decreaseRed();
> Pixel newPixel = pict.getPixel(0,0);
> System.out.println(newPixel);
Pixel red=127 green=255 blue=255
> System.out.println( 255 * 0.5);
127.5

Increasing red.

Let’s increase the red in the picture now. If multiplying the red component by 0.5 reduced 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 3.13 and Figure 3.45).

Recipe 10: Increase the red component by 30%

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

Section 3.3 Changing color values

Figure 3.13: Overly blue (left) and red increased by 30% (right)

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

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

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

End of Recipe 10

We can even get rid of a color completely. The below recipe erases the blue component from a picture by setting the blue value to 0 in all pixels (Figure 3.14 and Figure 3.46).

Recipe 11: 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[] pixels = this.getPixels();
    Pixel pixel = null;

    // loop through all the pixels
```
Chapter 3  Encoding and Manipulating Pictures

FIGURE 3.14: Original (left) and blue erased (right)

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

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

3.3.3 Creating a sunset

We can certainly do more than one picture manipulation at once. One that I wanted to do was to try to generate a sunset out of a beach scene. My first attempt was to increase the red, but that doesn’t work. Some of the red values in a given picture are pretty high. If you go past 255 for a channel value, you wrap-around. If you setRed of a pixel to 256, you’ll actually get zero! So, increasing red created bright blue-green (no red) spots.

My 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 I wrote for that:

/** *
 * Method to simulate a sunset by reducing the green and blue *
 */
public
public void makeSunset()
{

Section 3.3  Changing color values  71

FIGURE 3.15: Original beach scene (left) and at (fake) sunset (right)

```java
Pixel[] pixels = this.getPixels();
Pixel pixel = null;
int value = 0;

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

End of Recipe 12

What we see happening in Recipe 12 is that we’re changing both the blue and green channels—reducing each by 30%. The effect works pretty well, as seen in Figure 3.15 (and in the color section at Figure 3.47).

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?

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 ex-
ample might make that clearer. Consider the recipe to make a sunset (Recipe 12).
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 Recipe 13.

```
Recipe 13: Making a sunset as three methods

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

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

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

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

/**
 * Method to reduce the blue in the picture by 30%
 */
public void reduceBlue()
{
    Pixel[] pixels = this.getPixels();
    Pixel pixel = null;
    int value = 0;
```
Section 3.3  Changing color values

// loop through all the pixels
for (int i = 0; i < pixels.length; i++)
{
    // get the current pixel
    pixel = pixels[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 makeSunset()
{
    reduceGreen();
    reduceBlue();
}

End of Recipe 13

The first thing to note is that this actually does work. makeSunset() does the same thing here as in the previous recipe. It’s perfectly okay to have one method (makeSunset() in this case) use other methods in the same class (reduceBlue() and reduceGreen()). You use makeSunset() just as you had before. It’s the same recipe (it tells the computer to do the same thing), but with different methods. The earlier recipe did everything in one method, and this one does it in three. In fact, you can also use reduceBlue() and reduceGreen()—make a picture in the Command Area and pass it as input to either of them. They work just like decreaseRed().

What’s different is that the function 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 they more effectively communicate process to other humans. Notice that first we had a method called decreaseRed() that reduced the red in the picture by 50%. Later we added reduceBlue() that reduced the blue in the picture by 30%. Wouldn’t it be better to use similar names for methods that do similar things? We could rename the method that reduced the red as reduceRed(). What about the differences in the amount of reduction? Should we call the method that reduces the red 50% reduceRed50Percent?

What if we had written reduceBlue() and reduceGreen() so that each asked you to pick a file and created the picture before changing the color. We would be asked for the picture twice—once in each function. Because we wrote these functions to only reduce the blue and reduce the green (“one and only one thing”), we can use them in new functions like makeSunset().

Now, let’s say that we asked you to pick a picture and created the picture in makeSunset() before calling the other methods. The methods reduceBlue() and reduceGreen() are completely flexible and reusable again. But 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, 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 send each of these pictures the message makeSunset(). 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.

\footnote{There is an issue that the new 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 first, and worry about efficiency later.}
Section 3.3  Changing color values

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 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.

We could also write these functions with explicit filenames, by saying at the beginning of one of the programs:

```java
String fileName = "C:/intro-prog-java/mediasources/catapillarClipart.jpg";
```

That way, we wouldn’t get prompted for a file each time. But then the methods only work for the one file, and if we want them to work for some other file, we have to modify the function. Do you really want to change the method each time you use it? It’s easier to leave the method alone, and change the picture that you use to invoke the method.

Of course, we could change any of our methods to be handed a filename rather than a picture. For example, we could write:

```java
/**
 * Method to create a picture from the passed file name and 
 * reduce the green in it 
 * @param fileName the name of the file to create the picture from 
 * @return the created picture 
 */
public static void reduceGreen(String fileName)
{
    Picture picture = new Picture(fileName);
    Pixel[] pixels = picture.getPixels();
    Pixel pixel = null;
    int value = 0;

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

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

        // set the green value to 70% of what it was
        pixel.setGreen((int) (value * 0.7));
    }
}
```
Chapter 3 Encoding and Manipulating Pictures

```java
/**
 * Method to create a picture and reduce the blue in it
 * @param fileName the name of the file to use to create the picture
 * @return the created picture
 */
public static void reduceBlue(String fileName)
{
    Picture picture = new Picture(fileName);
    Pixel[] pixels = picture.getPixels();
    Pixel pixel = null;
    int value = 0;

    // loop through all the pixels
    for (int i = 0; i < pixels.length; i++)
    {
        // get the current pixel
        pixel = pixels[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 simulate a sunset by reducing the green and blue
 * @param fileName the name of the file to use
 */
public static void makeSunset(String fileName)
{
    reduceGreen(fileName);
    reduceBlue(fileName);
}
```

Notice that the methods are now class methods (because they have the static keyword in the method declarations). They are class methods because they no longer operate on the current picture object. Instead they create a picture object inside the method and work on that.

Is this better or worse than the code we saw before? At some level, it doesn’t matter—we can work with pictures or filenames, whichever makes more sense to us. The filename version does have several disadvantages, though. For one, it doesn’t work—the picture object gets made in each of reduceGreen(fileName) and reduceBlue(fileName), but then it doesn’t get saved, so it gets lost at
end of the function. We could fix that by saving the file out to disk after we’re done with each method, but then the functions are doing more than “one and only one thing.” There’s also the inefficiency of making the picture twice, and if we were to add in the saving, saving the picture twice. Again, the best functions do “one and only one thing.” Also in doing object-oriented programming the idea is to create objects and have the objects do the work. It isn’t a good idea put most of the work into class methods.

Even larger methods, like `makeSunset()`, do “one and only one thing.” `makeSunset()` makes a sunset-looking picture. It does that by reducing green and reducing blue. It calls two other methods to do that. What we end up with is a hierarchy of goals—the “one and only one thing” that is being done. `makeSunset()` does its one thing, by asking two other methods to do their one thing. We call this 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.

Names in methods are completely separate from names 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 did a `new Picture(fileName)` inside a method, you might want to output the picture object that you created inside the function. You can do that by using the return keyword, which we’ll talk more about later.

