BETA Version, January 10th, 2005:
Introduction to Computing and Programming in Java:
A Multimedia Approach

Mark Guzdial and Barbara Ericson

College of Computing/GVU
Georgia Institute of Technology

PRENTICE HALL, Upper Saddle River, New Jersey 07458
Copyright held by Mark Guzdial and Barbara Ericson, 2005.
Dedicated to our children Matthew, Katherine, and Jennifer
Contents

Contents iv
Preface 1

I Introduction 7
1 Introduction to Computer Science and Media Computation 8
  1.1 What is Computer Science About? 8
  1.2 What Computers Understand 13
  1.3 Media Computation: Why Digitize Media? 15
  1.4 Computer Science for Everyone 16
    1.4.1 It’s About Communication 17
    1.4.2 It’s About Process 17
2 Introduction to Java 19
  2.1 Java 19
    2.1.1 History of Java 19
    2.1.2 Introduction to Objects and Classes 20
  2.2 Introduction to DrJava 21
    2.2.1 Starting DrJava 21
  2.3 Java Basics 24
    2.3.1 Math Operators 24
    2.3.2 Printing the Result of a Statement 24
    2.3.3 Types in Math Expressions 27
    2.3.4 Casting 27
    2.3.5 Relational Operators 27
    2.3.6 Strings 29
  2.4 Variables 31
    2.4.1 Declaring Variables 31
    2.4.2 Using Variables in Calculations 31
    2.4.3 Memory Maps of Variables 32
    2.4.4 Object Variables 34
    2.4.5 Reusing Variables 36
    2.4.6 Multiple References to an Object 37
  2.5 Concepts Summary 39
    2.5.1 Statements 39
    2.5.2 Relational Operators 40
    2.5.3 Types 40
    2.5.4 Casting 40
    2.5.5 Variables 41
3 Introduction to Programming 43

3.1 Programming is About Naming 43
3.2 Files and their Names 45
3.3 Class and Object Methods 46
  3.3.1 Invoking Class Methods 46
  3.3.2 Invoking Object Methods 47
3.4 Working with Turtles 48
  3.4.1 Defining Classes 48
  3.4.2 Creating Objects 48
  3.4.3 Sending Messages to Objects 51
  3.4.4 Objects Control Their State 52
  3.4.5 Additional Turtle Capabilities 53
3.5 Creating Methods 56
  3.5.1 Methods that Take Input 61
3.6 Working with Media 64
  3.6.1 Creating a Picture Object 64
  3.6.2 Showing a Picture 66
  3.6.3 Variable Substitution 68
  3.6.4 Object references 70
  3.6.5 Playing a Sound 71
  3.6.6 Naming your Media (and other Values) 72
  3.6.7 Naming the Result of a Method 72
3.7 Concepts Summary 74
  3.7.1 Invoking Object Methods 74
  3.7.2 Invoking Class Methods 74
  3.7.3 Creating Objects 74
  3.7.4 Creating new Methods 75

II Pictures 79

4 Modifying Pictures using Loops 80

4.1 How Pictures are Encoded 80
  4.1.1 Color Representations 84
4.2 Manipulating Pictures 88
  4.2.1 Exploring Pictures 94
4.3 Changing color values 95
  4.3.1 Using While Loops in Pictures 96
  4.3.2 Increasing/Decreasing Red (Green, Blue) 100
  4.3.3 Creating a Sunset 113
  4.3.4 Making Sense of Methods 115
  4.3.5 Variable Name Scope 118
  4.3.6 Using a For Loop 122
  4.3.7 Lightening and Darkening 124
  4.3.8 Creating a Negative 125
  4.3.9 Converting to Grayscale 127
4.4 Concepts Summary 129
5 Modifying Pixels in a Matrix 136
5.1 Copying Pixels 136
5.1.1 Looping Across the Pixels with a Nested Loop 137
5.1.2 Mirroring a Picture 139
5.2 Copying and Transforming Pictures 147
5.2.1 Copying 148
5.2.2 Creating a Collage 154
5.2.3 Blending Pictures 159
5.2.4 Rotation 162
5.2.5 Scaling 166
5.3 Concepts Summary 171
5.3.1 Two-dimensional Arrays 171
5.3.2 Nested Loops 172
5.3.3 Returning a Value from a Method 173
5.3.4 Method Overloading 174
5.4 Color Figures 176

6 Conditionally Modifying Pixels 185
6.1 Conditional Pixel Changes 186
6.1.1 Comparing Colors 187
6.1.2 Replacing Colors 187
6.1.3 Reducing Red-Eye 192
6.2 Simple Edge Detection: conditionals with two options 194
6.2.1 Negation 194
6.2.2 Testing for Both Conditions 195
6.2.3 Conditionals with Two Options 195
6.2.4 Simple Edge Detection 196
6.3 Sepia-Toned and Posterized Pictures: Using multiple conditionals to choose the color 198
6.4 Highlighting Extremes 206
6.5 Combining Pixels: Blurring 207
6.6 Background Subtraction 210
6.7 Chromakey 214
6.8 Concepts Summary 218
6.8.1 Boolean Expressions 218
6.8.2 Combining Boolean Expressions 219
6.8.3 Conditional Execution 220

7 Drawing 224
7.1 Drawing on Images Using the Graphics Class 224
7.1.1 Drawing with Graphics methods 227
7.1.2 Vector and bitmap representations 233
7.1.3 Drawing Text (Strings) 235
9.5 Mirroring a Sound ........................................... 317
9.6 Concepts Summary ........................................ 318
  9.6.1 Ranges in Loops ...................................... 318
  9.6.2 Returning a Value from a Method ..................... 318

10 Combining and Creating Sounds ......................... 321
  10.1 Composing Sounds Through Addition ................... 321
  10.2 Blending Sounds ....................................... 323
  10.3 Creating an Echo ....................................... 324
    10.3.1 Creating Multiple Echoes ......................... 327
  10.4 How Sampling Keyboards Work .......................... 328
    10.4.1 Sampling as an Algorithm ......................... 334
  10.5 Additive Synthesis .................................... 334
    10.5.1 Making Sine Waves ................................. 334
    10.5.2 Creating Sounds Using Static Methods ............. 335
    10.5.3 Adding Sine Waves Together ....................... 337
    10.5.4 Checking our Result ................................ 338
    10.5.5 Square Waves ..................................... 339
    10.5.6 Triangle Waves ..................................... 342
  10.6 Modern Music Synthesis ................................ 343
    10.6.1 MP3 ................................................ 343
    10.6.2 MIDI .............................................. 344
    10.6.3 Private Methods .................................... 346
  10.7 Concepts Summary ..................................... 352
    10.7.1 Class Methods ..................................... 352
    10.7.2 Private Methods .................................... 353
    10.7.3 Build a Program from Multiple Methods .......... 353
List of Figures

1.1 A cooking recipe—you can always double the ingredients, but throwing in an extra cup of flour won’t cut it, and don’t try to brown the chicken after adding the tomato sauce! .................................................. 11
1.2 Comparing programming languages: A common simple programming task is to print the words “Hello, World!” to the screen. ...... 12
1.3 Eight wires with a pattern of voltages is a byte, which gets interpreted as a pattern of eight 0’s and 1’s, which gets interpreted as a decimal number. ................................................................. 13
2.1 DrJava Preferences Window ....................................................... 22
2.2 DrJava Splash Screen ............................................................... 22
2.3 DrJava (with annotations) ....................................................... 23
2.4 A calculator with a number in memory .................................... 31
2.5 Declaring primitive variables and memory assignment ............ 33
2.6 Showing the parent and child relationship between mammal and dog (left) and Object and String (right) .................................... 34
2.7 Declaring object variables and memory assignment ................ 35
2.8 Shows creation and reuse of an object variable. ....................... 37
2.9 An object with multiple references to it .................................. 38
2.10 An object with no references to it .......................................... 38
3.1 A window that shows a World object. ..................................... 49
3.2 A window that shows a Turtle object in a World object. .......... 50
3.3 A window that shows two Turtle objects in a World object. ...... 51
3.4 The result of messages to the first Turtle object. ..................... 52
3.5 The result of messages to the second Turtle object. ................. 53
3.6 The turtle won’t leave the world ............................................. 54
3.7 Drawing two squares with a turtle. ........................................ 55
3.8 Defining and executing drawSquare() .................................... 59
3.9 Diagram of an object ............................................................. 60
3.10 Showing the result of sending the width as a parameter to drawSquare .......................................................... 63
3.11 Creating a Picture object using new Picture() ....................... 64
3.12 The File Chooser ................................................................. 65
3.13 File chooser with media types identified .............................. 67
3.14 Picking, making, and showing a picture, using the result of each method in the next method. The picture used is matt-spaceman.jpg. .......................................................... 68
3.15 Picking, making, and showing a picture, when naming the pieces.
   The picture shown is katie.jpg. ............................................. 71
4.1 A depiction of the first five elements in an array .................... 81
4.2 An example matrix (two-dimensional array) of numbers .......... 82
4.3 The top left corner of the battleship guess board with a miss at B-3. 83
4.4 Upper left corner of DrJava window with part magnified 600% ... 83
LIST OF FIGURES

4.5 Image shown in the picture explorer: 100% image on left and 500% on right (close-up of the branch over the mountain) ............... 84
4.6 Merging red, green, and blue to make new colors ...................... 85
4.7 Picking colors using the HSB color model .............................. 85
4.8 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 86
4.9 The Macintosh OS X RGB color picker ................................. 87
4.10 Picking a color using RGB sliders from Java .......................... 87
4.11 RGB triplets in a matrix representation ............................... 88
4.12 Directly modifying the pixel colors via commands: Note the small black line on the left under the line across the leaf ............... 94
4.13 Exploring the caterpillar with the line ................................ 94
4.14 Using the MediaTools image exploration tools ....................... 95
4.15 Flowchart of a while loop ................................................. 98
4.16 The original picture (left) and red-decreased version (right) ....... 103
4.17 Using the picture explorer to convince ourselves that the red was decreased .......................................................... 110
4.18 Overly blue (left) and red increased by 30% (right) ................. 112
4.19 Original (left) and blue erased (right) ................................. 114
4.20 Original beach scene (left) and at (fake) sunset (right) ............. 115
4.21 Flowchart of a for loop ..................................................... 123
4.22 Original picture, lightened picture, and darkened picture ........... 126
4.23 Negative of the image ...................................................... 127
4.24 Color picture converted to grayscale .................................... 128
5.1 Once we pick a mirror point, we can just walk $x$ halfway and subtract/ add to the mirror point ................................. 140
5.2 Original picture (left) and mirrored along the vertical axis (right) . 141
5.3 A motorcycle mirrored horizontally, top to bottom (left) and bottom to top (right) ..................................................... 143
5.4 Temple of Hephaistos from the ancient agora in Athens, Greece .......... 143
5.5 Coordinates where we need to do the mirroring ....................... 144
5.6 The manipulated temple ................................................... 145
5.7 Copying a picture to a canvas .......................................... 149
5.8 Copying a picture midway into a canvas .............................. 150
5.9 Copying part of a picture onto a canvas ............................... 152
5.10 Flowers in the mediasources folder .................................. 154
5.11 Collage of flowers ......................................................... 157
5.12 Blending the picture of Katie and Jenny .............................. 162
5.13 Rotating some numbers in a table to the left 90 degrees ............. 163
5.14 Copying a picture to a blank page rotated to the left 90 degrees ... 164
5.15 Scaling the picture down ................................................. 167
5.16 Scaling up a picture ....................................................... 169
5.17 Merging red, green, and blue to make new colors .................. 176
5.18 Color: RGB triplets in a matrix representation ....................... 176
5.19 Color: The original picture (left) and red-decreased version (right) . 177
5.20 Color: Overly blue (left) and red increased by 30% (right) ........ 177
5.21 Color: Original (left) and blue erased (right) ....................... 177
5.22 Original beach scene (left) and at (fake) sunset (right) ........... 178
5.23 Color: Lightening and darkening the original picture ............... 178
5.24 Color: Negative of the image ........................................ 178
5.25 Color: Color picture converted to grayscale ........................... 179
5.26 Color: Increasing reds in the browns ................................. 179
5.27 Color: Increasing reds in the browns, within a certain range ....... 180
5.28 Finding the range where Jenny’s eyes are red, then changing them
to black ................................................................. 180
5.29 Frames from the slow sunset movie .................................... 181
5.30 Frames from the slow fade-out movie ................................ 181
5.31 Frames from the Mommy watching Katie movie ....................... 182
5.32 Frames from the original too dark movie ................................ 182
5.33 Frames from the modified lighter movie ................................ 183
5.34 Frames from the original movie with kids crawling in front of a blue
screen ................................................................. 183
5.35 Frames from the kids on the moon movie ............................... 184
6.1 Flowchart of an if statement ............................................ 186
6.2 Increasing reds in the browns ........................................... 188
6.3 On left the couch color changes, on right the couch color doesn’t change!190
6.4 Finding the range of where Jenny’s eyes are red ......................... 193
6.5 After fixing red-eye. .................................................... 194
6.6 Flowchart of an if with an else ......................................... 195
6.7 Original picture and after edge detection .............................. 198
6.8 Flowchart of an if, else if, and an else ................................ 199
6.9 Original scene (left) and using our sepia-tone program .......... 200
6.10 Reducing the colors (right) from the original (left) .................. 202
6.11 Pictures posterized to two levels (left) and four levels (right) .... 205
6.12 Original picture (left) and light or dark areas highlighted (right)) . 207
6.13 Making the flower bigger, then blurring to reduce pixellation ...... 210
6.14 A picture of a child (Katie), and her background without her ....... 210
6.15 A new background, the moon ......................................... 211
6.16 Katie on the moon ................................................... 212
6.17 Two people in front of a wall, and a picture of the wall ............ 214
6.18 Swapping a beach for the wall, using background subtraction, with
a threshold of 50 .................................................... 215
6.19 Mark in front of a blue sheet ....................................... 215
6.20 Mark on the moon .................................................. 217
6.21 Mark on the beach .................................................. 218
6.22 Student in front of a red background, and with flash on .......... 219
6.23 Using chromakey program with a red background .................... 219
7.1 Adding a grid of lines to a picture (barbara.jpg) ..................... 226
7.2 Viewing the Java API for java.awt.Graphics ........................ 229
7.3 A box washed up on the shore of the beach ....................... 229
7.4 An example drawn picture ........................................... 232
7.5 A drawn face (left) and the face with enclosing rectangles (right) . 233
7.6 Shows font information including the baseline ......................... 235
7.7 Drawing a string on a picture ........................................ 237
LIST OF FIGURES

7.8 Drawing a string centered on a picture .......................... 238
7.9 A programmed gray scale effect ................................. 239
7.10 Nested colored rectangles ..................................... 241
7.11 Nested outlined rectangles ................................. 242
7.12 Drawing a red X on a picture ................................. 245
7.13 Drawing a turtle on a beach .................................. 246
7.14 Documentation for Graphics2D ............................ 247
7.15 Original picture and picture scaled up 2 times in x and down by half in y .................................................. 249
7.16 Picture sheared by 1.0 in x ........................................ 251
7.17 A beach with a sun that is filled with a gradient from yellow to red ................................................... 253
7.18 Two pictures with a horizontal overlap ..................... 256
7.19 Clip picture using an ellipse .................................. 257
8.1 Raindrops causing ripples in the surface of the water, just as sound causes ripples in the air ........................................ 265
8.2 One cycle of the simplest sound, a sine wave ................ 266
8.3 The note A above middle C is 440 Hz ........................ 267
8.4 Some synthesizers using triangular (or sawtooth) or square waves .................................................. 268
8.5 Sound editor main tool ............................................. 268
8.6 Viewing the sound signal as it comes in ...................... 269
8.7 Viewing the sound in a spectrum view ........................ 270
8.8 Viewing a sound in spectrum view with multiple “spikes” .... 271
8.9 Viewing the sound signal in a sonogram view ............... 272
8.10 Area under a curve estimated with rectangles ............. 272
8.11 A depiction of the first five elements in a real sound array ............................................................... 274
8.12 A sound recording graphed in the MediaTools ............... 274
8.13 The sound editor open menu in MediaTools application .... 279
8.14 MediaTools application open file dialog ...................... 280
8.15 A sound opened in the editor in MediaTools application .... 280
8.16 Exploring the sound in the editor in MediaTools application ................................................................. 281
8.17 Comparing the graphs of the original sound (left) and the louder one (right) ................................. 286
8.18 Comparing specific samples in the original sound (left) and the louder one (right) ........................................ 287
8.19 Comparing the original sound with the normalized one .... 296
8.20 Comparing the original sound with one with all values set to extremes .................................................. 298
9.1 Exploring the "This is a test" to find the end of the first word ............................................................... 307
9.2 Exploring the sound clip ............................................. 309
9.3 Comparing the original sound (left) to the spliced sound (right) ............................................................... 314
9.4 Comparing the original sound (left) to the reversed sound (right) ............................................................... 317
9.5 Comparing the mirror point in the original sound (left) to the mirrored sound (right) ............................................................... 318
10.1 The top and middle waves are added together to create the bottom wave ............................................................... 322
10.2 The original “ahh” sound, the original bassoon note, and the blended sound ............................................................... 325
LIST OF FIGURES xiii

10.3 The original “This is a test” sound (left), and the sound with an echo (right) ................................................. 326
10.4 The original sound (left), and the sound with the frequency doubled (right) .................................................. 329
10.5 The sine wave with a frequency of 880 and a maximum amplitude of 4000 ..................................................... 337
10.6 The raw 440 Hz signal on top, then the 440+880+1320 Hz signal on the bottom .............................................. 339
10.7 FFT of the 440 Hz sound ............................................. 339
10.8 FFT of the combined sound ........................................ 339
10.9 The 440 Hz square wave (top) and additive combination of square waves (bottom) ........................................... 341
10.10 FFT’s of the 440 Hz square wave (top) and additive combination of square waves (bottom) ......................... 341
LIST OF FIGURES
Preface

One of the clearest lessons from the research on computing education is that one doesn’t just “learn to program.” One learns to program something [4, 15]. How motivating that something is can make the difference between learning to program or not [6].

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

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

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

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

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

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

TO TEACHERS

The media computation approach used in this book starts with what students use computers for: image manipulation, digital music, web pages, games, and so on.
2 LIST OF FIGURES

We then explain programming and computing in terms of these activities. We want students to visit Amazon (for example) and think, “Here’s a catalog website—and I know that this is implemented with a database and a set of programs that format the database entries as Web pages.” Starting from a relevant context makes transfer of knowledge and skills more likely, and it also helps with retention.

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

Researchers in computing education have been exploring why withdrawal or failure rates in college-level introductory computing courses have been so high. The rate of students withdrawing from college-level introductory computing courses or receiving a D or F grade (commonly called the WDF rate) has been reported in the 30–50% range, or even higher. One of the common themes from research into why the WDF rate is so high is that computing courses seem “irrelevant” and unnecessarily focusing on “tedious details” such as efficiency [20][1].

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

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

Memory is associative—we remember things based on what else we relate to those things. People can learn concepts and skills on the promise that it will be useful some day, but those concepts and skills will be related only to those promises, not to everyday life. The result has been described as “brittle knowledge” [7]—the kind of knowledge that gets you through the exam, but promptly gets forgotten because it doesn’t relate to anything but being in that class. If we want students to gain transferable knowledge (knowledge that can be applied in new situations), we have to help them to relate the knowledge to more general problems, so that the memories get indexed in ways that associate with those kinds of problems [18]. Thus, we teach with concrete experiences that students can explore and relate to (e.g., iteration for removing red-eye in pictures).

We do know that starting from the abstractions doesn’t really work for stu-
Students. Ann Fleury has shown that novice students just don’t buy what we tell them about encapsulation and reuse (e.g., [10]). Students prefer simpler code that they can trace easily, and actually think that code that an expert would hate is better. It takes time and experience for students to realize that there is value in well-designed systems, and without experience, it’s very difficult for students to learn the abstractions.

Another unusual thing about this book is that we start using arrays in chapter 4, in our first significant programs. Typically, introductory computing courses push arrays off until later, since they’re obviously more complicated than variables with simple values. But a relevant context is very powerful [15]. The matrices of pixels in images occur in the students’ everyday life—a magnifying glass on a computer monitor or television makes that clear.

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

Ways to Use This Book

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

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

- A short introduction to computing could be taught with just chapters 2-4. We have taught even single day workshops on media computation using just this material.

- Chapters 8 and 9 replicate the computer science basics from chapters 4 through 6, but in the context of sounds rather than images. We find the replication useful—some students seem to relate better to the concepts of iteration and conditionals better when working with one medium than the other. Further, it gives us the opportunity to point out that the same algorithm can have similar effects in different media (e.g., scaling a picture up or down and shifting a sound higher or lower in pitch is the same algorithm). But it could certainly be skipped to save time.

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

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

TYPOGRAPHICAL NOTATIONS

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

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

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

```
> 3 + 4
7
```

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

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

**Program 1: An Example Program**

Programs (recipes) appear like this:

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

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

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

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

ACKNOWLEDGEMENTS

Our sincere thanks go out to the following:

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

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

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

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

INTRODUCTION

Chapter 1  Introduction to Computer Science and Media Computation
Chapter 2  Introduction to Java
Chapter 3  Introduction to Programming
CHAPTER 1

Introduction to Computer Science and Media Computation

1.1 WHAT IS COMPUTER SCIENCE ABOUT?

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

Computer Science Idea: Computer science is the study of recipes (programs)
They’re a special kind of recipe—one that can be executed by a computational device, but that point is only of importance to computer scientists. The important point overall is that a computer science program defines exactly what’s to be done.

If you’re a biologist who wants to describe how migration works or how DNA replicates, or if you’re a chemist who wants to explain how an equilibrium is reached in a reaction, or if you’re a factory manager who wants to define a machine-and-belt layout and even test how it works before physically moving heavy things into
Section 1.1  What is Computer Science About?  9

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

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

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

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

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

- There are even computer scientists who worry about whether people like what the recipes produce, like the restaurant critics for the newspaper. Some of these are human-computer interface specialists who worry about whether people like how the recipes work (those “recipes” that produce an interface that people use, like windows, buttons, scrollbars, and other elements of what we think about as a running program).
• Just as some chefs specialize in certain kinds of recipes, like crepes or barbecue, computer scientists also specialize in special kinds of recipes. Computer scientists who work in graphics are mostly concerned with recipes that produce pictures, animations, and even movies. Computer scientists who work in computer music are mostly concerned with recipes that produce sounds (often melodic ones, but not always).

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

The recipe metaphor also works on another level. Everyone knows that some things in a recipe can be changed without changing the result dramatically. You can always increase all the units by a multiplier (say, double) to make more. You can always add more garlic or oregano to the spaghetti sauce. But there are some things that you cannot change in a recipe. If the recipe calls for baking powder, you may not substitute baking soda. If you’re supposed to boil the dumplings then saute’ them, the reverse order will probably not work well (Figure 1.1). Similarly, for software recipes (programs), there are usually things you can easily change: The actual names of things (though you should change names consistently), some of the constants (numbers that appear as plain old numbers, not as variables), and maybe even some of the data ranges (sections of the data) being manipulated. But the order of the commands to the computer, however, almost always has to stay exactly as stated. As we go on, you’ll learn what can be changed safely, and what can’t.

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

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

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


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

Python/Jython

def hello():
    print "Hello World"

Java

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

C++

#include <iostream.h>

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

Scheme

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

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

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

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

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

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

That’s where we come up with the claim that a computer knows about numbers.

Computer Science Idea: Binary Number System

Binary numbers are made up of only 2 digits (0 and 1). We usually work in the decimal number system which has the digits (0 to 9). The value of a decimal number is calculated by multiplying each digit by a power of 10 and summing the result. The powers of 10 start at 0 and increase from right to left. The value of a binary number is calculated by multiplying each digit by a power of 2 and summing the result (Figure 1.3).

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

FIGURE 1.3: Eight wires with a pattern of voltages is a byte, which gets interpreted as a pattern of eight 0’s and 1’s, which gets interpreted as a decimal number.

We’ll talk more about this level of the computer in Chapter 13.
Chapter 1  Introduction to Computer Science and Media Computation

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

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

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

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

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

We can similarly create layers of encodings in the computer for our specific tasks. We can teach a computer that cells contain mitochondria and DNA, and that DNA has four kinds of nucleotides, and that factories have these kinds of presses and these kinds of stamps. Creating layers of encoding and interpretation so that
Section 1.3 Media Computation: Why Digitize Media?

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

What if we encoded each dot in a picture as a collection of three bytes, one each for the amount of red, green, and blue at that dot on the screen? We could collect a bunch of these three-byte-sets to specify all the dots of a given picture. That’s a pretty reasonable way of representing pictures, and it’s essentially how we’re going to do it in Chapter 4.

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

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

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

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

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

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

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

### 1.4 COMPUTER SCIENCE FOR EVERYONE

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

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

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

So why should anyone who does not want to be a computer scientist study computer science? Why should you learn to program? Isn’t it enough to learn to use all this great software? The following two sections provide two answers to these
Section 1.4  Computer Science for Everyone  17

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 [12]. Perlis is an important figure in the field of computer science. The highest award that a computer scientist can be honored with is the ACM Turing Award. Perlis was the first recipient of that award. He’s also an important figure in software engineering, and he started several of the first computer science departments in the United States.

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

PROBLEMS

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

1.2. Find an ASCII table on the Web: A table listing every character and its corresponding numeric representation. Write down the sequence of numbers whose
Chapter 1 Introduction to Computer Science and Media Computation

ASCII values make up your name.

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

1.4. Consider the representation for pictures described in Section 1.3, where each “dot” (pixel) in the picture is represented by three bytes, for the red, green, and blue components of the color at that dot. How many bytes does it take to represent a $640 \times 480$ picture, a common picture size on the Web? How many bytes does it take to represent a $1024 \times 768$ picture, a common screen size? (What do you think is meant now by a “3 megapixel” camera?)

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

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

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

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

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

TO DIG DEEPER

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

Mitchel Resnick’s book *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds* [21] describes how ants, termites, and even traffic jams and slime molds can be described pretty accurately with hundreds or thousands of very small processes (programs) running and interacting all at once.

*Exploring the Digital Domain* [3] is a wonderful introductory book to computation with lots of good information about digital media.
CHAPTER 2

Introduction to Java

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

- To introduce objects and classes.
- To use DrJava to execute Java statements.
- To use Java math and relational operators.
- To recognize different types (encodings) of data, such as integers, floating point numbers, booleans, characters, and strings.
- To introduce casting.
- To introduce variables and show the difference between primitive and object variables.

2.1 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.

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

At a brainstorming session they decided to try to reposition the language for use with the internet. They created a web browser that had Java programs
Chapter 2 Introduction to Java

(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.

2.1.2 Introduction to Objects and Classes

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

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

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

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

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

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

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

2.2 INTRODUCTION TO DRJAVA

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

If you don’t wish to use DrJava you can use this book with another devel-
opment environment. Simply set the classpath (place to look for classes that you
are using in your program) to include the classes used in this book. Check your
documentation for your development environment to see how to do this. We rec-
ommend using DrJava because it is free, easy to use, has an interactions pane for
trying out Java statements, is written in Java so it works on all platforms, and it
includes a debugger.

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

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

2.2.1 Starting DrJava

How you start DrJava depends on your platform. In Windows, you’ll have a DrJava
icon that you’ll simply double-click. In Linux, you’ll probably cd into your Dr-
Java directory and type a command like java -jar drjava-DATE-TIME.jar where
DATE-TIME are values for the release of DrJava that you are using. On the Mac-
intosh, 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 that DrJava is running slowly, give it more memory. You can do that by quitting out of other applications that you’re running. Your email program, your instant messenger, and your digital music player all take up memory (sometimes lots of it!). Quit out of those and DrJava will run faster.

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

- The top left window pane is the files pane. It has a list of the open files in DrJava. In Java each class that you create is usually stored in its own file. Java programs often consist of more than one class, thus more than one file. You can click on a file name in the Files pane to view the contents of that file in the top right window pane (definitions pane).
Section 2.2 Introduction to DrJava

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

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

![FIGURE 2.3: DrJava (with annotations)](image)

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

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

2.3 JAVA BASICS

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

2.3.1 Math Operators

Try typing the following in the interactions pane.

```
> 34 + 56
90
> 26 - 3
23
> 3 * 4
12
> 4 / 2
2
```

As you can see Java understands how to recognize numbers, add, subtract, multiply and divide. You can type a mathematical expression in the interactions pane and then hit the “Enter” key and it will display the result of the expression. Go ahead and try it.

2.3.2 Printing the Result of a Statement

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 (as you saw in the interactions pane). If you do add the semicolon at the end of a Java statement in the interactions pane it will execute the statement but not automatically print the result in the interactions pane.

Even though you do not have to type the semicolon after statements in the interactions pane you must type the semicolon at the end of your statements in the definitions pane or the code will not compile.

Since you will need to provide the semicolon at the end of statements in the definitions pane you should get used to using them in the interactions pane too. But, then how do you show the result of a statement in the interactions pane? The phrase `System.out.println()` is an important one to know. The meaning
Section 2.3 Java Basics 25

for System.out.println() is “Use the PrintStream object known as out on the System class to print out the value of whatever is in the parentheses followed by an end-of-line character.” DrJava will print the result of an expression in the interactions pane when you use System.out.println(expression).

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
> System.out.println(26 - 3);
23
> System.out.println(3 * 4);
12
> System.out.println(4 / 2);
2
> System.out.println(9 % 4);
1
> System.out.println(9 / 5 * -3 + 32);
29
> System.out.println(3 + 2 * 4);
11
> System.out.println((3 + 2) * 4);
20

The code 34 + 56 is a numeric expression that Java understands. Obviously, it’s composed of two numbers and an operation that Java knows how to do, ‘+’ meaning “add.” In Java we call math symbols like ‘+’ and ‘-’ operators. The operator ‘-’ means subtract. The operator ‘*’ means multiply. The operator ‘/’ means divide. The operator ‘%’ means calculate the remainder of the first number divided by the second one.

Notice that you get a different result from System.out.println(3 + 2 * 4); than from System.out.println((3 + 2) * 4);. This is because multiplication has higher precedence than addition (meaning it is done first by default). You can use parentheses to change the default order of evaluation of an expression or
26 Chapter 2 Introduction to Java

to make the order clear.

Common Bug: Matching Parentheses
When you use parentheses you will need an open parenthesis for each close parenthesis. If you don’t have a match you will get an error.

```java
> System.out.println(3 + 2) * 4);
Syntax Error: ")"
> System.out.println((3 + 2 * 4);
Syntax Error: ";"
```

Making it Work Tip: Using Another Development Environment
If you are not using DrJava you will need to type all code that we show in the interactions pane in a main method instead. Compile and execute the class with the main method. To get the above example to work in another development environment I could have written the following class definition in a file named “Test.java”.