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 simulate a sunset by reducing the green and blue
 * @param fileName the name of the file to use
 */
public static void makeSunset(String fileName)
{
    reduceGreen(fileName);
    reduceBlue(fileName);
```
We are going to call `makeSunset()` with a statement like `// makeSunset("C:/intro-prog-java/mediasources/catapillarClipart.jpg")`.

Whatever string is in `// makeSunset("C:/intro-prog-java/mediasources/catapillarClipart.jpg")` becomes known as `fileName` while `makeSunset()` is running.

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 this version of `decreaseRed`:

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

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

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

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

We would use this one by saying something like `myPicture.decreaseRed(0.25)`. That particular use would reduce the red by 75%. We could say `myPicture.decreaseRed(1.25)` and increase red by 25%. Perhaps this function should be better named `changeRed`, because that's what it is now—a general way of changing the amount of red for every pixel in a picture. That's a pretty useful and powerful function.

Recall seeing in Recipe 11 this code:

```java
/**
 * Method to clear the blue from the picture (set
 * the blue to 0 for all pixels)
 */
public void clearBlue()
{
    Pixel[] pixels = this.getPixels();
```
Section 3.3  Changing color values

Pixel pixel = null;

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

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

We could also write that same recipe like this:

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

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

It’s important to note that this function achieves the exact same thing as the earlier recipe 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 recipe isn’t necessarily better.

3.3.5 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 recipes. Figure 3.16 (Figure 3.48) shows the lighter and darker versions of the original picture seen earlier.

Recipe 14: Lighten the picture

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

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

End of Recipe 14

Recipe 15: Darken the picture

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

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

        // get the current color
Section 3.3  Changing color values  81

3.3.6 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 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 recipe that does it, and you can see even from the grayscale image that it really does work (Figure 3.17 and Figure 3.49).

///
/**
 * Method to negate the picture
Chapter 3  Encoding and Manipulating Pictures

ERSIST

*/
public void negative()
{
    Pixel[] pixels = this.getPixels();
    Pixel pixel = null;
    int redValue, blueValue, greenValue = 0;

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

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

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

End of Recipe 16

FIGURE 3.17: Negative of the image

3.3.7 Converting to grayscale

Converting to grayscale is a fun recipe. 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 recipe and Figure 3.18 (and Figure 3.50 on page 136).

**Recipe 17: Convert to grayscale**

```java
/**
 * Method to change the picture to gray scale
 */
public void grayscale()
{
    Pixel[] pixels = this.getPixels();
    Pixel pixel = null;
    int intensity = 0;

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

**End of Recipe 17**

This is an overly simply notion of grayscale. Below is a recipe 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.

**Recipe 18: Convert to grayscale with more careful control of luminance**
**
* Method to change the picture to gray scale with luminance
*/
public void grayscaleWithLuminance()
{
    Pixel[] pixels = 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 < pixels.length; i++)
    {
        // get the current pixel
        pixel = pixels[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));
    }
}
3.4 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 process only some of the pixels in a picture (say, just the red in someone’s eyes, but not the red in her dress), we need to control which pixels we’re manipulating.

3.4.1 Looping across the pixels with a nested loop

Unlike sounds and samples (as we’ll see later), we can’t use just a single `for` loop if we want to address every pixel. We have to use two `for` loops—one to move horizontally across the pixels, 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
        pixel = getPixel(x,y);

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

For example, here’s Recipe 14 (page 79), but using explicit pixel references.

Recipe 19: Lighten the picture using nested loops

```java
/**
 * Method to lighten the colors in the picture
 */
public void lighten()
{...}
Chapter 3  Encoding and Manipulating Pictures

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);
    }
}

End of Recipe 19

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

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

2. java.awt.Color color = null; and Pixel pixel = null; declare the vari-
   ables 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. We can declare them once before the loop and reuse
them each time through the loop.

3. for (int x = 0; x < getWidth(); x++) declares a variable x of 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. for (int y = 0; y < getHeight(); y++) This declares a variable y of type
int which will be initialized to 0. The continuation condition 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. \( \text{pixel} = \text{getPixel}(x, y); \) This sets the variable \( \text{pixel} \) to refer to the Pixel object at the given \( x \) and \( y \) location in the picture.

6. \( \text{color} = \text{pixel}.\text{getColor}(); \) This sets the variable \( \text{color} \) to refer to the Color object at the current pixel.

7. \( \text{color} = \text{color}\.\text{brighter}(); \) This creates a new lighter (brighter) color object based on the original color object and sets the variable \( \text{color} \) to refer to that new color object.

8. \( \text{pixel}.\text{setColor}()\text{color}; \) This 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 then execution will move 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 then execution will continue at the statement following the body of the loop.

### 3.4.2 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 \( \text{mirrorPoint} \)—halfway across the picture, \( (\text{int}) (\text{picture}.\text{getWidth}()) / 2 \). (We want this to be an integer, a whole number, so we’ll apply \( (\text{int}) \) to it.) We’ll have the \( x \) value increment from 1 to the \( \text{mirrorPoint} \). At each value of \( x \), we want to copy the color at the pixel \( x \) pixels to the left of the \( \text{mirrorPoint} \) to the pixel \( x \) pixels to the right of the \( \text{mirrorPoint} \). The left would be \( \text{mirrorPoint} - x \) and the right would be \( \text{mirrorPoint} + x \). Take a look at Figure 3.19 to convince yourself that we’ll actually reach every pixel using this scheme. Here’s the actual recipe.

---

**Recipe 20: Mirror pixels in a picture along a vertical line**

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

* Method to mirror around a vertical line in the middle of the picture
  * based on the width

```java
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 3.20.

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

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 copy have $x$ range from 0 to less than the mirrorPoint and copy it to ($width - 1 - x$).
Can we mirror horizontally? Sure!

**Recipe 21: 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());
        }
    }
}
```
Chapter 3 Encoding and Manipulating Pictures

Now this last recipe copies from the top of the picture onto the bottom (see Figure 3.21). 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 3.21).

**Recipe 22: 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());
        }
    }
}
```

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());
        }
    }
}
Section 3.4  Copying pixels

{  
  topPixel = getPixel(x,(mirrorPoint - y));  
  bottomPixel = getPixel(x,(mirrorPoint + y));  
  topPixel.setColor(bottomPixel.getColor());  
}

End of Recipe 22

FIGURE 3.21: Santa mirrored horizontally, bottom to top (left) and top to bottom (right)

Mirroring usefully.

While mirroring is probably mostly used for interesting effects, occasionally it has some more serious (but still fun!) purposes. I took a picture of the Temple of Zeus in the ancient agora in Athens, Greece, when traveling to a conference (Figure 3.22). By sheer luck, I got the pediment dead horizontal. The Temple of Zeus had its pediment damaged. I wondered if I could “fix” it by mirroring the good part onto the broken part.

I used the picture explorer to figure out the range of values where I would need to do the mirroring and the point where I should mirror (Figure 3.23). The function I wrote to do the repair is below, and the final picture is in (Figure 3.24)–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.

Recipe 23: Mirror the Temple of Zeus
Chapter 3 Encoding and Manipulating Pictures

FIGURE 3.22: Temple of Zeus from the ancient agora in Athens, Greece

FIGURE 3.23: Coordinates where we need to do the mirroring

```java
/**
 * Method to mirror the pediment 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
```
Section 3.4  Copying pixels  93

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;

End of Recipe 23

In this recipe, we’re using Picture.getMediaPath(fileName). The function Picture.getMediaPath(fileName) is a shorthand. If you keep your media in one place, and you’d like to refer to it just by its base name, you can use Picture.getMediaPath(fileName), which actually just generates a complete path for you. However, before you use it you should Picture.setMediaPath(directory) first! Picture.setMediaPath(directory) lets you specify that place (directory) where you store your media. Picture.setMediaPath(directory) tells Picture.getMediaPath(fileName) what directory to use to construct the full path name.

FIGURE 3.24: The manipulated temple

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.
Chapter 3  Encoding and Manipulating Pictures

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.

```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;
    int count = 0;

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

    // tell how many pixels were copied
    System.out.println("We copied " + count + " pixels");

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

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 j 97) which is 96 - 27 + 1). We copy 262 columns of pixels (x goes from 1 to j 263 (276 - 13) which is 262 - 1 + 1 = 262). 70 * 262 is 18,340.
3.5 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 make use of the utility function `Picture.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 `Picture.setMediaPath(directory)` first! All that `Picture.getMediaPath(fileName)` does is to prepend the directory found in `Picture.setMediaPath(directory)` to the input filename.

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

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

3.5.1 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 recipe for copying a picture of Katie to the canvas.

**Recipe 24: Copying a picture to a canvas**
Section 3.5  Copying and transforming pictures  97

/**
 * 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 = Picture.getMediaPath("KatieFancy.jpg");
    Picture sourcePicture = new Picture(sourceFile);
    String targetFile = Picture.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;
}

End of Recipe 24

This program copies a picture of Katie to the canvas (blank picture) (Figure 3.25). 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, if you separate 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:

```java
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.

```java
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.

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
Section 3.5  Copying and transforming pictures

Figure 3.26: Copying a picture midway into a canvas to change where the target X and Y coordinates start. The rest stays exactly the same (Figure 3.26).

Recipe 25: 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 = Picture.getMediaPath("KatieFancy.jpg");
    Picture sourcePicture = new Picture(sourceFile);
    String targetFile = Picture.getMediaPath("7inx95in.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++)
        {
            sourcePixel = sourcePicture.getPixel(sourceX, sourceY);
            targetPixel = targetPicture.getPixel(targetX, targetY);
            targetPixel.set(sourcePixel); // Copy pixel value
        }
    }
    return targetPicture;
}
```
Chapter 3 Encoding and Manipulating Pictures

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;

End of Recipe 25

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 what the coordinates are where her face is located, then use those on the dimensions of sourceX and sourceY (Figure 3.27). The face is at (70, 3) to (136, 81).