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

The next step is to compile the Java source file. This changes it from something people can read and understand into something the computer can read and understand. To compile the source file with the free command line tools from Sun do the following:

```bash
> javac Test.java
```

When you compile a Java source file the compiler will create a class file, so if you compile the source file “Test.java” the compiler will create the file “Test.class”. This file will have the same name as the source file but will have an extension of “.class”. After you have compiled the source you can execute the class. To execute it using the free command line tools from Sun is to use:

```bash
> java Test
```

The ability to try things in the interactions pane without having to create a class and a main method is one of the major advantages to using DrJava.
2.3.3 Types in Math Expressions

Java takes types seriously. If it sees you using integers, it thinks you want an integer result from your expressions. If it sees you using 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
```

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

2.3.4 Casting

We could also have used casting to get the correct result from the division of two integers. Casting is like using a mold to give some material a new shape. It tells the compiler to change a value to a particular type even if it could lead to a loss of data. To cast you put the type that you want the value changed to inside an open and close parenthesis: (type). There are two floating point types in Java: float and double. The type double is larger than the type float and thus more accurate. We will use this type for most of the floating point numbers in this book. Notice that we can cast either the 1 or 2 to double and the answer will then be given as a double. We could cast both the 1 and 2 to double and the result would be fine. However, if we cast the result of the integer division to a double it is too late since the result of integer division of 1 by 2 is 0 since the result is an integer.

```java
> System.out.println((double) 1 / 2);
0.5
> System.out.println(1 / (double) 2);
0.5
> System.out.println((double) (1/2));
0.0
```

2.3.5 Relational Operators

We can write Java statements that do simple math operations. But if that was all we could do computers wouldn’t be terribly useful. Computers can also decide if something is true or false.

```java
> System.out.println(3 > 2);
true
> System.out.println(2 > 3);
false
> System.out.println(‘a’ < ‘b’);
true
```
Using symbols we can check if one value is greater than another '>', less than another '<', equal to another '==', not equal to another '!='; greater or equal to another '>=', and less than or equal to another '<='. You can use these relational operators on many items such as numbers and characters as shown above. A character can be specified between a pair of single quotes ('a').

You might find '==' odd as a way to test for equality. But, in Java '=' is used to assign a value, not check for equality, as you will see in the next section.

Notice that Java understands the concepts true and false. These are re-
served words in Java which means that they can’t be used as names.

**Making it Work Tip: Java Primitive Types**

- Integers are numbers without a decimal point in them. Integers are represented by the types: `int`, `byte`, `short`, or `long`. Example integers are: 3, -2350, 502893. In this book we will use only `int` to represent integers. Each integer takes up 32 bits of memory (4 bytes).

- Floating point numbers are numbers with a decimal point in them. Floating point numbers can be represented by the types: `double` or `float`. Example doubles are 3.0, -19.23, and 548.675. In this book we will use mostly use `double` to represent floating point numbers. Each double in memory takes up 64 bits (8 bytes).

- Characters are individual characters such as can be made with one key stroke on your keyboard. Characters are represented by the type: `char`. Characters are specified inside single quotes, like 'a' or 'A'. Each character in memory takes up 16 bits (2 bytes).

- True and false values are represented by the type `boolean`. Variables of type boolean can only have `true` or `false` as values. While a boolean could be represented by just one bit the size of a boolean is up to the virtual machine.

### 2.3.6 Strings

Computers can certainly work with numbers and even characters. They can also work with strings. Strings are sequences of characters. Try the following in the interactions pane.

```
> System.out.println("Mark");
Mark
> System.out.println("13 + 5");
13 + 5
```

Java knows how to recognize *strings* (lists of characters) that start and end with " (double quotes). Notice what happens when you enclose a math expression like `13 + 5` in a pair of double quotes. It doesn’t print the result of the math expression but the characters inside the pair of double quotes. Whatever is inside a pair of double quotes is not evaluated, the value of it is exactly what was entered.
Now try the following in the interactions pane.

```java
> System.out.println("Barbara" + "Ericson");
BarbaraEricson
> System.out.println("Barbara" + " " + "Ericson");
Barbara Ericson
> System.out.println("Barbara " + "Ericson");
Barbara Ericson
```

You can “add” strings together using a + operator as you see in "Barbara" + "Ericson". It simply creates a new string with the characters in the first string followed by the characters in the second string. This is called appending strings. Notice that no space is added automatically. If you want space in your string you will need to put it there using a space between a pair of double quotes as shown above with "Barbara" + " " + "Ericson". Or you can have a space inside a string as shown in "Barbara " + "Ericson".

Now try the following in the interactions pane.

```java
> System.out.println("The total is " + (13 + 5));
The total is 18
> System.out.println("The total is " + 13 + 5);
The total is 135
```

You can “add” a string and a number. It will turn the number into a string and then append the two strings. This does what you would expect to show the result of "The total is " + (13 + 15) but you may not expect what happens with "The total is " + 13 + 5.

The computer evaluates statements from left to right so the computer evaluates this as “add” the string "The total is" to the number 13 by turning the number 13 into a string "13". Next it sees the + 5 as adding a number to the string "The total is 13". It turns the second number into a string and results in The total is 135.

The way to get what you would expect is to use parentheses to enclose the math expression. Just like in algebra the parentheses change the order things are evaluated. The (13 + 5) will be evaluated before the append of the string and the resulting number 18.

If you want to put a double quote inside of a string you will need some way to tell the computer that this isn’t the ending double quote. In Java the backslash \ character is used to treat the next character differently. So using \" results in a double quote inside a string. Some other special characters are \n to force a new line and \t to force a tab.

```java
> System.out.println("Barb says, \"Hi\".");
Barb says, "Hi."
> System.out.println("This is on one line.\nThis is on the next");
This is on one line.
This is on the next
```
2.4 VARIABLES

We have used Java to do calculations and to append strings, but we have not stored the results. The results would be in memory but we don’t know where they are in memory and we can’t get back to them. On a calculator we can store the result of one calculation to memory (Figure 2.4). We can then use that stored value in other calculations. On a calculator you also have access to the result of the last calculation.

![Figure 2.4: A calculator with a number in memory](image)

2.4.1 Declaring Variables

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

When you declare a variable in Java you specify the type for the variable and a name (*type name*). You need to specify a type so that Java knows how many bits to reserve in memory and how to interpret the bits. You can also assign a value to a variable using the ‘=’ operator and provide a value or an expression (*type name = expression*). Don’t read ‘=’ as equals but as assign the value of the right side to the variable on the left (which makes using ‘==’ for equals make more sense). The bits in the variable will be set to represent the value. We will use the type `int` for storing integer values (numbers without decimal points) and the type `double` for storing floating point values (numbers with decimal points).

2.4.2 Using Variables in Calculations

What if you want to calculate the total bill for a meal including the tip? You would start with the bill value and multiply it by the percentage you want to tip (20%), that would give you the tip amount. You could then add the tip amount to the bill total to get the total amount to leave.

We will use type `double` to store the bill amount, tip, and total amount since they can have numbers after a decimal point. If we also wanted to calculate the cost per person we could divide the total by the number of people. We could use an integer variable to hold the number of people.
> int numPeople = 2;
> System.out.println(numPeople);
2
> double bill = 32.45;
> System.out.println(bill);
32.45
> double tip = bill * 0.2;
> System.out.println(tip);
6.490000000000001
> double total = bill + tip;
> System.out.println(total);
38.940000000000005
> double totalPerPerson = total / numPeople;
> System.out.println(totalPerPerson);
19.470000000000002

So, each person would need to pay 19.47, which they would probably round up to 19.50.

Making it Work Tip: Variable Names
By convention the first word in a variable name is lowercase. So if the variable name is just one word then the whole thing is lowercase such as bill. The first letter of each additional word in a variable name should be uppercase, as shown by the variables named numPeople and totalPerPerson. This is a Java convention (usual way something is done) and it will make your programs easier for others to read.

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

2.4.3 Memory Maps of Variables
In Java when you declare variables to be of the type int or double you are asking the computer to set aside space for a variable of that type (32 bits for int and 64 for double) and to remember the address of that space. When you assign a value to a variable using the ‘=’ operator you change the value in that space to represent the new value. The code int numPeople reserves 32 bits of space and associates the name “numPeople” with that reserved space (Figure 2.5). The code = 2 sets the value of that space to the integer value 2. The code double bill reserves 64 bits of space and associates the name “bill” with that space. The = 32.45 changes the values in the reserved space to represent the value 32.45.

When we print out a variable name using System.out.println(bill); the
Section 2.4 Variables

33

int numPeople = 2;

Reserves 32 bits (4 bytes)
and sets the value stored
in that space to 2. The
name "numPeople" is
associated with this space.

double bill = 32.45;

Reserves 64 bits (8 bytes)
and sets the value stored
in that space to 32.45. The
name "bill" is associated
with this space.

FIGURE 2.5: Declaring primitive variables and memory assignment

computer looks up the name bill to find the address of that variable in memory
and prints the value stored in that space. It knows how many bytes to use and
how to interpret the bytes in calculating the value based on the declared type of
the variable.

How would you calculate the cost of a shirt that was originally $47.99 dollars
but was 40% off? And, what if you also had a coupon for an additional 20% off the
sale price? First you would need to determine the first discount amount by multi-
plying 40% (0.40) times the original price. Next, calculate the first discount total
by subtracting the first discount amount from the original price. Next calculate
the second discount amount by multiplying 20% (0.20) times the second discount
amount. The second discount total is the first discount total minus the second
discount amount. We would need variables to hold the first discount amount, first
discount total, second discount amount, and second discount total. What type
should those variables be declared to be? Since they have fractional parts they can
be declared as double.

> double originalPrice = 47.99;
> double firstDiscountAmount = originalPrice * 0.40;
> System.out.println(firstDiscountAmount);
19.196
> double firstDiscountTotal = originalPrice - firstDiscountAmount;
> System.out.println(firstDiscountTotal);
28.794
> double secondDiscountAmount = firstDiscountTotal * 0.20;
> System.out.println(secondDiscountAmount);
5.758800000000001
> double secondDiscountTotal = firstDiscountTotal -
  secondDiscountAmount;
> System.out.println(secondDiscountTotal);
When these statements are executed 64 bits of space is reserved for each variable declared as a double. So how much memory does this calculation take? We have declared 5 doubles so we have used 5 times 64 bits of space. Each byte has 8 bits in it so how many bytes have we used? How much memory does your computer have and how much of it have you used? If your computer has 128 Megabytes of memory then that is 128000 bytes of memory and we used 40 bytes then we have only used 0.03125% of memory. That isn’t very much. We can declare lots of variables and still not use up all of the memory.

Each time we use the variable name above the computer substitutes the value in the memory location associated with that name. What are the values in each of the 5 declared variables when these statements are finished?

### 2.4.4 Object Variables

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

![Diagram showing the parent and child relationship between Mammal and Dog](image)

**FIGURE 2.6:** Showing the parent and child relationship between mammal and dog (left) and `Object` and `String` (right)

You can think of inheritance as saying that one class “is a kind of” another class like saying that a dog is a kind of mammal (Figure 7.14). If you need a mammal you can use a dog, but if you need a dog another mammal won’t do. Because a dog is a kind of mammal we know that it has the same characteristics that a mammal does such as breathing oxygen, bearing live young, having hair, etc. We say that it inherits characteristics from mammal. The `String` class is a child of the `Object` class so it is a kind of object (Figure 7.14). All of the classes that you define will inherit from the `Object` class either directly or indirectly.

When you declare a variable you specify the type and a name `type name;` or `type name = expression;`. What if you want to declare a variable that will refer to a string? What type can you use? Well it can’t be `int` or `double` because those represent numbers. It can’t be `char` because that represents a single character.

Java has a class `String` that is used to represent character strings. The `String` class inherits from the `Object` class. So to declare a variable that represents a string
Section 2.4 Variables

Object variables reserve space for something which refers to the address of the object. Object variables do not reserve space for the object. If the object variable doesn't reference any object yet it has the value null. The value null is a keyword that Java understands.

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

When the variable `test` was declared as type `String` space was reserved for an object reference (a way to find the address of the object) and the value was set to `null` (Figure 2.7). The default value for an object variable is `null` which means it isn't referring to any object yet. The compiler will create a `String` object when it sees characters enclosed in double quotes so the "Hi" creates an object of the `String` class and sets the characters in that `String` object to be the characters
"Hi". The code `test = "Hi"` changes the value of the space reserved for the object reference from null to a reference to the `String` object with the characters "Hi".

What happens to the `String` object with the characters "Hi" in it when you changed the variable `test` to refer to the new `String` object with the characters "Bye"? Java keeps track of used space and if there are no valid references to the used space it will put it back into available space. This is called garbage collection.

The fact that Java automatically handles freeing used memory when it is no longer needed is one of the advantages to Java over languages like C++ which required the programmer to free memory when it was no longer needed. Programmers aren’t very good at keeping track of when memory is no longer needed and so many programs never free memory when it is no longer needed. This is called a memory leak and it is why some programs use more and more memory while they are running. Sometimes programmers free memory when it is still being used which can cause major problems such as incorrect results and even cause your computer to crash.

### 2.4.5 Reusing Variables

Once we have declared variables we can reuse them by assigning new values to them.

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

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

#### Making it Work Tip: Variables versus Literals

Notice that we have changed the value of the variable `test` several times. We call items like `test` variables because the values inside of them can change. This is different from literals such as the string literal "Hi" in that the value of that won’t change. You can set the value of a variable to a literal but you can’t set the value of a literal to a variable.

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);
```
String myName = "Mark";
Creates a String object and
sets the characters in that
object to "Mark". Sets
the value in the object
reference "myName" to
refer to the String object.

myName

Mark

myName = "Barb";
Creates a String object and
sets the characters in that
object to "Barb". Changes
the value in the object
reference "myName" to refer
to the new String object. The
String object with "Mark" in
it still exists but can be
garbage collected since
there is no reference to it.

Mark

Barb

Mark

FIGURE 2.8: Shows creation and reuse of an object variable.

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 (by clicking on the Reset button). The relationship between names and data in the interactions pane only exists during a session of DrJava.

Common Bug: Redefinition Error
You can't declare a variable with the same name more than once in the interactions pane. If you do you will get a "Redefinition Error". If you want to "start over" click the Reset button in DrJava to let it know that you want to get rid of all currently defined variables. Or, just remove the types and you won't be redeclaring the variables, just changing their values (reusing them).

2.4.6 Multiple References to an Object
You can have several variables that reference an object. You can use any of the references to access the object.

> String name1 = "Suzanne Clark";
> System.out.println(name1);
Suzanne Clark
> String name2 = name1;
When the compiler encounters the characters inside the pair of double quotes it creates a *String* object. The code `String name1` creates a variable `name1` that will refer to this string object. Print out `name1` to see what it refers to using `System.out.println(name1);`. Next the code `String name2 = name1;` creates another variable `name2` and sets the value of it to refer to the same string. Printing the new variable `name2` will result in the same string being printed.

> System.out.println(name2);

Suzanne Clark

An object can only be garbage collected when there are no current references to it. To allow the *String* object with the characters "Suzanne Clark" in it to be garbage collected set the variables that refer to it to null.

```java
> name1 = null;
> System.out.println(name1);
null
> System.out.println(name2);
Suzanne Clark
> name2 = null;
> System.out.println(name2);
null
```

Now all references to the *String* object are set to null so the object can be garbage collected.

```java
> System.out.println(name2);
null
> System.out.println(name2);
null
```

![FIGURE 2.9: An object with multiple references to it](image1)

![FIGURE 2.10: An object with no references to it](image2)
2.5 CONCEPTS SUMMARY

This chapter introduced many basic concepts: printing the result of a statement (expression), math operators, relational operators, types, casting, and variables.

2.5.1 Statements

Java programs are made of statements or expressions. Java statements can end in semicolons ';' just like sentences can end in periods '.' in English. When you type statements in the definitions pane (when you define methods) they must have some sort of punctuation to show the end of the statement. One way to do this is to use a semicolon ';'.

If you leave off the semicolon in the interactions pane it will print the result of the statement. If you do end a statement with a semicolon in the interactions pane, and you want to print the result use `System.out.println(expression);` to print the result of the expression.

```java
> int numPeople = 3;
> double bill = 52.49;
> double amountPerPerson = bill / numPeople;
> System.out.println("Each person should pay: " + amountPerPerson);
Each person should pay: 17.496666666666666
```

Math Operators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
<td>Used to add numbers together (3 + 4) = 7</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>Used to subtract one number from another (5 - 2) = 3</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>Used to multiply two numbers together (2 * 3) = 6</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>Used to divide one number by another (18 / 2) = 9</td>
</tr>
<tr>
<td>%</td>
<td>Modulus (Remainder)</td>
<td>Used to find the remainder of one number divided by another (19 % 2) = 1</td>
</tr>
</tbody>
</table>
2.5.2 Relational Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>Less Than</td>
<td>Used to check if one value is less than another (2 &lt; 3) = true</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater Than</td>
<td>Used to check if one value is greater than another (3 &gt; 2) = true</td>
</tr>
<tr>
<td>==</td>
<td>Equals</td>
<td>Used to check if two values are the same (2 == 2) = true</td>
</tr>
<tr>
<td>!=</td>
<td>Not Equals</td>
<td>Used to check if two values aren't equal (2 != 3) = true</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less Than or Equal</td>
<td>Used to check if one value is less than or equal to another (2 &lt;= 3) = true</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater Than or Equal</td>
<td>Used to check if one value is greater than or equal to another (3 &gt;= 2) = true</td>
</tr>
</tbody>
</table>

2.5.3 Types

A type is a description of the “kind of” thing something is. It affects how much space is reserved for a variable and how the bits in that space are interpreted. In this chapter, we talk about several kinds of types (encodings) of data.

<table>
<thead>
<tr>
<th>Type</th>
<th>Java primitive types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floating point numbers</td>
<td>Java primitive types double or float e.g., 5.2, -3.01, 928.3092</td>
<td>Numbers with a decimal point in them.</td>
</tr>
<tr>
<td>Integers</td>
<td>Java primitive types int, byte, short, long e.g., -3, 5239, 0</td>
<td>Numbers without a decimal point—they can’t represent fractions.</td>
</tr>
<tr>
<td>Characters</td>
<td>Java primitive type char e.g., 'a', 'b', '?'</td>
<td>A character is delimited by a pair of single quotes.</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>Booleans</td>
<td>Java primitive type boolean with only two possible values</td>
<td>The value of a boolean can be the reserved word true or the reserved word false.</td>
</tr>
</tbody>
</table>

2.5.4 Casting

Java compilers recognize integer (-3) and floating point values (32.43). The result of a mathematical expression depends on the types involved in the expression. Expressions that involve integer values will have integer results. Expressions that have floating point (decimal) values will have floating point results.

This can lead to unexpected results.

> 1 / 2
There are two ways to fix this problem. One is to make one of the numbers a floating point number by adding '.0' (it doesn’t matter which one) and the other is to use casting to change the type of one of the numbers to a floating point number (the primitive type float or double).

\[
\begin{align*}
> 1.0 / 2 \\
& 0.5 \\
> (\text{double}) 1 / 2 \\
& 0.5
\end{align*}
\]

2.5.5 Variables

Variables are used to store and access values. You create variables by declaring them: type name; or type name = expression; Declaring a variable reserves space for the variable and allows the computer to map the variable name to the address of that reserved space.

We introduced two types of variables: primitive and object. Primitive variables are any of the types: int, byte, short, long, float, double, char, or boolean. Object variables refer to an object of a class. Use the class name as the type when declaring object variables ClassName name; or ClassName name = expression;.

Primitive variables store a value in the reserved space for that variable. You can change the value using variableName = value;. You can access the value using variableName.

Object variables store a reference to an object in the reserved space for that variable. Object variables do not just store the address of the object. They store something that allows the address to be determined.

If the object variable doesn’t refer to any object yet it has the value null. You can change the what object it references using variableName = objectReference;. You can access the referenced object using variableName.

PROBLEMS

2.1. Some computer science concept questions:

- What is an object?
- What is a class?
- What is a type? Why are types important?
- What is casting? What is it used for?
- What is a variable? When do you need one?

2.2. What objects would you encounter in a bank?

- What objects would you encounter in going to a movie?
- What objects would you encounter in a clothing store?
- What objects are involved in a soccer game?
- What objects are involved in an airplane flight?
2.3. Use the interactions pane to calculate how long it will take to travel 770 miles at an average speed of 60 miles per hour? How much shorter will it take if you average 70 miles per hour?
2.4. Use the interactions pane to calculate how much money will you make if you work 40 hours at $13.00 and 10 hours at time and a half?
2.5. Test your understanding of Java with the following:
   • What does `System.out.println();` do?
   • What does the statement `System.out.println(3 + 2);` do?
   • What does the statement `System.out.println("The answer is: " + 3 + 2);` do?
   • What does the statement `System.out.println("Hi " + " there");` do?
   • What does the code `int x = 3; System.out.println("The result is " + x);` do?
   • What does the code `String firstName = "Sue"; System.out.println(firstName);` do?
2.6. Declare variables for each of the following:
   • the number of people in your family
   • the cost of a video game
   • your name
   • answer to, "Are you righthanded?"
   • the temperature in your room
   • the number of items in a shopping cart
   • your grade point average
   • your telephone number
   • the number of times you were absent from class
   • the number of miles from your home to school
   • answer to, "Do you wear glasses?"
   • your credit card number

TO DIG DEEPER

There is a wealth of material for Java on Sun’s Java web site http://java.sun.com including tutorials, papers, and APIs. To learn more about DrJava see the web site http://www.drjava.org/. Thinking in Java by Bruce Eckel is a good book for those who have some coding experience and like to understand a language deeply. Beginners might want to start with Headfirst Java by Kathy Sierra and Bert Bates.
CHAPTER 3

Introduction to Programming

3.1 PROGRAMMING IS ABOUT NAMING
3.2 FILES AND THEIR NAMES
3.3 CLASS AND OBJECT METHODS
3.4 WORKING WITH TURTLES
3.5 CREATING METHODS
3.6 WORKING WITH MEDIA
3.7 CONCEPTS SUMMARY

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

- To create a World object and Turtle objects and move the turtles to draw shapes.
- To create Picture objects and show them.
- To create Sound objects and play them.

The computer science goals for this chapter are:

- To invoke class and object methods.
- To create objects using the new keyword.
- To write methods (functions).

3.1 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 string of letters; with a media element like a file, sound, or picture; or even with more abstract concepts, like a named recipe (a program or method) or a named encoding (a type or class).

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

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

A programming language is really a set of names that a computer has en-
codings 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 names—which allows us to create our own layers of encod-
ing. We can associate a name with a location in memory, this is called declaring
a variable. We can associate a name with a group of Java statements, we call this
defining a method (function). In Java you can also assign a name to a group of
related variables and methods (functions) when you define a class (type).

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

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

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

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

How the units and values (data) of a program can be interpreted is often also
named. Remember how we said in Section 1.2 (page 13) that everything is stored
in groups of eight bits called bytes, and we can interpret those bytes as numbers?
In some programming languages, you can say explicitly that some value is a byte,
and later tell the language to treat it as a number, an integer (or sometimes int).
Similarly, you can tell the computer that these series of bytes is a collection of
numbers (an array of integers), or a collection of characters (a String), or even
as a more complex encoding of a single floating point number (any number with a
decimal point in it).
Section 3.2 Files and their Names

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

3.2 FILES AND THEIR NAMES

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

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

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

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

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

Think about your computer’s memory as your desk. You would want to keep books that you are currently working with on your desk but when you are done you will probably move those books to a book shelf. You may have many more books and want to keep them in your home library.

In contrast, your disk drives store files that you have already worked with, and you want to keep them for future reference. This is why we have disk drives to store our files, and why we want to keep them on our hard disk drives, not in our memories.
on your book shelf than can fit on your desk. A computer can fit much more data on the hard disk than can fit in memory. However, data must be read from disk into memory before you can work with it.

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

### 3.3 CLASS AND OBJECT METHODS

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

However, you can’t just call a function or method in Java like you can in some other languages. Every method or function in Java must be defined inside a class. There are two types of methods in Java: class methods or object methods. Class methods are methods that can be invoked using the class name or on an object of the class. Object methods can only be invoked on an object of the class. Class methods are used for general methods that don’t pertain to a particular object and object methods work with a particular object’s data.

#### 3.3.1 Invoking Class Methods

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

Making it Work Tip: Wrapper Classes

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

One of the class methods for the Character class takes a character as the input value (the value that goes into the box) and returns (the value that comes out of the box) the number that is the integer value for that character. Characters
Section 3.3 Class and Object Methods

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 numeric value.

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

### Debugging Tip: Common typos

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

```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'
```

### 3.3.2 Invoking Object Methods

Object methods are methods that must be invoked on an object using: objectReference.methodName()

An object reference can be the name of an object variable. You can’t invoke object methods using the class name like you can with class methods.

In Java there is a String class which is how you represent lists of characters (letters), like the letters of a person’s name. Objects of the String class are created by the compiler whenever it sees string literals (characters enclosed with double quotes), like "Barbara" or "cat.jpg". The double quotes tell the compiler that this is an object of the String class and not a variable name.

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

```java
> String name = "Fred Farmer";
> System.out.println(name);
```

Chapter 3  Introduction to Programming

Fred Farmer

> String lowerName = name.toLowerCase();
> System.out.println(lowerName);
fred farmer

> String upperName = name.toUpperCase();
> System.out.println(upperName);
FRED FARMER

> System.out.println(name);
Fred Farmer

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

3.4 WORKING WITH TURTLES

Seymour Papert, at MIT, used robot turtles to help children think about how to specify a procedure in the late 1960s. The turtle had a pen in the middle of it that could be raised and lowered to leave a trail of its movements. As graphical displays became available he used a virtual turtle on a computer screen.

We are going to work with some turtle objects that move around a world. The turtles know how to move forward, turn left, turn right, and turn by some specified angle. The turtles have a pen in the middle of them that leaves a trail to show their movements. The world keeps track of the turtles that are in it.

3.4.1 Defining Classes

How does the computer know what we mean by a world and a turtle? We have to define what a world is, what it knows about, and what it can do. We have to define what a turtle is, what it knows about, and what it can do. We do this by writing class definitions for World and Turtle. In Java each new class is usually defined in a file with the same name as the class and an extension of “.java”. Class names start with a capital letter and the first letter of each additional word is capitalized. So we define the class Turtle in the file Turtle.java. We define the class World in the file World.java. The class Turtle inherits from a class called SimpleTurtle (notice that the first letter of each additional word is capitalized). We have defined these classes for you so that you can practice creating and sending messages to objects.

3.4.2 Creating Objects

Object-oriented programs consist of objects. But, how do we create those objects? The class knows what each object of that class needs to keep track of and what it should be able to do, so the class creates the objects of that class. You can think of the class as an object factory. It can create many objects of that class.

To create and initialize an object use new Class(parameterList) where the parameter list is a list of items used to initialize the new object. This asks the object that defines the class to reserve space in memory for the data that an object of that
class needs to keep track of and also keep a reference to the object that defines the class. The new object’s data will be initialized based on the items passed in the parameter list. There can be several ways to initialize a new object and which one you are using depends on the order and types of things in the parameter list.

One way to create an object of the class World use `new World()`. We don’t have to pass any parameters to initialize the new world. Objects can have default values.

> System.out.println(new World());
A 640 by 480 world with 0 turtles in it.

![Figure 3.1: A window that shows a World object.](image)

When you type the above in the interactions pane you will see a window appear with the title “World” as shown in Figure 3.1. We have created an object of the World class which has a width of 640 and a height of 480. The world doesn’t have any turtles in it yet. We would like to add a turtle to this world, but we have a problem. We don’t have any way to refer to this World object. We didn’t declare a variable that refers to that object in memory, so it will just be garbage collected after you close the window. Go ahead and close the window and let’s try again, but this time we will declare a variable to let us refer to the World object again.

When we declare a variable we are associating a name with the memory location so that we can access it again using it’s name. To declare a variable in Java you must give the type of the variable and a name for it `Type name;` The `Type` is the name of the class if you are creating a variable that refers to an object. So to create a variable that will refer to a World object we need to say the type is World and give it a name. The first word in the variable name should be lowercase but the first letter of each additional word should be uppercase. The name should describe what the variable represents. So, let’s declare a variable that refers to an object of the class World using the name `worldObj`.
Chapter 3  Introduction to Programming

> World worldObj = new World();
> System.out.println(worldObj);
A 640 by 480 world with 0 turtles in it.

This says to create a variable with the name of worldObj that will be of type World (will refer to an object of the class World). It will refer to the object created by the World class because of the code: new World(). We can use System.out.println(worldObj) to ask the new World object to print out some information about it.

To create a turtle object in this world we will again use new Class(parameterList). This time we will ask the Turtle class to create the object in our World by passing a reference to the world to create it in. We will declare a variable so that we can refer to the Turtle object again.

> Turtle turtle1 = new Turtle(worldObj);
> System.out.println(turtle1);
No name turtle at 320, 240 heading 0.

Now a Turtle object appears in the middle of the World object as shown in Figure 3.2. This turtle hasn’t been assigned a name and has a location of (320,240) and a heading of 0 which is north. The default location for a new turtle is the middle of the World object. The default heading is 0 (north).

We can create another Turtle object and this time we can say what location we want it to appear at. To do this we need to pass more than one parameter in the parameter list of items to use to initialize the new object. To do this separate the values with commas.

> Turtle turtle2 = new Turtle(30,50,worldObj);
> System.out.println(turtle2);
No name turtle at 30, 50 heading 0.
Notice that the second turtle appears at the specified location (30,50). The top left of the window is location (0,0). The x values increase going to the right and the y values increase going down.

![A window that shows two Turtle objects in a World object.](image)

### 3.4.3 Sending Messages to Objects

In object-oriented programming we send messages to objects to ask them to do things. The full syntax for sending a message is

```
objectReference.message(parameterList);
```

The `objectReference` is a reference to an object, `message` is what we want the object to do, and `parameterList` is any additional information that more fully describes what we want the object to do. The `.` and `()` are required even if there is no parameter list.

Turtles know how to go forward, turn left, turn right, turn by a specified angle, change their color, and set their names. So if we want turtle1 to go forward 20 steps we would use `turtle1.forward(20);`. If we want it to turn left we would use `turtle1.turnLeft();`. If we want it to turn right we would use `turtle1.turnRight();`. If we want it to turn by an angle to the left by 45 degrees we would use `turtle1.turn(-45);`. To turn turtle1 to the right 45 degrees use `turtle1.turn(45);`. Negative angles turn to the left and positive angles turn that amount to the right.

```java
> turtle1.forward(20);
> turtle1.turnLeft();
> turtle1.forward(30);
> turtle1.turnRight();
> turtle1.forward(40);
> turtle1.turn(-45);
> turtle1.forward(30);
```
> turtle1.turn(90);
> turtle1.forward(20);

![turtle1](image1)

**FIGURE 3.4:** The result of messages to the first Turtle object.

In Figure 3.4 we see the trail of the first turtle’s movements. Notice that all of the messages were sent to the first Turtle object that is referenced by the turtle1 variable. The messages only get sent to that object. Notice that the second Turtle object didn’t move. It didn’t get any messages yet. To send a message to the second Turtle object we use the variable name that refers to that Turtle object which is turtle2.

> turtle2.turnRight();
> turtle2.forward(200);
> turtle2.turnRight();
> turtle2.forward(200);

In Figure 3.5 we see the trail of the second turtle’s movement. Can you draw a square with a turtle? Can you draw a triangle with a turtle? Can you draw a pentagon with a turtle? How about a circle?