Recipe 26: Cropping a picture onto a canvas

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

    // loop through the columns
Section 3.5 Copying and transforming pictures

FIGURE 3.27: Copying part of a picture onto a canvas

for (int sourceX = 70, targetX = 100;
     sourceX < 135; sourceX++, targetX++)
{
    // loop through the rows
    for (int sourceY = 3, targetY = 100;
            sourceY < 80; 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;
How does that work? Let’s look at a small example to see what’s going on in the copying recipe. 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.

3.5.2 Creating a Collage

In the mediasources folder are a couple images of flowers (Figure 3.28), 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.
Recipe 27: Creating a collage

/**
 * 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(Picture.getMediaPath("flower1.jpg"));
    Picture flower2Picture = new Picture(Picture.getMediaPath("flower2.jpg"));
    Picture canvasPicture = new Picture(Picture.getMediaPath("640x480.jpg"));

    // declare the source and target pixel variables
    Pixel sourcePixel = null;
    Pixel targetPixel = null;

    // 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() - flower1Picture.getHeight() - 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() - flower2Picture.getHeight() - 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.negative();
for (int sourceX = 0, targetX = 200;
    sourceX < flower1Picture.getWidth();
    sourceX++, targetX++)
{
    for (int sourceY = 0,
         targetY = canvasPicture.getHeight() - flower1Picture.getHeight() - 5;
         sourceY < flower1Picture.getHeight();
         sourceY++, targetY++)
    {
        sourcePixel = flower1Picture.getPixel(sourceX,sourceY);
        targetPixel = canvasPicture.getPixel(targetX,targetY);
        targetPixel.setColor(sourcePixel.getColor());
Section 3.5  Copying and transforming pictures

// 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() - flower2Picture.getHeight() - 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() - flower1Picture.getHeight() - 5;
         sourceY < flower1Picture.getHeight();
         sourceY++, targetY++)
    {
       sourcePixel = flower1Picture.getPixel(sourceX,sourceY);
       targetPixel = canvasPicture.getPixel(targetX,targetY);
       targetPixel.setColor(sourcePixel.getColor());
    }
}

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

End of Recipe 27

Here’s how we run the collage(Figure 3.29):
> Picture flowerCollage = Picture.createCollage();
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 them 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. We can make a new method which copies a passed picture object into the current picture object starting at a passed x value.

/**
 * 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(Picture.getMediaPath("flower1.jpg"));
    Picture flower2Picture = new Picture(Picture.getMediaPath("flower2.jpg"));
    Picture canvasPicture = new Picture(Picture.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);
3.5.3 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 recipe below, we blend a picture of the two sisters with an overlap of some 70 (the width of Barbara minus 150) columns of pixels (Figure 3.30).

Recipe 28: Blending two pictures

```java
/**
 * Method to blend pictures of Katie and Jenny
 * @return the blended picture
 */
public static Picture blendPictures()
{
    // create the three pictures
    Picture katiePicture = new Picture(Picture.getMediaPath("KatieFancy.jpg"));
    Picture jennyPicture = new Picture(Picture.getMediaPath("JenParty.jpg"));
    Picture canvasPicture = new Picture(Picture.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;

// 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);
        targetPixel.setColor(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)));
    }
}

// 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++)
    {

FIGURE 3.30: Blending the picture of Katie and Jenny

```java
jennyPixel = jennyPicture.getPixel(sourceX,sourceY);
targetPixel = canvasPicture.getPixel(targetX,targetY);
targetPixel.setColor(jennyPixel.getColor());
}
}

// show the canvas
canvasPicture.show();

// return the canvas
return canvasPicture;
```
3.5.4 Rotation

Transformations to the image occur by using the index variables differently or incrementing them differently, but otherwise keeping the same recipe. Let’s rotate Katie 90 degrees. We’ll do that by simply swapping the X and Y variables in the target—we increment them the exact same way, but we’ll use them X for Y and Y for X (Figure 3.31).

Recipe 29: Rotating a picture

```java
/**
 * Method to copy Katie rotated to the left 90 degrees
 * @return the picture after Katie has been copied and rotated to the left 90
 */
public static Picture copyKatieSideways()
{
    String sourceFile = Picture.getMediaPath("KatieFancy.jpg");
    Picture sourcePicture = new Picture(sourceFile);
    String targetFile = Picture.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++)
            
```
Chapter 3 Encoding and Manipulating Pictures

```java
{ // set the target pixel color to the source pixel color
    sourcePixel = sourcePicture.getPixel(sourceX, sourceY);
    targetPixel = targetPicture.getPixel(targetY, targetX);
    targetPixel.setColor(sourcePixel.getColor());
}

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

return targetPicture;
}
```

End of Recipe 29

**FIGURE 3.31:** 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.
Section 3.5  Copying and transforming pictures  113

Now, as we increment the Y variables, we’re moving down the source, but across the target. As we increment the X variables we’re moving across the source but down the target.

When we’re done, we’ve done the same copy, but the result is completely different.

3.5.5 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 place 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 daisyMed.jpg picture which is 302 (width) by 202 (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 302/2 and the height will be...
202/2. The result is a smaller flower in the canvas (Figure 3.32).

### Recipe 30: 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(Picture.getMediaPath("daisyMed.jpg"));
    Picture canvasPicture = new Picture(Picture.getMediaPath("640x480.jpg"));
    Pixel sourcePixel = null;
    Pixel targetPixel = null;

    // 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;
}
```

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...
FIGURE 3.32: Scaling the picture down

The result is a larger form of the picture (Figure 3.33).

Recipe 31: Scaling the picture up (larger)

```java
/**
 * Method to copy a flower but scaled to 2x normal size
 * @return the larger flower
 */
public static Picture copyFlowerLarger()
{
    Picture flowerPicture = new Picture(Picture.getMediaPath("daisyMed.jpg"));
    Picture canvasPicture = new Picture(Picture.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;

End of Recipe 31

FIGURE 3.33: Scaling up a picture

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 \texttt{new Picture(width, height)} and creates a picture of a desired width and height (both specified in pixels). \texttt{new Picture(640, 480)} would create a picture object that is 640 pixels wide by 480 pixels tall—just like the canvas.

\textbf{How did that work?}

We start from the same place as the original copy.

When we increment \texttt{sourceY} by 0.5, we end up referring to the same pixel in the source, but the target has moved on to the next pixel.
When we increment `sourceY` a second time by 0.5, we now move on to the next pixel, which we’ll end up copying the same pixel twice.

And eventually, we cover every pixel. Notice that the end result is degraded—it’s choppier than the original.

### 3.6 REPLACING COLORS

Replacing colors with another color is pretty easy. We can do it broadly, or just within a range.

Here’s a recipe that tries to replace the brown color in Katie’s hair with red. I 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 increase the redness of those pixels. I played a lot with the value that I used for distance (here, 50.0) and the amount of redness increase (here, 100% increase). However, it turned part of the couch and carpet red too. (Figure 3.34 and Figure 3.51).

**Recipe 32: Color replacement: Turn Katie into a redhead**

```java
/**
   * Method to turn to turn Katie into a red head
   */
public static Picture turnKatieRedHead()
```
Chapter 3 Encoding and Manipulating Pictures

```java
{
    Color brown = new Color(42, 25, 15);
    Color currentColor = null;
    Picture katiePicture = new Picture(Picture.getMediaPath("KatieFancy.jpg"));
    Pixel[] pixels = katiePicture.getPixels();
    Pixel pixel = null;

    // 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 reduce blue and green
        if (pixel.colorDistance(brown) < 50.0)
            pixel.setColor(new Color((int) (currentColor.getRed() * 2.0), currentColor.getGreen(), currentColor.getBlue()));
    }

    // show the result
    katiePicture.show();

    return katiePicture;
}
```

Recipe 32: Color replacement in a range

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 3.35 and Figure 3.52).

Recipe 33: 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);
    Color currentColor = null;
    Picture katiePicture = new Picture(Picture.getMediaPath("KatieFancy.jpg"));
    Pixel pixel = null;
```
FIGURE 3.34: Increasing reds in the browns

// 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 reduce blue and green
        if (pixel.colorDistance(brown) < 50.0)
            pixel.setColor(new Color((int) (currentColor.getRed() * 2.0),
                                       currentColor.getGreen(),
                                       currentColor.getBlue()));

    }
}

// show the result
katiePicture.show();
return katiePicture;
}

End of Recipe 33

FIGURE 3.35: Increasing reds in the browns, within a certain range

3.6.1 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” (we use a distance from red of 165 works well) to red, then insert a replacement color.