### 3.4.4 Objects Control Their State

In object-oriented programming we ask an object to doing something by sending it a message. The object can refuse to do what you ask it to do. Why would an object refuse? An object *should* refuse when you ask it to do something that would cause it’s data to be wrong. The world that the turtles are in is 640 by 480. Try asking the Turtle object to go forward past the end of the world. What happens? First click the Reset button to reset the interactions pane. When you reset the interactions pane you get rid of any currently declared variables. Then create a new World and Turtle.
Section 3.4 Working with Turtles

> World world1 = new World();
> Turtle turtle1 = new Turtle(world1);
> System.out.println(turtle1);
No name turtle at 320, 240 heading 0.
> turtle1.turnRight();
> turtle1.forward(400);
> System.out.println(turtle1);
No name turtle at 639, 240 heading 90.
> System.out.println(world1.getWidth());
640

Remember that Turtle objects are first created in the middle of the world (320,240) facing the top of the world. When the turtle turned right it was facing the right side of the window. If the turtle went forward 300 steps it would it would be past the right edge of the window (320 + 400 = 720) since the x values increase to the right. Notice that the turtle stops when the middle of it reaches the limit of the window (639) Figure 3.6. This means your turtle will always have at least part of it in the world.

It may seem strange that turtle stopped when it reached 639 but the first pixel is at 0 and the last is 639. If I asked you to count 10 numbers starting at 0 you should end at 9. The number of items is the ending value minus the starting value plus 1. So 639 - 0 + 1 is 640, which means that a window with a width of 640 that starts with 0 must end at 639.

3.4.5 Additional Turtle Capabilities

You may not want to see the turtle, but just the trail of it’s movements. To ask the turtle to stop drawing itself, send it the message hide(). To start drawing the turtle again send it the message show().
On the other hand you may not want to see the trail. Ask the turtle to stop showing the trail by asking it to pick up the pen \texttt{penUp()}. To start showing the trail again send the turtle the message \texttt{penDown()}. You can ask a turtle to move to a particular location by sending it the message \texttt{moveTo(x,y)} where \(x\) is the \(x\) value that you want to move to and \(y\) is the \(y\) value that you want to move to.

You can ask a turtle to use a particular name by sending it the message \texttt{setName(name)} where \texttt{name} is the new name to use. If you print the variable that refers to a turtle you will see the name printed. You can also get a turtle’s name by sending it the message \texttt{getName()}. We can use these new messages to draw two squares with a turtle. First reset the interactions pane and create a world and a turtle. Name the turtle “Jane”. Draw one square with an upper left corner at (50,50) and a width and height of 30. Draw another square at (200,200) with a width and height of 30. We can use the way to create a \texttt{Turtle} object that takes a location to start \texttt{new Turtle(x,y,world)}. Remember that to create an object we ask the class using \texttt{new Class(parameterList)}. Let’s turn off seeing the turtle when we draw the second square by sending it the message \texttt{hide()}.

```java
> World world1 = new World();
> Turtle turtle1 = new Turtle(50,50,world1);
> turtle1.setName("Jane");
> turtle1.turnRight();
> turtle1.forward(30);
> turtle1.turnRight();
> turtle1.forward(30);
> turtle1.turnRight();
> turtle1.forward(30);
> turtle1.turnRight();
> turtle1.forward(30);
> turtle1.turnRight();
```
Section 3.4 Working with Turtles

> turtle1.forward(30);
> turtle1.penUp();
> turtle1.moveTo(200,200);
> turtle1.hide();
> turtle1.penDown();
> turtle1.turnRight();
> turtle1.forward(30);
> turtle1.turnRight();
> turtle1.forward(30);
> turtle1.turnRight();
> turtle1.forward(30);
> turtle1.turnRight();
> turtle1.forward(30);
> turtle1.forward(30);
> turtle1.forward(30);
> System.out.println(turtle1);
Jane turtle at 200, 200 heading 0.

FIGURE 3.7: Drawing two squares with a turtle.

You can see the result of these commands in Figure 3.7.

Making it Work Tip: Reuse the previous line in DrJava
You can use the up arrow on the keyboard to bring up previous lines you have typed in the interactions pane in DrJava. This is easier than typing the same line in again.
3.5 CREATING METHODS

We had to send many messages to our Turtle object just to draw two squares. Do you notice any similarities in how we draw the squares? Each time we draw a square we turn right and go forward by 30 steps for a total of 4 times. It would be nice to name the list of steps for drawing a square and then just do the list of steps when a turtle is asked to draw a square. We do this by creating a method that knows how to draw a square. Methods are named blocks of commands that are defined inside a class definition. Once we have defined a method and successfully compiled the class definition the objects of the class will respond to a message with the same name and parameters as the new method. So if we want Turtle objects to understand the message drawSquare() we define a method drawSquare().

Computer Science Idea: Messages Map to Methods
When we send an object a message it must map to a method that objects of that class understand. If objects of the class don’t understand the message you will get an error when you compile. Be sure that the parameter list is correct because if it isn’t you will get an error that says such a method does not exist. Make sure that you compile after you add a new method before you try and use it.

You have seen how you declare variables in Java

\[ \text{type name}; \text{ or type name} = \text{expression}; \]

To declare a method in Java use:

\[ \text{visibility type methodName}\left(\text{parameterList}\right) \]

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

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

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

By convention method names start with a lowercase letter and the first letter of each additional word is uppercase: `turnRight()`. The name of this method is `turnRight`. 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. The type can be any primitive type or class name. The name can be used by the code in the body of the method to refer to the passed value.

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

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

---

**Debugging Tip: Proper Method Declarations**

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

We can now define our first program (method)! Open Turtle.java by clicking on the Open button near the top of the window and using the file chooser to pick “Turtle.java”. Type the following code into the definitions pane of DrJava before the last closing curly brace '}' (which ends the class definition). When you’re done, save the file and click the Compile All button near the top of the window.
(Figure 3.8).

Program 2: Draw a Square

```
public void drawSquare ()
{
    this.turnRight ( ) ;
    this.forward (30) ;
    this.turnRight ( ) ;
    this.forward (30) ;
    this.turnRight ( ) ;
    this.forward (30) ;
    this.turnRight ( ) ;
    this.forward (30) ;
}
```

Making it Work Tip: Copying and pasting

Text can be copied and pasted between the interactions pane and definitions pane. To copy text select it and click copy (in the Edit menu), then click in the definitions pane and click on paste (also in the Edit menu). You can also use keyboard shortcuts for copy (Control-c) and paste (Control-v). This means to hold the “Ctrl” key and then press the “c” key to copy and hold the “Ctrl” key and the ‘v’ key to paste. You can copy entire methods in the definitions pane by selecting the text in the method and then copying and pasting it. You can select a method name in the definitions pane and paste it in the interactions pane to send a message asking for that method to be executed. You can also try things out in the interactions pane and later save them in a method in the definitions pane.

Notice that we changed turtle1.turnRight() to this.turnRight(). The variable turtle1 isn’t defined inside the method drawSquare(). Variables names are known in a context (area that they apply). The variables that we define in the interactions pane are only known in the interactions pane, they aren’t known inside methods. We need some other way to reference the object that we want to turn. Object methods are implicitly passed a reference to the object the method was invoked on. You can refer to that current object using the keyword this.

Compiling a Java class definition “Turtle.java” will produce a “Turtle.class” file. Compiling translates the Java source code which is in a format that humans understand into a format that computers understand. One of the advantages to Java is that the “.class” files aren’t specific to any particular type of computer. They can be understood by any computer that has a Java run-time environment. So you can create your Java source code on a Window’s based computer and run
Section 3.5  Creating Methods  59

3. Try new method here

1. Type method here
2. Compile

FIGURE 3.8: Defining and executing drawSquare()

the compiled code on an Apple computer.

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

This code creates a method with the name drawSquare that takes no parameters and whenever the method is executed it will execute the statements inside of the open and close curly braces. It is a public method. It doesn’t return anything so it uses the keyword void to indicate this. This method must be called on an object of the Turtle class. The this is a keyword that refers to the object this method was invoked on. Since this method is defined in the Turtle class the keyword this will refer to a Turtle object.

Once the method has successfully compiled you can ask for it to be executed by sending a message to a Turtle object with the same name and parameter list as the method. Click on the INTERACTIONS tab in the interactions pane (near the bottom of the window). This method doesn’t take any parameters so just finish with the open and close parenthesis and the semicolon. When you compile the interactions pane will be reset, meaning that all the variables that we have defined
in the interactions pane will no longer be understood. We will need to create a
World and Turtle object again.

> World world1 = new World();
> Turtle turtle1 = new Turtle(world1);
> turtle1.drawSquare();

Whenever the Java run-time encounters a name of a new class it will first
load the class definition from the “.class” file. This creates an object that defines
the class Figure 3.9. This object knows how to create objects of that class and
contains the code for the methods defined in that class. So when you type World
world1 = new World(); the “World.class” file is loaded and an object is created
that represents the definition of the World class. That object is asked to create
a new World object. Enough memory is reserved to hold the data (fields) for the
object and a reference to the object that defines the class. A reference to the new
World object is stored in the space reserved for the variable world1.

When you invoke the method drawSquare() on the Turtle object referenced
by the variable turtle1 the code for that method is found in the object that
defines the Turtle class and the method is passed the object referenced by turtle1.
Inside the method you can refer to the passed object using the keyword this. The
instructions in the body of that method are executed.

FIGURE 3.9: Diagram of an object

What if we want to draw a larger or smaller square? We could change each
of the this.forward(30); lines to the new width and height and then compile.
But, it would be easier to declare a variable in the method that would represent the
width of the square and then use that variable name as the amount to go forward
by like this: this.forward(width);. Then if we want to change the size of the
square we only have to change 1 line. You can declare a variable anywhere in the
body of a method but you must declare it before you use it. The name will be
known and the value substituted for each occurrence of the name in the rest of the
method. But, the name will only be known inside the method it is declared in.

We can’t have two methods with the same name and the same parameter list
so we need a new name for this method. We simply named it drawSquare2 to show
that it is the second version. We can copy the first method and paste it and rename it and then change it to declare and use the width variable.

Program 3: Draw Square Using a Variable for Width

```java
public void drawSquare2() {
    int width = 30;
    this.turnRight();
    this.forward(width);
    this.turnRight();
    this.forward(width);
    this.turnRight();
    this.forward(width);
    this.turnRight();
    this.forward(width);
}
```

Compile and run this method and check that you get the same results as with drawSquare().

> World world = new World();
> Turtle turtle1 = new Turtle(world);
> turtle1.drawSquare2();

3.5.1 Methods that Take Input

This is a bit better than the first version and a bit easier to change. But, you still have to recompile after you change the width to draw a larger or smaller square. Wouldn’t it be nice if there was a way to tell the method what size you want when you ask for the method to be executed by sending a message that matches the method? Well you can! That is what the parameter list is for.

We can make the width of the square a parameter. Remember that if a method takes a parameter you must list the type and name for the parameter in the parameter list. What type should we use for width? Well, in the second version we used int because the turtle only takes whole steps not fractional ones so let’s use that. What should we call this method? We could call it `drawSquare3(int width)` but someone may thing this means it draws a square with a width of 3. We could call it `drawSquareWithPassedWidth(int width)` but that is rather long and you can tell it takes a passed width by looking at the parameter list. How about if we just call it `drawSquare(int width)`? You may think that isn’t allowed since we have a method `drawSquare()` but that method doesn’t take any parameters and our new method does. Java allows you to use the same method name as another method as long as the parameter list is different. This is called method overloading.

Program 4: Draw Square With Width as a Parameter

```java
public void drawSquare3(int width) {
    // code
}
```

```java
public void drawSquareWithPassedWidth(int width) {
    // code
}
```

```java
public void drawSquare(int width) {
    // code
}
```
public void drawSquare(int width)
{
    this.turnRight();
    this.forward(width);
    this.turnRight();
    this.forward(width);
    this.turnRight();
    this.forward(width);
    this.turnRight();
    this.forward(width);
    this.turnRight();
    this.forward(width);
}
Section 3.5 Creating Methods

FIGURE 3.10: Showing the result of sending the width as a parameter to \texttt{drawSquare}

leads to general solutions that work in lots of situations.

Making it Work Tip: Use names that make sense
We called the first method \texttt{drawSquare()} and the second \texttt{drawSquare2()}. Does it matter? Absolutely not! Well, not to the computer, at any rate. The computer doesn’t care what names you use—they’re entirely for your benefit. Pick names that (a) are meaningful to you (so that you can read and understand your program), (b) are meaningful to others (so that others can read and understand it), and (c) are easy to type. long names, like, \texttt{drawARectangleWithEqualWidthAndHeight} are meaningful, easy-to-read, but are a pain to type. Does this mean that you can use ”orange” as a method name? Yes, you can, but it may be confusing even for you, and especially confusing for others. It helps to use method names that indicate what the method does.

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

When you evaluate the function, by specifying its name with input values (also
Chapter 3  Introduction to Programming

3.6 WORKING WITH MEDIA

What if we want to create and manipulate media like pictures or sounds? Just as we created the World and Turtle classes to define what we mean by these to the computer we have created Picture and Sound classes.

3.6.1 Creating a Picture Object

How would you create a picture? The syntax for creating an object is

```
new Class(parameterList)
```

Try entering the following in the interactions pane.

```
> System.out.println(new Picture());
```

Picture, filename null height 100 width 200

It looks like we create a Picture object with a height of 100 and a width of 200, but why don’t we see it? New objects of the class Picture aren’t shown automatically. You have to ask them to show themselves using the message `show()`.

So let’s ask the Picture object to show itself. Oops, we forgot to declare a variable to refer to the Picture object so we don’t have any way to access it. Let’s try it again and this time declare a variable for it. The syntax for declaring a variable is `type name = expression;`. The type is the name of the class so we will use a type of Picture. What should the name be? Well the name should describe what the object is so let’s use `picture1`.

```
> Picture picture1 = new Picture();
> picture1.show();
```

Now we can see the created picture in Figure 3.11.

Why doesn’t it have anything in it? When you create a Picture object using `new Picture()` the default width is 200 and the default height is 100 and...
the default is that all of the pixels in the picture are white. How can we create a picture from data in a file from a digital camera? We can use `new Picture(String fileName)` which takes an object of the `String` class which is the full file name of a file to read the picture information from.

What is the full file name of a file? The full or complete name of a file is the path to the file as well as the base file name and extension. How can we get the full file name for a file? One way is to use another class we have created for you. The `FileChooser` class has a class method `pickAFile()` which will display a dialog window that will help you pick a file.

```java
> System.out.println(chooser.pickAFile());
```

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

![FIGURE 3.12: The File Chooser](image)
Click on the file name to select it and then click Open, or double-click, to select a file.

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

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

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

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

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

Files that have an extension of “.jpg” or a type of “JPEG Image” are JPEG files. They contain pictures. (To be picky, they contain data that can be interpreted to be a representation of a picture – but that’s close enough to “they contain pictures.”) JPEG is a standard encoding (a representation) for any kind of image. The other kind of media files that we’ll be using frequently are “.wav” files (Figure 3.13). 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.

3.6.2 Showing a Picture

So now we know how to get a complete file name: Path and base name. This doesn’t mean that we have the file itself loaded into memory. To get the file into memory, we have to tell Java how to interpret this file. We know that JPEG files are pictures, but we have to tell Java explicitly to read the file and make a Picture object from it (an object of the Picture class).
The way we create and initialize new objects in Java is to ask the class to create a new object using `new ClassName(parameterList)`. The class contains the description of the data each object of the class needs to have so it is the thing that knows how to create objects of that class. You can think of a class as a factory for making objects of that class. So, to create a new object of the `Picture` class from a file name use `new Picture(fileName)`. The `fileName` is the name of a file as a string. We know how to get a file name using `FileChooser.pickAFile()`.

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

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

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

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

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

- We can do it all at once because the result from one method can be used in the next method: `new Picture(FileChooser.pickAFile()).show()`. That's what we see in figure 3.14. This code will first invoke the `pickAFile()` class method of the class `FileChooser` because it is inside the parentheses. The `pickAFile()` method will return the name of the selected file as a string. Next it will create a new `Picture` object with the selected file name. And finally it will ask the created `Picture` object to show itself.

![Figure 3.14](matt-spaceman.jpg)

**FIGURE 3.14:** Picking, making, and showing a picture, using the result of each method in the next method. The picture used is matt-spaceman.jpg.

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

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

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

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

Try the following in the interactions pane. Pick a file name that ends in “.jpg”.

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

As you can see we can name the file that we get from `FileChooser.pickAFile()` by using using (`String fileName =`). This says that the variable named `fileName` will be of type `String` (will refer to an object of the `String` class) and that the `String` object that it will refer to will be returned from `FileChooser.pickAFile()`. In a similar fashion we can create a variable named `pictureObj` that will refer to an object of the `Picture` class that we get from creating a new `Picture` object with the `fileName` using `Picture pictureObj = new Picture(fileName)`. We can then ask that `Picture` object to show itself by sending it the `show()` message.
using `pictureObj.show()`. That’s what we see in figure 3.15.

### Making it Work Tip: Java Conventions

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

### Debugging Tip: Methods names must be followed by parentheses!

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

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

### 3.6.4 Object references

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

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

Object variables (references) reserve space for a reference to an object of the given class. A reference allows the computer to determine the address of the actual object (it isn’t just the address of the object). If the object variable is declared but not assigned to an object the reference is set to `null` which means that it doesn’t
FIGURE 3.15: Picking, making, and showing a picture, when naming the pieces. The picture shown is katie.jpg.

refer to any object yet.

3.6.5 Playing a Sound

We can replicate this entire process with sounds.

- We still use `FileChooser.pickAFile()` to find the file we want and get its file name.
- We use `new Sound(fileName)` to make a `Sound` object using the passed `fileName` as the file to read the sound information from.
- We will use `play()` to play the sound. The method `play()` is an object method (invoked on a `Sound` object). It plays the sound one time. It doesn’t return anything.

Here are the same steps we saw previously with pictures:

```
> System.out.println(FileChooser.pickAFile());
C:\intro-prog-java\mediasources\croak.wav
> System.out.println(new Sound(FileChooser.pickAFile()));
Sound file: croak.wav length: 17616
> new Sound(FileChooser.pickAFile()).play();
```
Chapter 3    Introduction to Programming

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

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

3.6.6 Naming your Media (and other Values)

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

```
> String fileName = FileChooser.pickAFile();
> Sound soundObj = new Sound(fileName);
> soundObj.play();
```

3.6.7 Naming the Result of a Method

We can assign names to the results of methods (functions). If we name the result from `FileChooser.pickAFile()`, each time we print the name, we get the same result. We don’t have to re-run `FileChooser.pickAFile()`. Naming code in order to re-execute it is what we’re doing when we define methods (functions), which comes up in Section 3.5.

```
> String fileName = FileChooser.pickAFile();
> System.out.println(fileName);
C:\intro-prog-java\mediasources\beach-smaller.jpg
> System.out.println(fileName);
C:\intro-prog-java\mediasources\beach-smaller.jpg
```

In the below example, we declare variables (assign names) for the file name (a `String` object) and the `Picture` object.

```
> 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());
```

because we set `myFileName` to be equal to the result of `FileChooser.pickAFile()`. The values get substituted for the names when the expression is evaluated. `new Picture(myFileName)` is an expression which, at evaluation time, gets expanded into

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

because `C:\intro-prog-java\mediasources\katie.jpg` is the name of the file that was picked when `FileChooser.pickAFile()` was evaluated and the returned value was named `myFileName`.

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

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

Or even substitute for the name.

```java
> 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
```

Assuming, of course, that you picked the same file.
Computer Science Idea: We can substitute names, values, and methods. We can substitute a value, a name assigned to that value (the variable name), and the method returning that value interchangeably. The computer cares about the values, not if it comes from a string, a name (a variable), 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 method that doesn’t return anything (and so is pretty useless to `System.out.println()`), we can just ask the method to be executed by on an object by typing the variable name for the object followed by a ‘.’ and then the method name and its input (if any) in parentheses followed by a semicolon.

```java
> 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 `myFileName = FileChooser.pickAFile()`, and `aPicture.show()`. These are more than expressions: They’re telling the computer to do something.

### 3.7 CONCEPTS SUMMARY

This chapter introduced many concepts: invoking object and class methods, creating objects, and how to create new methods.

#### 3.7.1 Invoking Object Methods

You must invoke an object method on an object.

```java
objectReference.methodName(parameterList);
```

The object that the method is invoked on will be implicitly passed to the method and can be referred to using the keyword `this` inside of the method. Object methods usually work with the data in the current object.

#### 3.7.2 Invoking Class Methods

You can invoke a class method using the name of the class.

```java
ClassName.methodName(parameterList);
```

Class methods are used for general methods like `Math.abs(-3)`. Class methods do not have access to object data.

#### 3.7.3 Creating Objects

To create an object ask the class to create and initialize a new object.

```java
new ClassName(parameterList)
```
3.7.4 Creating new Methods

To create a method open the class definition file ClassName.java and put the method before the closing curly brace at the end of the file.

To define a method use:

```java
public returnType methodName(parameterList)
{
    // statement in the body of the method
}
```

If the method doesn’t return a value use the keyword “void” as the return type. Each parameter in the parameter list has a type and name. Parameters are separated by commas. Method and parameter names start with a lowercase letter, but the first letter of each additional word is capitalized.

OBJECTS AND METHODS SUMMARY

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

<table>
<thead>
<tr>
<th>Objects</th>
<th>Encoding Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pictures</td>
<td>objects of our Picture class. Pictures are encodings of images, typically coming from a JPEG file.</td>
</tr>
<tr>
<td>Sounds</td>
<td>objects of our Sound class. Sounds are encodings of sounds, typically coming from a WAV file.</td>
</tr>
<tr>
<td>Strings</td>
<td>Java String object e.g., &quot;Hello!&quot;. A sequence of characters (including spaces, punctuation, etc.) delimited on either end with a double quote character.</td>
</tr>
<tr>
<td>Turtles</td>
<td>objects of our Turtle class. Turtles can move forward, turn left, turn right, turn by a specified angle, and leave a trail.</td>
</tr>
<tr>
<td>Worlds</td>
<td>objects of our World class. Worlds can hold objects such as objects of the Turtle class.</td>
</tr>
</tbody>
</table>

Here are the methods (functions) introduced in this chapter:
<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character.getNumericValue(Character character)</td>
<td>Returns the equivalent numeric value in Unicode for the input character.</td>
</tr>
<tr>
<td>FileChooser.pickAFFile()</td>
<td>Lets the user pick a file and returns the complete path name as a string.</td>
</tr>
<tr>
<td>Math.abs(int number)</td>
<td>Takes a number and returns the absolute value of it.</td>
</tr>
<tr>
<td>picture.show()</td>
<td>Shows the Picture object that it is invoked on. No return value.</td>
</tr>
<tr>
<td>sound.play()</td>
<td>Plays the sound object (object of the Sound class) that it is invoked on.</td>
</tr>
<tr>
<td>turtle.forward(int numberOfSteps)</td>
<td>Asks the Turtle object that it is invoked on to move forward by the passed number of steps. No return value.</td>
</tr>
<tr>
<td>turtle.setPenDown(boolean value)</td>
<td>Asks the Turtle object that it is invoked on to set the pen down value to the passed value. If the pen is down and you set it to false then the pen is lifted. No return value.</td>
</tr>
<tr>
<td>turtle.hide()</td>
<td>Asks the Turtle object that it is invoked on to stop showing itself. No return value.</td>
</tr>
<tr>
<td>turtle.moveTo(int x, int y)</td>
<td>Asks the Turtle object that it is invoked on to move to the specified x and y location. No return value.</td>
</tr>
<tr>
<td>turtle.penDown()</td>
<td>Asks the Turtle object that it is invoked on to put down the pen and draw the trail of future movements. No return value.</td>
</tr>
<tr>
<td>turtle.penUp()</td>
<td>Asks the Turtle object that it is invoked on to pick up the pen so you don’t see the trail of future movements. No return value.</td>
</tr>
<tr>
<td>turtle.show()</td>
<td>Asks the Turtle object that it is invoked on to show (draw) itself. No return value.</td>
</tr>
<tr>
<td>turtle.turn(int angle)</td>
<td>Asks the Turtle object that it is invoked on to turn by the specified angle. A negative angle will turn that much to the left and a positive angle will turn that much to the right. No return value.</td>
</tr>
<tr>
<td>turtle.turnLeft()</td>
<td>Asks the Turtle object that it is invoked on to turn left 90 degrees. No return value.</td>
</tr>
<tr>
<td>turtle.turnRight()</td>
<td>Asks the Turtle object that it is invoked on to turn right 90 degrees. No return value.</td>
</tr>
</tbody>
</table>
PROBLEMS

3.1. Some computer science concept questions:
   • What is an algorithm?
   • What is a program?
   • What is Moore’s Law?
   • What is a method?
   • What creates new objects?
   • What does “pass by value” mean?

3.2. Test your understanding of Java with the following:
   • What does picture.show() do?
   • What does FileChooser.pickAFile() do?
   • What does turtle.turnLeft() do?
   • What does turtle.forward() do?
   • What does turtle.turn(-45) do?
   • What does turtle.turn(45) do?
   • What does turtle.penUp() do?
   • What does turtle.hide() do?

3.3. Which of the following are class methods and which are object methods? How do you tell which are which?
   • Math.abs(-3);
   • sound.play();
   • FileChooser.pickAFile();
   • picture.show();
   • ColorChooser.pickAColor();
   • turtle.turnLeft();

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

3.5. How many and what kind of variables (primitive or object) are created in the code below? Draw what the variables look like.
   > World world = new World();
   > Turtle turtle1 = new Turtle();
   > turtle1.forward(30);
   > Turtle turtle2 = new Turtle();
   > turtle2.turnRight();
   > turtle2.forward(30);

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

3.7. Create a World object and a Turtle object and use the Turtle object to draw a house.

3.8. Create a World object and a Turtle object and use the Turtle object to draw a star.
TO DIG DEEPER

The best (deepest, most material, most elegant) computer science textbook is *Structure and Interpretation of Computer Programs* [2], by Abelson, Sussman, and Sussman. It’s a hard book to get through, though. Somewhat easier, but in the same spirit is the new book *How to Design Programs* [9].

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* [16] which combine good computer science with creative and fun projects.
PART TWO

PICTURES

Chapter 4  Modifying Pictures using Loops
Chapter 5  Modifying Pixels in a Matrix
Chapter 6  Conditionally Modifying Pixels
Chapter 7  Drawing
CHAPTER 4

Modifying Pictures using Loops

4.1 HOW PICTURES ARE ENCODED
4.2 MANIPULATING PICTURES
4.3 CHANGING COLOR VALUES
4.4 CONCEPTS SUMMARY

Chapter Learning Objectives

The media learning goals for this chapter are:

• To understand how images are digitized by taking advantage of limits in human vision.

• To identify different models for color, including RGB, the most common one for computers.

• To manipulate color values in pictures, like increasing or decreasing red values.

• To convert a color picture to grayscale, using more than one method.

• To convert a color picture to its negative representation.

The computer science goals for this chapter are:

• To introduce arrays.

• To write object methods.

• To do iteration with *while* and *for* loops.

• To introduce comments.

• To understand the scope of a variable name.

• To introduce breaking a method into smaller methods.

4.1 HOW PICTURES ARE ENCODED

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

Pictures are two-dimensional arrays of *pixels* (which is short for picture element). In this section, each of those terms will be described.
Section 4.1  How Pictures areEncoded

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.

![Array Diagram](image)

**FIGURE 4.1: A depiction of the first five elements in an array**

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

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

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

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

In Figure 4.2, you see an example matrix. At coordinates \((0, 0)\) (horizontal, vertical), you’ll find the matrix element whose value is 15. The element at \((1, 1)\) is 7, \((2, 1)\) is 43, and \((3, 1)\) is 23. We will often refer to these coordinates as \((x, y)\) \(((\text{horizontal, vertical})\).

![FIGURE 4.2: An example matrix (two-dimensional array) of numbers](image)

Have you ever played the game battleship? If you have then you had to specify both the row and column of your guess (B-3) This means row B and column 3 (Figure 4.3). Have you ever gone to a play? Usually your ticket has a row and seat number (row E, seat 10). These are both examples of two-dimensional arrays. Picture data is also represented as a two-dimensional array.

What’s stored at each element in the picture is a pixel. The word “pixel” is short for “picture element.” It’s literally a dot, and the overall picture is made up of lots of these dots. Have you ever taken a magnifying glass to pictures in the newspaper or magazines, or to a television or even your own computer monitor? (Figure 4.4 was generated by capturing as an image the of the top left part of the DrJava window and then magnifying it 600%. It’s made up of many, many dots. When you look at the picture in the magazine or on the television, it doesn’t look like it’s broken up into millions of discrete spots, but it is.
Section 4.1 How Pictures are Encoded

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 200%, so that each individual pixel is visible (Figure 4.5).

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 an eagle. We actually have more than one kind of visual 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. This is an evolutionary advantage since it allows you to pick out the saber-tooth sneaking up on you from the side.

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

Picture encoding is actually more complex than sound encoding. A sound
modifying pictures using loops

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

4.1.1 Color Respresentations

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

Based on how we perceive color, as long as we encode what hits our three kinds of color sensors, we’re recording our human perception of color. Thus, we can encode each pixel as a triplet of numbers. The first number represents the amount of red in the pixel. The second is the amount of green, and the third is the amount of blue. We can make up any human-visible color by combining red, green, and blue light (Figure 4.6) (repeated at Figure 5.17 (page 176). Combining all three gives us pure white. Turning off all three gives us black. We call this the RGB color model.

There are other models for defining and encoding colors besides the RGB color model. There’s the HSV color model which encodes Hue, Saturation, and Value (sometimes also called the HSB color model for Hue, Saturation, and Brightness). The nice thing about the HSV model is that some notions, like making a color “lighter” or “darker” map cleanly to it, e.g., you simply change the saturation

FIGURE 4.5: Image shown in the picture explorer: 100% image on left and 500% on right (close-up of the branch over the mountain)
Section 4.1  How Pictures are Encoded  85

FIGURE 4.6: Merging red, green, and blue to make new colors

(Figure 4.7). Another model is the CMYK color model, which encodes Cyan, Magenta, Yellow, and black (“B” could be confused with Blue). The CMYK model is what printers use—those are the inks they combine to make colors. However, the four elements means more to encode on a computer, so it’s less popular for media computation. RGB is the most popular model on computers.

FIGURE 4.7: Picking colors using the HSB color model

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

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

We actually perceive borders of objects, motion, and depth through a separate vision system. We perceive color through one system, and luminance (how light/dark things are) through another system. Luminance is not actually the amount of light, but our perception of the amount of light. We can measure the amount of light (e.g., the number of photons reflected off the color) and show that a red and a blue spot each are reflecting the same amount of light, but we’ll perceive the blue as darker. Our sense of luminance is based on comparisons with the surroundings—the optical illusion in Figure 4.8 highlights how we perceive gray levels. The two end quarters are actually the same level of gray, but because the two mid quarters end in a sharp contrast of lightness and darkness, we perceive that one end is darker than the other.

**FIGURE 4.8:** The ends of this figure are the same colors of gray, but the middle two quarters contrast sharply so the left looks darker than the right.

Most tools for allowing users to pick out colors let the users specify the color as RGB components. The Macintosh offers RGB sliders in its basic color picker (Figure 4.9). The color chooser in Java offers a similar set of sliders (Figure 4.10).

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 4.11 (replicated at Figure 5.18 (page 176) in the color pages) is a representation of pixel RGB triplets in a matrix representation. Thus, the pixel at (1, 0) has color (30, 30, 255) which means that it has a red value of 30, a green value of 30, and a blue value of 255—it’s a mostly blue color, but not pure blue. Pixel at (2, 1) has pure green but also more red and blue ((150, 255, 150)), so it’s a fairly light green.

Images on disk and even in computer memory are usually stored in some kind
Section 4.1 How Pictures are Encoded

FIGURE 4.9: The Macintosh OS X RGB color picker

FIGURE 4.10: Picking a color using RGB sliders from Java

of compressed form. The amount of memory needed to represent every pixel of even small images is pretty large (Table 4.1). A fairly small image of 320 pixels across by 240 pixels wide, with 24-bits per pixel, takes up 230,400 bytes—that’s roughly 230 kilobytes (1000 bytes) or 1/4 megabyte (million bytes). A computer monitor with 1024 pixels across and 768 pixels vertically with 32-bits per pixel takes up over 3
88 Chapter 4 Modifying Pictures using Loops

![RGB triplets in a matrix representation](image)

**FIGURE 4.11:** RGB triplets in a matrix representation

**TABLE 4.1:** Number of bytes needed to store pixels at various sizes and formats

<table>
<thead>
<tr>
<th>Format</th>
<th>320x240 image</th>
<th>640x480</th>
<th>1024x768</th>
</tr>
</thead>
<tbody>
<tr>
<td>24-bit color</td>
<td>230, 400 bytes</td>
<td>921, 600</td>
<td>2, 359, 296</td>
</tr>
<tr>
<td>32-bit color</td>
<td>307, 200</td>
<td>1, 228, 800</td>
<td>3, 145, 728</td>
</tr>
</tbody>
</table>

megabytes just to represent the screen.

**Computer Science Idea: Kilobyte (kB) versus Kibibyte (Kib or K or KB)**

The term kilobyte has caused problems because it has been interpreted differently by different groups. Computer scientists used it to mean $2$ to the $10$th power which is $1024$ bytes. Telecommunications engineers have used it to mean $1000$ bytes. The International Electrotechnical Commission (IEC) decreed in 1998 to call $1,024$ bytes a kibibyte (KiB) and $1,000$ bytes a kilobyte. Similarly a mebibyte is defined to be $2$ raised to the $20$th power and a megabyte is $1,000,000$ bytes (one million bytes). A gibibyte is defined to be $2$ raised to the $30$th power and a gigabyte is defined to be $1,000,000,000$ (one billion bytes).

### 4.2 MANIPULATING PICTURES

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

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

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