We probably don’t want to change the whole picture. In the Figure 3.36, 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 MediaTools, we find the upper left and lower right corners of her eyes. Those points were (109, 91) and (202, 107).

Recipe 34: Remove red eye
Section 3.6 Replacing Colors

FIGURE 3.36: Finding the range of where Jenny’s eyes are red

```java
/**
 * Method to remove red eye from the current picture object in the rectangle
 * define 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);
            }
        }
    }
}
```
End of Recipe 34

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.show();
```

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 3.37). (See also Figure 3.53.)

![Figure 3.37: After fixing red-eye.](image)

### 3.6.2 Sepia toned and posterized pictures: Using conditionals to choose the color

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, and choosing to replace with some function of the original color or by changing to a specific color. The results are quite interesting.

For example, we might want to generate sepia-toned prints. Older prints sometimes have a yellow-ish 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, shadows–and treating them differently, we can get a better effect (Figure 3.38).

The way we do this is to first convert everything to grayscale, both because older prints were in a gray scale, and because it makes it a little easier to work with. We then look for high 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 3.6 Replacing Colors

Recipe 35: Convert a picture to sepia-tones

/**
 * 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)
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 3.39).
Recipe 36: Posterizing a picture

/**
 * Method to posterize (reduce the number of colors) in the picture
 * The number of reds, greens, and blues will be 4
 */
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);

End of Recipe 36

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. Below is the recipe for a flexible number of
levels, and Figure 3.40 shows a couple of examples.

Recipe 37: Posterize by levels

/**
 * 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++) {
            // check for red range
            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);
        }
    }
}
// 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);
}
}

End of Recipe 37

FIGURE 3.40: Pictures posterized to two levels (left) and four levels (right)
3.7 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 side of the pixel. It takes a picture, and a number (size) of pixels to compute the average.

Recipe 38: 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()) {
                        samplePixel = this.getPixel(xSample,ySample);

```
Section 3.7 Combining pixels: Blurring

```java
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);
}
}

End of Recipe 38

Figure 3.41 shows the flower from the collage made bigger, then blurred. You can see the pixellation in the bigger version—the sharp, blocky edges. With the blur, some of that pixellation 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 pixellation without removing sharpness.

FIGURE 3.41: Making the flower bigger, then blurring to reduce pixellation
Chapter 3  Encoding and Manipulating Pictures

FUNCTIONS AND OBJECTS SUMMARY

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

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

Here are the functions used or introduced in this chapter:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileChooser.pickAFile()</td>
<td>Lets the user pick a file and returns the complete path name as a string. Takes no input.</td>
</tr>
<tr>
<td>new Picture(fileName)</td>
<td>Takes a filename as input, reads the file, and creates a picture object from it. Returns the picture object.</td>
</tr>
<tr>
<td>picture.show()</td>
<td>Must be called on a Picture object. Shows the picture object. No return value.</td>
</tr>
<tr>
<td>picture.getPixels()</td>
<td>Must be called on a Picture object. Returns an array of Pixel objects in the picture.</td>
</tr>
<tr>
<td>picture.getPixel()</td>
<td>Must be called on a Picture object. It takes an (x) position and a (y) position (two numbers), and returns the Pixel object at that point in the picture.</td>
</tr>
<tr>
<td>picture.getWidth()</td>
<td>Must be called on a picture object. Returns the width of the picture in pixels.</td>
</tr>
<tr>
<td>picture.getHeight()</td>
<td>Must be called on a picture object. It returns the height in pixels.</td>
</tr>
<tr>
<td>picture.writePictureTo(fileName)</td>
<td>Must be called on a picture object. It takes a file name (string) as input, then writes the picture 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>
### Section 3.7 Combining pixels: Blurring

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

<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>new Color(red,green,blue)</td>
<td>Takes three inputs: For the red, green, and blue components (in order), then creates and returns a color object.</td>
</tr>
<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 of what you picked.</td>
</tr>
<tr>
<td>pixel.colorDistance(color)</td>
<td>Takes a Color object and returns a single number representing the distance between the color and the current pixel’s color. The red, green, and blue values of the colors are taken as a point in ((x, y, z)) space, and the cartesian distance is computed.</td>
</tr>
<tr>
<td>color.darker(), color.brighter()</td>
<td>Must be called on a Color object. The methods return a slightly darker or lighter (respectively) version of the color.</td>
</tr>
</tbody>
</table>

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.

### PROBLEMS

**3.1.** Recipe 9 (page 61) 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.

**3.2.** Write the blue and green versions of Recipe 9 (page 61).

**3.3.** Each of the below is equivalent to Recipe 10 (page 68). 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()
{
```
Chapter 3 Encoding and Manipulating Pictures

Pixel[] pixels = this.getPixels();

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

/**
 * Method to increase the amount of red by 1.3
 */
public void increaseRed3()
{
    Pixel[] pixels = this.getPixels();
    Pixel pixel = null;
    int red = 0;
    int green = 0;
    int blue = 0;
    int newRed = 0;

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

3.4. If you keep increasing the red, eventually the red looks like it disappears, and you eventually get errors about illegal arguments. What do you think is going on?

3.5. Write new methods like Recipe 11 (page 69) to clear red and green. For each of these, which would be the most useful in actual practice? How about combinations of these?

3.6. Write a new method to maximize blue (i.e., setting it to 255) instead of clearing it use Recipe 11 (page 69) as a starting point. Is this useful? Would the red or green versions be useful?

3.7. There is more than one way to compute the right grayscale value for a color value. The simple recipe that we use in Recipe 17 (page 83) may not be what your grayscale printer uses when printing a color picture. Compare the color (rel-
Section 3.7  Combining pixels: Blurring

Atively unconverted by the printer) grayscale image using our simple algorithm in Figure 3.50 with the original color picture that the printer has converted to grayscale (left of Figure 3.11). How do the two pictures differ?

3.8. Write a method to do mirroring along the diagonal (from \((0,0)\) to \((\text{width} - 1, \text{height} - 1)\)?

3.9. 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* [7]. It’s highly recommended.