```java
Picture, filename c:\intro-prog-java\mediasources\caterpillar.jpg
height 150 width 329
```

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

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

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

We can get any particular pixel from a picture using getPixel(x,y) where x and y are the coordinates of the pixel desired. The x coordinate starts at 0 at the top left of the picture and 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(). This just grabs all the pixels in the first column from top to bottom and then all the pixels in the second column from top to bottom and so on till it has all of the pixels.

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

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

```java
> System.out.println(pixelObject.getX());
0
> System.out.println(pixelObject.getY());
0
```
Chapter 4  Modifying Pictures using Loops

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

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

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

### Making it Work Tip: Importing Classes from Packages

$\text{Color}$ is a Java class in the package java.awt. A package is a group of related classes. Java uses packages to group classes that you need for a particular purpose. To use classes in packages other than java.lang (which contains System and Math) you will need to import them. Importing a class or all classes in a package allows you to use the name of a class without fully qualifying it. To fully qualify a name use the package name followed by a period (dot) and the class name. The fully qualified name for the $\text{Color}$ class is $\textit{java.awt.Color}$. 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 $\textit{import java.awt.*;}$. To import just the Color class from the package java.awt use $\textit{import java.awt.Color;}$. Importing doesn’t make your class larger, it is just used to determine what class you mean.

### Debugging Tip: Undefined Class Error

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

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

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

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

Common Bug: Seeing changes in the picture

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

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

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

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

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

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

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

```java
> pictureObject.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 methods (functions) just described. Please reset the interactions pane by clicking the Reset button at the top of DrJava before you do the following.

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

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

The result showing a small black line on the left side below the middle of the
leaf appears in Figure 4.12. The black line is 100 pixels down, and the pixels 10 through 19 from the left edge have been turned black.

![Figure 4.12](image)

**FIGURE 4.12:** Directly modifying the pixel colors via commands: Note the small black line on the left under the line across the leaf

### 4.2.1 Exploring Pictures

On your CD, you will find the *MediaTools* application with documentation for how to get it started. You can also open a picture explorer in DrJava. Both the MediaTools application and the picture explorer will let you get pixel information from a picture. You can see the picture explorer in Figure 4.13 and the MediaTools application appears in Figure 4.14. Both of these will display the x, y, red, green, and blue values for a pixel. They will also both let you zoom in or out.

![Figure 4.13](image)

**FIGURE 4.13:** Exploring the caterpillar with the line

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

The picture explorer allows you to zoom at various levels of magnification, by choosing one in the ZOOM menu. As you move your cursor around in the picture, press down with the mouse button. You’ll be shown the \((x, y)\) (horizontal, vertical) coordinates of the pixel your mouse cursor is currently over, and the red, green, and blue values at that pixel. You can use the next and previous buttons to change
Section 4.3 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, 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.

### 4.3.1 Using While Loops in Pictures

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

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

Making it Work Tip: Variable Names

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

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

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

The syntax for a while loop is:

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

Let’s talk through the pieces here.

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

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

**Computer Science Idea: Flowcharts**

Figure 4.15 shows the flowchart of a while loop. A flowchart is a visual representation of the execution of a method or function. It shows the order in which statements are executed and branches or conditional execution. Normal statements are shown in rectangles. Tests are shown in diamonds and have a true branch which is executed when the test is true and a false branch that is executed when the test is false. A flowchart can help you understand what a method is doing.

So, a typical while loop will look like the following code.
int count = 0;
while (count < target)
{
    // commands to be done inside loop
    count = count + 1;
}

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

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

### Debugging Tip: Stopping an Infinite Loop

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

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

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

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

Let’s talk through this code.

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

The loop starts at 0 and indexes through the array of pixels until the end.

The `setColor()` method of the `Pixel` class is used to change the color of the pixel from its current color to black. The index is incremented by one each time through the loop, so it stops once it has processed all of the pixels in the picture.
Chapter 4  Modifying Pictures using Loops

- We tell the Picture object to show (display) itself using `pict.show();`.

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

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

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

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

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

### Debugging Tip: Loops and Variable Declarations

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

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

#### 4.3.2 Increasing/Decreasing Red (Green, Blue)

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

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

**Program 5: Decrease the amount of red in a picture by 50%**
Section 4.3 Changing color values

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

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

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

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

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

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

Go ahead and type the above into your DrJava definitions pane before the last curly brace in the Picture.java file. Click Compile ALL to get DrJava to compile the new method. Why do we have to compile the file before we can use the new method? Computers don’t understand the Java source code directly. We must compile it which translates the class definition from something people can read and understand into something a computer can read and understand. When we successfully compile a ClassName.java file the compiler outputs a ClassName.class file which contains the instructions that a computer can understand. If our compile is not successful we will get error messages that explain what is wrong. We have
Making it Work Tip: Comments in Java
You may notice that there are some interesting characters in the reduceRed method. The ‘/**’ and ‘//’ are comments in Java. Comments are descriptions of what your code is doing. Use comments to make the code easier to read and understand (not only for yourself but also for others). There are actually three kinds of comments in Java. The ‘//’ starts a comment and tells the computer to ignore everything else till the end of the current line. You can use ‘/\*’ followed at some point by ‘\*/’ for a multi-line comment. The ‘/**’ followed at some point by ‘*/’ creates a JavaDoc comment. JavaDoc is a utility that pulls the JavaDoc comments from your class files and creates hyperlinked documentation from them. All of the Java class files written by Sun have JavaDoc comments in them and that is how the API documentation was created.

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

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

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

The original picture and its red-decreased version appear in Figure 4.16 (and at Figure 5.19 on page 177). 50% is obviously a lot of red to reduce! The picture looks like it was taken through a blue filter.
Section 4.3  Changing color values  103

FIGURE 4.16: The original picture (left) and red-decreased version (right)

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

Tracing the program: How did that work?.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

![Diagram](image)

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

Variables that you declare inside methods are not automatically initialized for you so you **should** initialize them when you declare them.
Chapter 4 Modifying Pictures using Loops

Computer Science Idea: Scope

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

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

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

Next in the body of the loop is `value = pixel.getRed();`. This sets the variable `value` to the amount of red in the current pixel. Remember that the amount of red can vary from a minimum of 0 to a maximum of 255.
Next in the body of the loop is `value = (int) (value * 0.5);`. This sets the variable `value` to the integer amount that you get from multiplying the current contents of `value` by 0.5. The `(int)` is a cast to integer so that the compiler doesn’t complain about losing precision since we are storing a floating point number in an integer number. Any numbers after the decimal point will be discarded. We do this because colors are represented as integers. The `(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 `(int)`. This is called casting and is required whenever a larger value is being placed into a smaller variable. So if the result of a multiplication has a fractional part that fractional part will just be thrown away so that the result can fit in an `int`.

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

After the statements in the body of the loop are executed the `index = index`
+ 1; will be executed which will add one to the current value of index. Since index was initialized to 0 this will result in index holding the value 1.

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

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

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

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

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

Eventually, we get Figure 4.16 (and at Figure 5.19 on page 177). 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
Chapter 4  Modifying Pictures using Loops

FIGURE 4.17: Using the picture explorer to convince ourselves that the red was decreased picture, but did we really decrease the red? By 50%?

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

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

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

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

Let’s increase the red in the picture now. If multiplying the red component by 0.5 decreased it, multiplying it by something over 1.0 should increase it. I’m going to apply the increase to the exact same picture, to see if we can reduce the blue (Figure 4.18 and Figure 5.20).

Program 6: Increase the red component by 30%

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

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

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

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

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

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

This method works much the same was as the method `decreaseRed`. We set up some variables that we will need such as the array of pixel objects, the current pixel, the current value, and the current index. We loop through all the pixels in the array of pixels and change the red value for each pixel to 1.3 times its original...
Adding one or subtracting one from a current value is something that is done frequently in programs. Programmers have to do lots of typing so they try to reduce the amount of typing that they have to do for things they do frequently. Notice the `index++;` in the increase red program. This has the same result as `index = index + 1;` and can also be written as `++index;`. You can also use `index--;` or `--index;` which will have the same result as `index = index - 1;`. Be careful of using this when you are also assigning the result to a variable. If you do `int x = index++;` x will be assigned the original value of index and then index will be incremented. If you do `int x = ++index;` first index will be incremented and then the value assigned to x.

We can even get rid of a color completely. The method below erases the blue component from a picture by setting the blue value to 0 in all pixels (Figure 4.19).
Section 4.3  Changing color values

and Figure 5.21).

Program 7: Clear the blue component from a picture

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

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

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

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

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

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

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

4.3.3 Creating a Sunset

We can certainly do more than one color manipulation at once. Mark wanted to try to generate a sunset out of a beach scene. His first attempt was to increase the red, but that doesn’t always work. Some of the red values in a given picture are pretty high. If you go past 255 for a channel value it will keep the value at 255.
His second thought was that maybe what happens in a sunset is that there is less blue and green, thus emphasizing the red, without actually increasing it. Here was the program that we wrote for that:

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

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

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

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

         // increment the index
         i++;
     }
 */
```
Section 4.3 Changing color values

FIGURE 4.20: Original beach scene (left) and at (fake) sunset (right)

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

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

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

What we see happening in Method 8 is that we’re changing both the blue and green channels—reducing each by 30%. The effect works pretty well, as seen in Figure 4.20 (and in the color section at Figure 5.22).

4.3.4 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 every method? 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 example might make that clearer. Consider the program to make a sunset (Method 8). That works by reducing the green and blue, each by 30%. What if we rewrote that method so that it called two smaller methods that just did the two pieces of the manipulation? We’d end up with something like Method 9.

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

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

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

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

        // increment the index
        i++;
    }
}

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

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

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

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

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

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

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

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

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

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. However, this could also be handled by a method that changes each color by some passed in amount. This would be a very general and reusable method.

Now, let’s say that we asked you to pick a picture and created the picture in 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 of Picture objects, to each of which you want to add a sunset look? Do you really want to pick out each of those few hundred frames? Or would you rather write a method to go through each of the frames (which we’ll learn how to do in a few chapters) and invoke makeSunset() on each Picture object. That’s why we make methods general and reusable—you never know when you’re going to want to use that method again, in a larger context.

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.

Even larger methods, like makeSunset(), do “one and only one thing.” makeSunset() makes a sunset-looking picture. It does that by decreasing green and decreasing blue. It calls two other methods to do that. What we end up with is a 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.

4.3.5 Variable Name Scope

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

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

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

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

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

```
Program 10: General change red by a passed amount

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

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

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

    // increment i
    i++;
} }

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

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

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

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

Program 11: Change all pixel colors by the passed amounts

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

       // loop through all the pixels
       while( i < pixelArray.length)
Section 4.3 Changing color values

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

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

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

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

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

We could use this method as shown below:

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

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

Recall seeing in Method 7 this code:

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

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

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

Chapter 4  Modifying Pictures using Loops

// increment index
index++;
}

We could also write that same algorithm like this:

/**
 * Method to clear the blue from the picture (set the blue to 0 for all pixels)
 */
public void clearBlue2() {
    Pixel[] pixelArray = this.getPixels();
    int i = 0;
    // loop through all the pixels
    while (i < pixelArray.length) {
        pixelArray[i].setBlue(0);
        i++;
    }
}

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

4.3.6 Using a For Loop

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

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

The syntax for a for loop is:

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

Let's talk through the pieces here.

• First comes the required Java keyword for.

• Next we have a required opening parenthesis
Section 4.3  Changing color values  123

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

- Next comes the required semicolon.

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

- Next comes the required semicolon.

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

- Next is the required closing parenthesis.

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

![Flowchart of a for loop](image-url)

**FIGURE 4.21:** Flowchart of a for loop
Chapter 4: Modifying Pictures using Loops

Compare the flowchart (Figure 4.21) for a for loop with the flowchart for a while loop (Figure 4.15). They look the same because for loops and while loops execute in the same way even though the code looks different. Any code can be written using either. The syntax of the for loop just makes it easier to remember to declare a variable for use in the loop and to change it each time through the loop since all of that is written at the same time that you write the test. To change clearBlue() to use a for loop simply move the declaration and initialization of the index variable i to be done in the initialization area and the increment of i to be done in the change area.

Program 12: Another clear blue method

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

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

4.3.7 Lightening and Darkening

To lighten or darken a picture is pretty simple. It’s the same pattern as we saw previously, but instead of changing a color component, you change the overall color. Here’s lightening and then darkening as methods. Figure 4.22 (Figure 5.23) shows the lighter and darker versions of the original picture seen earlier.