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 [11].
3.8 COLOR FIGURES

FIGURE 3.42: Merging red, green, and blue to make new colors

FIGURE 3.43: Color: RGB triplets in a matrix representation
FIGURE 3.44: Color: The original picture (left) and red-reduced version (right)

FIGURE 3.45: Color: Overly blue (left) and red increased by 30% (right)

FIGURE 3.46: Color: Original (left) and blue erased (right)

FIGURE 3.47: Original beach scene (left) and at (fake) sunset (right)
Chapter 3  Encoding and Manipulating Pictures

FIGURE 3.48: Color: Lightening and darkening the original picture

FIGURE 3.49: Color: Negative of the image

FIGURE 3.50: Color: Color picture converted to grayscale
FIGURE 3.51: Color: Increasing reds in the browns
FIGURE 3.52: Color: Increasing reds in the browns, within a certain range

FIGURE 3.53: Finding the range where Jenny’s eyes are red, then changing them to black
Section 3.8  Color Figures  139

FIGURE 3.54: Frames from the slow sunset movie

FIGURE 3.55: Frames from the slow fade-out movie
Chapter 3  Encoding and Manipulating Pictures

FIGURE 3.56: Frames from the Mommy watching Katie movie

FIGURE 3.57: Frames from the original too dark movie
Section 3.8 Color Figures 141

FIGURE 3.58: Frames from the modified lighter movie

FIGURE 3.59: Frames from the original movie with kids crawling in front of a blue screen
FIGURE 3.60: Frames from the kids on the moon movie
CHAPTER 4

Advanced Pictures

4.1 BACKGROUND SUBTRACTION
4.2 CHROMAKEY
4.3 DRAWING ON IMAGES WITH PIXELS
4.4 DRAWING WITH DRAWING COMMANDS
4.5 PROGRAMS AS SPECIFYING DRAWING PROCESS

In this chapter, we’ll talk about techniques for creating pictures. We’ll compose pictures in new ways (essentially, pulling someone out and putting them in a new setting) and create pictures from scratch without explicitly setting every pixel.

4.1 BACKGROUND SUBTRACTION

Let’s imagine that you have a picture of someone, and a picture of where they stood without them there (Figure 4.1). 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 4.2)?

FIGURE 4.1: A picture of a child (Katie), and her background without her

Recipe 39: Subtract the background and replace it with a new one

/**
 * 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
 */
FIGURE 4.2: A new background, the moon

* @param newBackground a picture with the new background to use
*/
public void swapBackground(Picture oldBackground, Picture newBackground)
{
    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 color at the current pixel is within 15.0 of the old background pixel
            /* 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(Picture.getMediaPath("kid-in-frame.jpg"));
> Picture oldBg = new Picture(Picture.getMediaPath("bgframe.jpg"));
> Picture newBg = new Picture(Picture.getMediaPath("moon-surface.jpg"));
> p.swapBackground(oldBg, newBg);
> p.show();
```

We can, but the effect isn't as good as we would like (Figure 4.3). 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.

![Katie on the moon](image)

**FIGURE 4.3:** Katie on the moon

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 4.4) 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 4.5).

```java
> Picture p = new Picture(Picture.getMediaPath("wall-two-people.jpg"));
> Picture oldBg = new Picture(Picture.getMediaPath("wall.jpg"));
> Picture newBg = new Picture(Picture.getMediaPath("jungle3.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.

Recipe 40: Better Swap Background

```java
/**
 * Method to replace the background in the current picture with the background from another picture for pixels that have a color distance to the old background of under the passed threshold
 * @param oldBackground a picture with the old background to replace
 * @param newBackground a picture with the new background to use
 * @param threshold the distance from the old background color to use to figure out which pixels to swap with the new background
 */
public void swapBackgroundForThreshold(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);
```
Section 4.2 Chromakey

/* if the color at the current pixel is within 15.0 of the old background pixel * then swap in the new background pixel */
if (currPixel.colorDistance(oldPixel.getColor()) < threshold)
{
    newPixel = newBackground.getPixel(x,y);
    currPixel.setColor(newPixel.getColor());
}
}

End of Recipe 40

FIGURE 4.4: Two people in front of a wall, and a picture of the wall

FIGURE 4.5: Swapping a jungle for the wall, using background subtraction, with a threshold of 50

4.2 CHROMAKEY

The way that weatherpersons do it is to 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 4.6).

![Mark in front of a blue sheet](image)

**FIGURE 4.6:** Mark in front of a blue sheet

### Recipe 41: 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)
{
    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
Section 4.2 Chromakey

* 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());
}
}

End of Recipe 41

The effect is really quite striking (Figure 4.7). Do note the “folds” in the lunar surface, though. The really cool thing is that this recipe works for any background that’s the same size as the image (Figure 4.8). To put Mark on the moon and in the jungle try this:

> Picture mark = new Picture(Picture.getMediaPath("blue-mark.jpg"));
> Picture newBg = new Picture(Picture.getMediaPath("moon-surface.jpg"));
> mark.chromakey(newBg);
> mark.show();
> mark = new Picture(Picture.getMediaPath("blue-mark.jpg"));
> newBg = new Picture(Picture.getMediaPath("jungle3.jpg"));
> mark.chromakey(newBg);
> mark.show();

FIGURE 4.7: Mark on the moon
There's another way of writing this code, which is shorter but does the same thing.

### Recipe 42: 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[] pixels = this.getPixels();
    Pixel currPixel = null;
    Pixel newPixel = null;

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

        /* if the color at the current pixel mostly red (red value is
        * greater than green and blue combined, then use new background color
        */
        if (currPixel.getRed() + currPixel.getGreen() < currPixel.getBlue()) {
```
Section 4.2  Chromakey

newPixel = newBg.getPixel(currPixel.getX(), currPixel.getY());
currPixel.setColor(newPixel.getColor());
}
}

End of Recipe 42

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 4.9—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 jungle-ified. The one with the flash was better, but the flash is still clear after the swap (Figure 4.10). It’s clear why moviemakers and weather people use blue or green backgrounds.

FIGURE 4.9: Student in front of a red background, and with flash on

FIGURE 4.10: Using chromakey recipe with red background
4.3 DRAWING ON IMAGES WITH PIXELS

Sometimes you want to create your own images from scratch. We know that this is just a matter of setting pixel values to whatever we want, but setting individual pixel values to draw a line or a circle or some letters is hard. One way of drawing on images is to simply set the pixels appropriately. Here's an example that we can use to create a grid on the picture of Santa (Figure 4.11).

Recipe 43: Draw lines by setting pixels

```java
/**
 * Method to add vertical and horizontal lines to the current picture
 */
public void addLines()
{
    addHorizontalLines();
    addVerticalLines();
}

/**
 * Method to add a horizontal line every 5 pixels in the current picture
 */
public void addHorizontalLines()
{
    // loop through rows
    for (int y = 0; y < getHeight(); y+=5)
    {
        // loop through the columns
        for (int x = 0; x < getWidth(); x++)
        {
            // set the pixel to black
            this.getPixel(x,y).setColor(Color.black);
        }
    }
}

/**
 * Method to add a vertical line every 5 pixels in the current picture
 */
public void addVerticalLines()
{
    // loop through the columns
    for (int x = 0; x < getWidth(); x+=5)
    {
        // loop through the rows
        // set the pixel to black
        this.getPixel(x,y).setColor(Color.black);
    }
}
```
Section 4.3 Drawing on images with pixels

FIGURE 4.11: Santa with a grid of drawn lines

```java
for (int y=0; y < getHeight(); y++)
{
    // set the pixel to black
    this.getPixel(x,y).setColor(Color.black);
}
```

End of Recipe 43

To run this try:

```java
> Picture santa = new Picture(Picture.getMediaPath("santa.jpg"));
> santa.addLines();
> santa.show();
```

You'll notice that this program is using the color name `Color.black`. Java pre-defines for you a bunch of colors: `Color.black`, `Color.white`, `Color.blue`, `Color.red`, `Color.green`, `Color.gray`, `Color.lightGray`, `Color.darkGray`, `Color.yellow`, `Color.orange`, `Color.pink`, `Color.magenta`, and `Color.cyan`. You can use any of these when you need a color. Each of these is a class field of the `Color` class which is why you can use them using the Class 'dot' field notation (Color.black).

We can imagine drawing anything we want like this. We could draw rectangles or circles, simply by figuring out what pixels need to be what color. We could even draw letters—by setting the appropriate pixels to the appropriate colors, we could make any letter we want.
4.4 DRAWING WITH DRAWING COMMANDS

Actually, drawing lines and other shapes by changing the colors of individual pixels is amazingly painful to even imagine. Fortunately, there’s a much easier way.

Java (like most modern programming languages) provides methods for drawing a variety of shapes and text. There are actually two different classes that you can use for drawing java.awt.Graphics and java.awt.Graphics2D. The Graphics class contains methods for doing simple drawing while the Graphics2D class can be used for more sophisticated drawing. Here are some of the methods in the Graphics class:

- `drawString(string, x, y)` draws the string starting at position \((x, y)\) in the current font and color.
- `drawLine(x1, y1, x2, y2)` draws a line from position \((x1, y1)\) to \((x2, y2)\) in the current color.
- `drawRect(x1, y1, width, height)` draws a rectangle with the upper left hand corner at \((x1, y1)\), a width of \(width\), and a height of \(height\) in the current color.
- `fillRect(x1, y1, width, height)` draws a rectangle with the upper left hand corner at \((x1, y1)\), a width of \(width\), and a height of \(height\) and filled with the current color.
- `drawOval(x1, y1, width, height)` draws an oval that is enclosed by a rectangle with the upper left hand corner of the rectangle at \((x1, y1)\), a width of \(width\) and a height of \(height\) in the current color.
- `fillOval(x1, y1, width, height)` draws a filled oval that is enclosed by a rectangle with the upper left hand corner of the rectangle at \((x1, y1)\), a width of \(width\) and a height of \(height\) in the current color. The oval is filled with the current drawing color.
- `drawArc(x1, y1, width, height, startAngle, arcAngle)` draws an arc which is part of the oval specified by the enclosing rectangle with a upper left hand corner of the rectangle at \((x1, y1)\), a width of \(width\) and a height of \(height\) in the current color. The arc starts at the given start angle and has an angle of \(arcAngle\). A start angle of 0 means the 3 o’clock position. A negative value is clockwise rotation and a positive value is counter-clockwise rotation.
- `fillArc(x1, y1, width, height, startAngle, arcAngle)` draws a filled arc which is part of the oval specified by the enclosing rectangle with a upper left hand corner of the rectangle at \((x1, y1)\), a width of \(width\) and a height of \(height\) in the current color. The arc is filled with the current drawing color.

You can get an object of the Graphics class from a Picture object using `getGraphics()`. Then you can do the drawing using the graphics object and repaint the picture to see the changes.
Below is an example of using these drawing commands (Figure 4.12).

Recipe 44: An example of using drawing commands

```java
/**
 * Method to show using drawing on a picture
 * @return the example picture
 */
public static Picture drawExample()
{
    // start with a white 640 by 480 picture
    Picture p = new Picture(Picture.getMediaPath("640x480.jpg"));

    // get the graphics object to use for drawing
    Graphics graphics = p.getGraphics();

    // start with a black color
    graphics.setColor(Color.black);

    // draw the string with a upper left corner at x=10, y=75
    graphics.drawString("This is a test of drawing a string on a picture",10,75);

    // draw a line from (10,20) to (300,50)
    graphics.drawLine(10,20,300,50);

    // set the color to yellow
    graphics.setColor(Color.yellow);

    // draw a filled rectangle (filled with yellow) at upper left (0,200) with
    // a width of 300 and height 250
    graphics.fillRect(0,200,300,250);

    // set the color back to black
    graphics.setColor(Color.black);

    // draw the outline of a rectangle with the upper left at (10,210) and
    // a width of 200 and a height of 100
    graphics.drawRect(10,210,200,100);

    /* draw an oval enclosed by a rectangle with the top left corner at
    * (400,10) and a width of 200 and a height of 100
    */
    graphics.drawOval(400,10,200,100);

    /* draw an arc which is part of an oval enclosed by a rectangle with the
    */
```
FIGURE 4.12: A small, drawn picture

* top left corner at (400,300) a width of 200, and a height of 150. The
* arc starts at 0 (3 o’clock position) and goes 180 degrees counter-clockwise
* to the 9 o’clock position
*/
graphics.fillArc(400,300,200,150,0,180);

// show the picture
p.show();

// return the picture
return p;

4.4.1 Vector and Bitmap Representations

Here’s a thought: Which of these is smaller—the picture (Figure 4.12) or the
Recipe 44 (page 155)? The picture, on my disk, is about 24 kilobytes (a kilo-
byte is about a thousand bytes). The recipe is about 1 kilobyte (calculated by
comparing the Picture.java file size before and after adding the method). But for
many purposes, they are equivalent. What if you just saved the program and not
the pixels? That’s what a vector representation for graphics is about.
Vector-based graphical representations are basically executable programs that generate the picture when desired. Vector-based representations are used in Postscript, Flash, and AutoCAD. When you make a change to an image in Flash or AutoCAD, you are actually making a change to the underlying representation—essentially, you’re changing the program, like the one in Recipe 4.12 (page 156). The program is then executed again to make the image appear. But thanks to Moore’s Law, that execution-and-new-display occurs so fast that it feels like you’re changing the picture.

Font definitions languages like Postscript and TrueType actually define miniature programs (or equations) for each and every letter or symbol. When you want the letter or symbol at a particular size, the program is run to figure out which pixels should be set to what values. (Some actually specify more than one color to create the effect of smoother curves.) Because the programs are written to handle desired font size as an input, the letters and symbols can be generated at any size.

Bitmap graphical representations, on the other hand, store every individual pixel, or some compressed representation of the pixels. Formats like BMP, GIF, and JPEG are essentially bitmap representations. GIF and JPEG are compressed representations—they don’t represent each and every pixel with 24 bits. Instead, they use some techniques to represent the same information but with fewer bits.

There are several benefits to vector-based representations over bitmap representations. If you can represent the picture you want to send (say, over the Internet) using a vector-based representation, it’s much smaller than sending all the pixels. Essentially, you’re sending the instructions for how to make the picture, rather than sending the picture itself. For very complex images, however, the instructions can be as long as the image itself (imagine sending all the directions on how to paint the Mona Lisa!), so there is no benefit. But when the images are simple enough, representations like those used in Flash make for faster upload and download times than sending the same JPEG images.

The real benefit of vector-based notations come when you want to change the image. Let’s say that you’re working on an architectural drawing, and you extend a line in your drawing tool. If your drawing tool is only working with bitmapped images (sometimes called a painting tool) then all you have are more pixels on the screen that are adjacent to the other pixels on the screen representing the line. There’s nothing in the computer that says that all those pixels represent a line of any kind—they’re just pixels. But if your drawing tool is working with vector-based representations (sometimes called a drawing tool) then extending a line means that you’re changing an underlying representation of a line.

Why is that important? The underlying representation is actually a specification of the drawing, and it can be used anywhere that a specification is needed. Imagine taking the drawing of a part, then actually running the cutting and stamping machines based on that drawing. This happens regularly in many shops, and it’s possible because the drawing isn’t just pixels—it’s a specification of the lines and their relationships, which can then be scaled and used to determine the behavior of machines.

You might be wondering, ”But how could we change the program? Can we write a program that would essentially re-type the program or parts of the program?” Yes, we can, and we’ll do that in the chapter on text.
4.5 PROGRAMS AS SPECIFYING DRAWING PROCESS

What we can do with drawing functions like these is create pictures that are exactly specified—things that might be too hard to do by hand. Take, for example, Figure 4.13.

This doesn't work quite as well as a professionally done picture, but it's simple to understand how it works. Our eyes tell us that the left half of the picture is lighter than the right half, even though the end quarters are exactly the same shade of gray. The effect is caused by the sharp boundary between the middle quarters, where one moves (left-to-right) from gray to white, and the other moves black to gray.

The image in Figure 4.13 is a carefully defined and created picture. It would be very hard to do with pencil and paper. It would be possible to do with something like Photoshop, but it wouldn't be easy. Using the graphics functions in this chapter, however, we can easily specify exactly what that picture should be.

Recipe 45: Draw the gray effect picture

```java
/**
 * Method to draw a gray effect picture
 * @return the picture that shows the gray effect
 */
public static Picture drawGrayEffect()
{
    // create a picture to draw on
    Picture pic = new Picture(400,100);

    // create a medium gray color to use
    Color medGray = new Color(100,100,100);

    // Do 100 columns of medium gray
    for (int x = 0; x < 100; x++)
        for (int y = 0; y < 100; y++)
            pic.getPixel(x,y).setColor(medGray);

    // Do 100 columns of gray starting at medium gray and getting lighter
    for (int x=100, grayLevel=100; x < 200; x++,grayLevel++)
```
Section 4.5 Programs as Specifying Drawing Process

```java
for (int y=0; y < 100; y++)
    pic.getPixel(x,y).setColor(new Color(grayLevel,grayLevel,grayLevel));

// Do 100 columns starting at black and getting lighter
for (int x=200, grayLevel=0; x < 300; x++, grayLevel++)
    for (int y=0; y < 100; y++)
        pic.getPixel(x,y).setColor(new Color(grayLevel,grayLevel,grayLevel));

// Do 100 columns of medium gray
for (int x=300; x < 400; x++)
    for (int y=0; y < 100; y++)
        pic.getPixel(x,y).setColor(medGray);

// show the picture
pic.show();

// return the picture
return pic;
```

End of Recipe 45

Making it Work Tip: Class Methods versus Object Methods

Notice that both of the methods that create and return pictures are declared as `static`. This makes them class methods and means that they can be called (invoked) using the class name followed by a dot and then the method name: `Picture.drawExample()` and `Picture.drawGrayEffect()`. Why are they class methods instead of object methods (a method called on an object of the class)? When we call these methods do we have a current picture object to work on? No, these methods create a picture object so there is no picture object to call the methods on. So, when there is no current object to call the method on use a class method and when there is a current object to call the method on use an object method.

Graphics functions are particularly good at drawings that are repeated where the positions of lines and shapes and the selection of colors can be made by math-
emathetical relationships.

Recipe 46: Draw the picture in Figure 4.14

```java
/**
 * Method to draw a picture with a succession of filled rectangles
 * with the top left corner the darkest and the bottom right the
 * lightest
 * @return the picture with the filled rectangles
 */
public static Picture drawFilledRectangles()
{
    Picture p = new Picture(250,250);
    Graphics g = p.getGraphics();
    Color color = null;

    // loop 25 times
    for (int i = 25; i > 0; i--)
    {
        color = new Color(i * 10, i * 5, i);
        g.setColor(color);
        g.fillRect(0,0,i*10,i*10);
    }

    // show the picture
    p.show();

    // return the picture
    return p;
}
```

End of Recipe 46

Recipe 47: Draw the picture in Figure 4.15

```java
/**
 * Method to draw a picture with a succession of rectangles
 * @return the picture with the filled rectangles
 */
public static Picture drawRectangles()
{

```
Section 4.5 Programs as Specifying Drawing Process

FIGURE 4.14: Nested filled rectangles image

```java
Picture p = new Picture(Picture.getMediaPath("640x480.jpg"));
Graphics g = p.getGraphics();
Color color = null;

// loop 25 times
for (int i = 25; i > 0; i--)
{
    g.setColor(Color.black);
    g.drawRect(i,i,i*3,i*4);
    g.drawRect(100+i*4,100+i*3,i*8,i*10);
}

// show the picture
p.show();

// return the picture
return p;
```

4.5.1 Why do we write programs?

Why do we write programs, especially to draw pictures like this? Could we draw pictures like these in Photoshop? Certainly we can, but we’d have to know how, and that’s not easy knowledge to come by. Could I teach you how to do this in Photoshop? Probably, but that may take a lot of effort?

But if I give you these programs, you can create the picture anytime you want. What’s more, by giving you the program, I’m giving you the exact definition that
you can go and change for yourself.

Computer Science Idea: We write programs to encapsulate and communicate process

The reason why we write programs is to exactly specify a process and to communicate it to others.

Imagine that you have some process to communicate. It doesn’t have to be drawing—imagine that it’s a financial process (such that you could do it in a spreadsheet or in a program like Quicken) or something that you do with text (such as laying out text for a book or a brochure). If you can do something by hand, you should just do it. If you need to teach someone else to do it, consider writing a program to do it. If you need to explain to lots of people how to do it, definitely use a program. If you want lots of people to be able to do the process themselves, without someone having to teach them something first, definitely write a program and give the people the program.

FUNCTIONS AND OBJECTS SUMMARY

Here are the functions introduced in this chapter:
Section 4.5 Programs as Specifying Drawing Process

<table>
<thead>
<tr>
<th>Method Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>graphics.drawString(string,x,y)</code></td>
<td>Draws the string starting at position ((x, y)) in the picture using the graphics object.</td>
</tr>
<tr>
<td><code>graphics.drawLine(x1,y1,x2,y2)</code></td>
<td>Draws a line from position ((x_1, y_1)) to ((x_2, y_2)) using the graphics object.</td>
</tr>
<tr>
<td><code>graphics.drawRect(x1,y1,width,height)</code></td>
<td>Draws a rectangle with the upper left hand corner at ((x_1, y_1)), a width of (width), and a height of (height) using the graphics object.</td>
</tr>
<tr>
<td><code>graphics.fillRect(x1,y1,width,height)</code></td>
<td>Draws a rectangle filled with the current color with the upper left hand corner at ((x_1, y_1)), a width of (width), and a height of (height).</td>
</tr>
<tr>
<td><code>graphics.drawOval(x1,y1,width,height)</code></td>
<td>Draws an oval enclosed by the rectangle with the upper left hand corner at ((x_1, y_1)), a width of (width), and a height of (height) using the graphics object.</td>
</tr>
<tr>
<td><code>graphics.fillOval(x1,y1,width,height)</code></td>
<td>Draws a filled oval enclosed by the rectangle with the upper left hand corner at ((x_1, y_1)), a width of (width), and a height of (height) using the graphics object.</td>
</tr>
<tr>
<td><code>graphics.drawArc(x1,y1,width,height,startAngle,arcAngle)</code></td>
<td>Draws a part of an oval enclosed by the rectangle with the upper left hand corner at ((x_1, y_1)), a width of (width), and a height of (height) using the graphics object. The (startAngle) says at what angle to start drawing the arc and the (arcAngle) is what is the total angle of the arc.</td>
</tr>
<tr>
<td><code>graphics.fillArc(x1,y1,width,height,startAngle,arcAngle)</code></td>
<td>Draws a filled part of an oval enclosed by the rectangle with the upper left hand corner at ((x_1, y_1)), a width of (width), and a height of (height) using the graphics object. The (startAngle) says at what angle to start drawing the arc and the (arcAngle) is what is the total angle of the arc.</td>
</tr>
<tr>
<td><code>graphics.setColor(color)</code></td>
<td>Changes the drawing color to the passed in color object.</td>
</tr>
</tbody>
</table>

**PROBLEMS**

1. Try doing chromakey in a range—grab something out of its background where the something is only in one part of a picture. Now, composite the something into a new picture (new background).

2. Using the drawing tools presented here, draw a house—just go for the simple
Chapter 4  Advanced Pictures

child’s house with one door, two windows, walls, and a roof.

4.3. Using the house tool from the last exercise, draw a town with dozens of houses at different sizes.

4.4. Using the drawing tools presented here, draw a face with two eyes, a nose, and a smile.

4.5. Draw a rainbow.
CHAPTER 5

Advanced Sounds: Synthesizing Sounds
CHAPTER 6

Design and Debugging
Bibliography


9. Idit Harel and Seymour Papert, Software design as a learning environment, Interactive Learning Environments 1 (1990), no. 1, 1–32.


Index

=, 32
==, 55

abstraction, 40
ACM Turing Award, 14
acuity, 47
algebra, 19
algorithm, 6
algorithms, 7
alpha channel, 50
American Standard Code for Information Interchange (ASCII), 10
analogue, 12
applets, 19
argument, 40
array, 17
artificial intelligence, 7
ASCII, 10
assignment, 32
AutoCAD, 157

background subtraction, 143
base file name, 27
binary number, 9
binding, 32, 33
bit, 9
bitmap, 46
bitmap graphical representations, 157
blend, 108
blending
   pictures, 108
block, 36
blue, 11
blurring, 128
BMP, 157
body of method, 36
byte, 9, 17

C, 8
calculus, 14
capitalization, 26
casting, 66
changeRed, 78
channel, 49
Index

chromakey, 147
class, 16, 20
   fully qualified name, 54
class methods, 25
CMYK color model, 49
code, 17
collage, 102
color
   comparing, 55
   sepia, 122
color replacement, 117
color.brighter(), 56
   color.darker(), 56
   ColorChooser.pickAColor(), 56
commands, 35
common bug
   An Example Common Bug, 2
   Backslashes and Slashes, 34
   DrJava is slow to start, 21
   End with .jpg, 56
   Java’s types can produce odd results, 24
   Making DrJava run faster, 21
   Patience: for loops can take a long time, 63
   Saving a file quickly—and how to find it again, 56
   Seeing changes in the picture, 55
comparing colors, 55
compressed, 18, 46, 51
computational recipe, 6
computer, 9
computer music, 7
computer science idea
   An Example Idea, 2
   Computer science is the study of recipes, 6
   Computers can layer encodings, 10
   Moore’s Law, 11
   Much of programming is about naming, 16
   Programs are for people, not computers., 16
   Programs are for people., 73
   Scope, 65
   The most important skill is tracing, 63
   We can substitute names, values, and methods., 34
   We write programs to encapsulate and communicate process, 162
constants, 8, 131
convention, 17
coordinates, 47
copying, 96
cropping, 100
Index

data, 17
 data representation, 11
 data structures, 7, 11
 databases, 7
debugging
 using print statements, 94
debugging
 using print statements, 93
debugging tip
 == and .equals, 55
 An Example Debugging Tip, 2
 Common typos, 25
 Loops and Variable Declarations, 61
 Methods names must be followed by parentheses, 30
 Undefined Class Error, 54
decimal number, 9
def, 40
definitions pane, 22
Digital, 12
Digital media, 12
digitization, 12
digitizing media
 pictures, 48
 why?, 12
directory, 18, 27
disk, 18
distance, 55
double quote, 24
drawArc(x1,y1,width,height,startAngle,arcAngle), 154
drawing tool, 157
drawLine(x1,y1,x2,y2), 154
drawOval(x1,y1,width,height), 154
drawRect(x1,y1,width,height), 154
drawString(string,x,y), 154
DrJava, 20
 definitions pane, 22
 interactions pane, 22
 files pane, 22
 installing, 20
 running slowly, 22
 starting, 21
 Dynabook, 14

editor, 20
emergent properties, 7
encoding, 9, 27
equals(), 55
evaluation, 33, 40
explore(), 52
expression, 24, 32

file, 18
file extension, 27
files pane, 22
fillArc(x1,y1,width,height,startAngle,arcAngle), 154
fillOval(x1,y1,width,height), 154
fillRect(x1,y1,width,height), 154
Flash, 157
floating point
typing, 25
floating point number, 14, 18
font
how defined, 157
for, 59
for loop, 59
in command area, 61
full file name, 26
fully qualified name, 54
function
arguments, 40
input values, 40
input variables, 40
parameters, 40
when to make one, 41
function calls, 34
functions, 25
general, 72
getColor(), 53
getHeight(), 52
getPixel(x,y), 53
getPixels(), 53
getRed(), 53
getWidth(), 52
getX(), 53
getY(), 53
GIF, 157
going digital, 12
graphics, 7
grayscale, 82
green, 11

hard disk, 18
hierarchical decomposition, 77
hierarchy, 77
HSV color model, 49
HTML, 11
human-computer interface, 7
import, 54
  how to, 54
  why, 54
infinite loop, 111
input, 25
input value, 40
input variable, 40
integer, 17, 33
  typing, 25
Intel, 11
intelligence, 7
intelligent systems, 7
intensity, 83
interactions pane, 22
interface, 7
Java, 9, 19
  capitalization, 26
  installing, 20
JPEG, 27, 157
kilobyte, 156
knots, 7
liberal education, 14
Lisp, 8
lossy compression, 46
luminance, 48, 50, 83
making it work
  Add an input parameter to generalize a method, 146
  An Example How To Make It Work, 2
  Class Methods versus Object Methods, 159
  Comments in Java, 62
  Copying and pasting, 39
  Don’t just trust your programs, 67
  Don’t start by trying to write applications, 74
  Get to know your Help, 23
  Importing Classes from Packages, 53
  Java Conventions, 30
  Java Keywords, Operators, and Classes, 17
  Name the names you like, 38
  Optional parts of the for loop, 110
Index 173

Try every recipe, 35
Types in Java, 28
Using dot notation for public fields, 46
matrix, 47
  definition, 47
media computation, 11
MediaTools, 58
  picture tools, 58
memory, 9, 18, 22
method, 16, 25
method body, 36
methods, 25
methods vs. recipes, 73
modulus, 24
Moore's Law, 11
Moore, Gordon, 11

negative image, 81
nested, 85
networking, 7
new Color(redValue, greenValue, blueValue), 53
new Picture(fileName), 52
new Picture(width, height), 116
new Sound(fileName), 30
null, 26

object
  definition, 19
object methods, 25
object-oriented, 19
operating system, 18
operators, 24

package
  definition, 54
painting tool, 157
parameter, 40
path, 27
path separator, 27
percentage, 58
Perlis, Alan, 13
physics
  color, 11
pickAFile(), 26
picture
  show(), 28
picture element, 12
Index

picture objects, 46
picture tool, 58
Picture.getMediaPath(fileName), 93, 96
picture.repaint(), 55
picture.write(fileName), 56
pixel, 12, 46, 47
  definition, 47
pixelation, 128
pixelization, 48
pixels, 46
placeholder, 77
play(), 30
posterizing, 124
Postscript, 157
primitive, 29
primitive type, 29
process, 14
program, 6, 16, 17
  defined, 17
programming language, 16
programming languages, 8

ranges, 8

recipe, 6
  A simple blur, 128
  An example of using drawing commands, 155
  An Example Recipe, 2
  Better Swap Background, 146
  Blending two pictures, 108
  Chromakey, shorter, 150
  Chromakey: Replace all blue with the new background, 148
  Clear the blue component from a picture, 69
  Color replacement in a range, 118
  Color replacement: Turn Katie into a redhead, 117
  Convert a picture to sepia-tones, 123
  Convert to grayscale, 83
  Convert to grayscale with more careful control of luminance, 83
  Copy elsewhere into the canvas, 99
  Copying a picture to a canvas, 96
  Create the negative of the original picture, 81
  Creating a collage, 103
  Cropping a picture onto a canvas, 100
  Darken the picture, 80
  Draw lines by setting pixels, 152
  Draw the gray effect picture, 158
  Draw the picture in Figure 4.14, 160
  Draw the picture in Figure 4.15, 160
Increase the red component by 30%, 68
Lighten the picture, 79
Lighten the picture using nested loops, 85
Making a sunset, 70
Making a sunset as three methods, 72
Mirror pixels horizontally, bottom-to-top, 90
Mirror pixels horizontally, top-to-bottom, 89
Mirror pixels in a picture along a vertical line, 87
Mirror the Temple of Zeus, 91
Pick and play a sound, 38
Pick and show a picture, 37
Play a specific sound, 39
Play the sound file whose file name is input, 41
Posterize by levels, 126
Posterizing a picture, 125
Reduce the amount of red in a picture by 50%, 61
Remove red eye, 120
Rotating a picture, 111
Scaling a picture down (smaller), 114
Scaling the picture up (larger), 115
Show a specific picture, 39
Show the picture file whose file name is input, 40
Show the picture provided as input, 41
Subtract the background and replace it with a new one, 143
recipes vs. methods, 73
red, 11
return, 77
reusable, 72
RGB model, 49
rounding errors, 56

sample, 12
sampling, 113
scope, 65
sepia-toned, 122
setColor(color), 53, 154
setRed(redValue), 53
show(), 28, 52
software engineering, 7, 13
sound
   new Sound(fileName), 30
   play(), 30
source, 96
specification, 157
stepping, 63
String, 17, 34
string, 33
strings, 24
strongly typed, 18
substitution, 33
substitution, 40
Sun, 20
symbols, 16
syntax, 35
systems, 7
takes on, 40
target, 96
testing
  using print statements, 93
testing, 7
trace, 63
tracing
  using print statements, 94
transforming, 96
transistor, 11
transparency, 50, 108
TrueType, 157
type, 16
types
  byte, 17
double, 18
float, 18
int, 17
integer array, 17
String, 17
Unicode, 10
  getting the mapping, 25
utility function, 96
variable, 28
  declaration, 28
variables, 19, 32
  reusing, 32
vector, 46
vector representation, 156
visibility, 35
walking through, 63
WAV, 27