Program 13: Lighten the picture

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

    // loop through all the pixels
    for (int i = 0; i < pixelArray.length; i++)
```
{  
    // get the current pixel  
    pixel = pixelArray[i];  
    
    // get the current color  
    color = pixel.getColor();  
    
    // get a lighter color  
    color = color.brighter();  
    
    // set the pixel color to the lighter color  
    pixel.setColor(color);  
}

Program 14: Darken the picture

/**
   * Method to darken the color in the picture
   */
   
   public void darken()
   {
      Pixel[] pixelArray = this.getPixels();  
      Color color = null;  
      Pixel pixel = null;  
      
      // loop through all the pixels  
      for (int i = 0; i < pixelArray.length; i++)  
      {
         // get the current pixel  
         pixel = pixelArray[i];  
         
         // get the current color  
         color = pixel.getColor();  
         
         // get a darker color  
         color = color.darker();  
         
         // set the pixel color to the darker color  
         pixel.setColor(color);  
      }
   }

4.3.8 Creating a Negative

Creating a negative image of a picture is much easier than you might think at first. Let’s think it through. What we want is the opposite of each of the current values
for red, green, and blue. It’s easiest to understand at the extremes. If we have a red component of 0, we want 255 instead. If we have 255, we want the negative to have a zero.

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

Here’s the program that does it, and you can see even from the grayscale image that it really does work (Figure 4.23 and Figure 5.24).

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

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

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

Figure 4.22: Original picture, lightened picture, and darkened picture
Section 4.3 Changing color values

blueValue = pixel.getBlue();

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

FIGURE 4.23: Negative of the image

4.3.9 Converting to Grayscale

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

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

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

\[
\text{intensity} = \frac{(red + green + blue)}{3}
\]

This leads us to the following simple program and Figure 4.24 (and Figure 5.25 on page 179).

Program 16: Convert to grayscale

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

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

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

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

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

Program 17: Convert to grayscale with more careful control of luminance

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

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

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

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

        // set the pixel color to the new color
        pixel.setColor(new Color(luminance, luminance, luminance));
    }
}
You can get an element of the array using `arrayReference[index]`. Where the index values can range from 0 to `arrayReference.length-1`.

```java
Pixel = pixelArray[i];
```

### 4.4.2 Loops

Loops are used to execute a block of statements while a boolean expression is true. Most loops have variables that change during the loop which eventually cause the boolean expression to be false and the loop to stop. Loops that never stop are called infinite loops.

We introduced two types of loops in this chapter: `while` and `for`. The `while` loop is usually used when you don’t know how many times a loop needs to execute and the `for` loop is usually used when you do know how many times the loop will execute. We introduced the `while` loop first because it is easier for beginners to understand.

The `while` loop has the keyword `while` followed by a boolean expression and then a block of statements between an open and close curly brace. If the boolean expression is true the body of the loop will be executed. If the boolean expression is false execution will continue after the body of the loop (after the close curly brace). If you just want to execute one statement in the body of the loop then you don’t need the open and close curly braces, but you should indent the statement.

```
while (boolean expression)
{
    statement1;
    statement2;
    ...
}
```

If you use a `while` loop to execute a block of statements a set number of times you will need to declare a variable before the `while` and that variable will need to be changed in the body of the loop. You may also need to declare other variables that you use in the loop before the `while`. Don’t declare variables inside the loop because you will use more memory that way.

```java
int index = 0;

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

    // do something to the pixel

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

The `for` loop does the same thing as a `while` loop but it lets you declare the variables that you need for the loop, specify the boolean expression to test, and
specify how to change the loop variables all in one place. This means you are less likely to forget to do each of these things.

```java
// loop through all the pixels
for (int index = 0; index < pixelArray.length; index++)
{
    // get the current pixel
    pixel = pixelArray[index];
    // do something to the pixel
}
```

### 4.4.3 Comments

Comments are text that the programmer adds to the code to explain the code. The compiler ignores the comments when it translates the code into a form that the computer understands.

There are several types of comments in Java. To tell the compiler to ignore all text till the end of the current line use `//`.

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

To tell the compiler to ignore several lines use a starting `/*` and ending `*/`.

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

To put special comments in that can be parsed out by the `javadoc` utility to make html documentation use a starting `/**` followed by an ending `*/`.

```java
/**
 * Method to change the red by an amount
 * @param amount the amount to change the red by
 */
```

### OBJECTS AND METHODS SUMMARY

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

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

**Picture methods**
### Chapter 4 Modifying Pictures using Loops

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

#### Pixel methods

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

#### Color methods

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

#### ColorChooser methods
Section 4.4 Concepts Summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ColorChooser.pickAColor()</td>
<td>Displays a window with ways to pick a color. Find the color you want, and the function will return the Color object that you picked.</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.pink`, `Color.yellow`, `Color.orange`, `Color.lightGray`, `Color.magenta`, `Color.cyan`. Notice that these are not method calls but are class variables (fields) so they can be accessed using `ClassName.fieldName`.

PROBLEMS

4.1. Some questions about picture concepts:

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

4.2. Program 5 (page 100) 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.

4.3. Write the blue and green versions of Program 5 (page 100).

4.4. Each of the below is equivalent to Program 6 (page 111). Test them and convince yourself that they are equivalent. Which do you prefer and why?

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

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

/**
 * Method to increase the amount of red by 1.3
 */
public void increaseRed3()
```
```java
{  
    Pixel[] pixelArray = 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 < pixelArray.length; i++)
    {
        // get the current pixel
        pixel = pixelArray[i];

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

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

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

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

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

4.7. Write new methods like Program 7 (page 113) to clear red and green. For each of these, which would be the most useful in actual practice? How about combinations of these?

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

4.9. Write a new method to maximize blue (i.e., setting it to 255) instead of clearing it use Program 7 (page 113) as a starting point. Is this useful? Would the red or green versions be useful?

4.10. There is more than one way to compute the right grayscale value for a color value. The simple method that we use in Program 16 (page 127) may not be what your grayscale printer uses when printing a color picture. Compare the color (relatively unconverted by the printer) grayscale image using our simple algorithm in Figure 5.25 with the original color picture that the printer has converted to grayscale (left of Figure 4.16). How do the two pictures differ?

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

A wonderful new book on how vision works, and how artists have learned to manipulate it, is *Vision and art: The biology of Seeing* by Margaret Livingstone [19].
Chapter Learning Objectives
The media learning goals for this chapter are:

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

The computer science goals for this chapter are:

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

5.1 COPYING PIXELS
We can only get so far in our image processing with getPixels() before we need to know where a pixel is. For example, if we want to copy just part of a picture to another picture we will need to know the x and y values to start with and end at.
5.1.1 Looping Across the Pixels with a Nested Loop

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

Your loops will look something like this:

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

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

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

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

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

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

Does it matter which way you process the pixels? Not if all you are trying to do is process all the pixels. Both of these loops will process all the pixels in a picture.
For example, here’s Program 13 (page 124), but using explicit pixel references.

**Program 18: Lighten the picture using nested loops**

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

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

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

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

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

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

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

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

3. The code `for (int x = 0; x < getWidth(); x++)` declares a variable `x` of
Section 5.1 Copying Pixels

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

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

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

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

7. Next comes \( \text{color = color.brighter()} \). 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. The code \( \text{pixel.setColor}(\text{color}); \) sets the current pixel’s color to be the lighter color.

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

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

5.1.2 Mirroring a Picture

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

First, let’s think through what we’re going to do. We’ll pick a horizontal mirrorPoint—halfway across the picture, \((\text{int}) (\text{picture.getWidth}() / 2)\). (We want this to be an integer, a whole number, so we’ll cast it using \((\text{int})\).) We’ll have the \( x \) value increment from 1 to the mirrorPoint. At each value of \( x \), we want to copy the color at the pixel \( x \) pixels to the \text{left} of the mirrorPoint to the pixel \( x \) pixels to the \text{right} of the mirrorPoint. The left would be mirrorPoint-x and the right would be mirrorPoint+x. Take a look at Figure 5.1 to convince yourself that we’ll actually reach every pixel using this scheme. Here’s the actual program.
Program 19: Mirror pixels in a picture along a vertical line

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

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

We’d use it like this, and the result appears in Figure 5.2.

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

Another way to code this would be to copy the colors for the pixels starting
Section 5.1 Copying Pixels

FIGURE 5.2: Original picture (left) and mirrored along the vertical axis (right)

with the left-most x (x=0) into the right-most pixel (width - 1). To do this have x range from 0 to less than the mirrorPoint and copy it to (width - 1 - x).

Can we mirror horizontally? Sure!

Program 20: Mirror pixels horizontally, top-to-bottom

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

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

We'd use it like this, and the result appears in Figure 5.3.

```java
> String fileName = "C:/intro-prog-java/mediasources/redMotorcycle.jpg";
> System.out.println(fileName);
C:/intro-prog-java/mediasources/redMotorcycle.jpg
> Picture picture = new Picture(fileName);
> picture.show();
> picture.mirrorHorizontal();
```
Chapter 5  Modifying Pixels in a Matrix

> picture.repaint();

Now this last method copies from the top of the picture onto the bottom (see Figure 5.3). You can see that we’re getting the color from \texttt{topPixel} which is from \texttt{mirrorPoint - y} that will always be \textit{above} \texttt{mirrorPoint} since smaller values of \texttt{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 5.3).

\begin{verbatim}
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());
        }
    }
}
\end{verbatim}

We’d use it like this, and the result appears in Figure 5.3.

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

\textbf{Mirroring Usefully.}

While mirroring is probably mostly used for interesting effects, occasionally it has some more serious (but still fun!) purposes. Mark took a picture of the Temple of Hephaistos which is in the ancient agora in Athens, Greece, when traveling to a conference (Figure 5.4). By sheer luck, Mark got the pediment dead horizontal.
Section 5.1  Copying Pixels  143

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

This time we don’t want to mirror one-half of the picture onto the other half. We just want to mirror the pixels from the good side of the pediment on the left onto the bad side on the right. We also don’t want to mirror all the pixels in the y direction. We just need the pixels from the top of the pediment to the bottom of the pediment. We can use the explorer to find out the value for those pixels (Figure 5.5). The pediment starts at x=13 and the middle is at x=276. The highest part of the pediment is at y=27 and it ends at y=97.

How do we mirror just a small part of a picture? Well, we still need a point to mirror around. We will use 276 for that instead of half the width like we did in the method mirrorVertical(). Let’s start by copying the mirrorVertical() method and changing the name to mirrorPartVertical() and setting the mirror point to the value 276. We also don’t want to stop when x gets all the way to the mirror point. We just want to copy from x=13 to x=276. We can start x at 1 and copy while x is less than the number of pixels to copy (276-13=263). We can start y at the top of the pediment 27 (instead of 0) and copy while it is less than 97 (instead of the
height of the picture).

### Program 22: Mirror Part of a Picture Vertically

```java
/**
 * Method to mirror part of the picture around a vertical line
 * at a mirror point
 */
public void mirrorPartVertical()
{
    int mirrorPoint = 276;
    Pixel leftPixel = null;
    Pixel rightPixel = null;

    // loop through the rows
    for (int y = 27; y < 97; y++)
    {
        // loop from 1 to just before the mirror point
        for (int x = 1; x < 263; x++)
        {
            leftPixel = getPixel((mirrorPoint - x), y);
            rightPixel = getPixel((mirrorPoint + x), y);
            rightPixel.setColor(leftPixel.getColor());
        }
    }
}
```

To use this method try this.

```java
> String fileName = "C:/intro-prog-java/mediasources/temple.jpg";
```
Section 5.1  Copying Pixels 145

> Picture picture = new Picture(fileName);
> picture.explore();
> picture.mirrorPartVertical();
> picture.explore();

The final picture is in (Figure 5.6)–it worked pretty well! Of course, it is possible to tell that it was digitally manipulated. For example, if you check the shadows, you can see that the sun must have been on the left and the right at the same time.

![The manipulated temple](image)

**FIGURE 5.6:** The manipulated temple

You may be tired of typing the full path name to each file. You can save the name of the directory that has your media in it and then use `FileChooser.getMediaPath(fileName)` to get the full path name. The method `FileChooser.getMediaPath(fileName)` generates a complete path for you by returning a string with the saved directory name followed by the base file name. The default media directory is “c:/intro-prog-java/mediasources/”. If you wish to use a different media directory you should execute `FileChooser.setMediaPath(directory)` first! `FileChooser.setMediaPath(directory)` lets you specify the place (directory) where you store your media.

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

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

```java
/**
 * Method to mirror part of the picture around a vertical line
 * at a mirror point
 */
public void mirrorPartVertical()
```
Chapter 5  Modifying Pixels in a Matrix

```java
{  int  mirrorPoint = 276;
 Pixel  leftPixel = null;
 Pixel  rightPixel = null;
 int  count = 0;

 // loop through the rows
 for (int y = 27; y < 97; y++)
 {  // loop from 1 to just before the mirror point
    for (int x = 1; x < 263; x++)
    {  System . out . println ( " copying color from " +
        mirrorPoint - x );
        leftPixel = getPixel((mirrorPoint - x), y);
        System . out . println ( " to " + mirrorPoint + x );
        rightPixel = getPixel((mirrorPoint + x), y);
        rightPixel . setColor ( leftPixel . getColor ( ) );
        count = count + 1;
    }
 }  System . out . println ( "We copied " + count + " pixels" );
}
```

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 we 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 part of the picture around a vertical line
 *  at a mirror point
 */
 public  void  mirrorPartVertical()
 {  int  mirrorPoint = 276;
```
Section 5.2  Copying and Transforming Pictures

Pixel leftPixel = null;
Pixel rightPixel = null;
int count = 0;

// loop through the rows
for (int y = 27; y < 97; y++)
{
    // loop from 1 to just before the mirror point
    for (int x = 1; x < 263; x++)
    {
        leftPixel = getPixel((mirrorPoint - x), y);
        rightPixel = getPixel((mirrorPoint + x), y);
        rightPixel.setColor(leftPixel.getColor());
        count = count + 1;
    }
}
System.out.println("We copied " + count + " pixels");

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

5.2 COPYING AND TRANSFORMING PICTURES

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

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

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

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

Our target will be the paper-sized JPEG file in the mediasources directory,
Chapter 5 Modifying Pixels in a Matrix

which is 7x9.5 inches, which will fit on a 9x11.5 inch lettersize piece of paper with one inch margins.

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

5.2.1 Copying

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

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

```
Program 23: Copying a picture to the current picture

/**
 * Method to copy the picture of Katie to the upper left corner of the current picture
 */
public void copyKatie()
{
    String sourceFile =
        FileChooser.getMediaPath("KatieFancy.jpg");
    Picture sourcePicture = new Picture(sourceFile);
    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 = this.getPixel(targetX, targetY);
            targetPixel.setColor(sourcePixel.getColor());
        }
    }
```
To use this method create a picture from the file that has a blank paper-sized picture. The picture of Katie will be copied to the top left corner of the blank picture Figure 5.7.

```java
> Picture targetPicture = new Picture(Chooser.getMediaPath("7inx95in.jpg"));
> targetPicture.show();
> targetPicture.copyKatie();
> targetPicture.show();
```

FIGURE 5.7: Copying a picture to a canvas

This method copies a picture of Katie to the canvas (blank picture) (Figure 5.7). Here’s how it works:

- The first two lines are just setting up the source (`sourcePicture`).
- Then we have the declaration of variables to keep track of the target and source pixels.
- Next comes the loop for managing the x index variables, `sourceX` for the source picture and `targetX` for the target (current) picture. The `for` loop declares both variables and initializes them to 0. You can have more than one variable declared and initialized in the initialization area of a `for` loop, just separate them with commas. Next the continuation test checks if the `sourceX` is less than the width of the source picture. Finally in the change area we increment both the `sourceX` and `targetX` variables each time after
the statements in the body of the loop have been executed. You can change more than one variable in the change area as long as you separate the changes with commas. The for loop for looping through the columns is:

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

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

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

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

![Image of a girl in a white dress]

**FIGURE 5.8:** Copying a picture midway into a canvas

Of course, we don’t have to copy from (0, 0) in the source to (0, 0) in the target. We can easily copy to another location in the target picture. All we have to do is to change where the target X and Y coordinates start. The rest stays exactly the same (Figure 5.8).

[Program 24: Copy elsewhere into the current picture]
Section 5.2  Copying and Transforming Pictures

/**
 * Method to copy the picture of Katie to (100,100) in the current picture
 */
public void copyKatieMidway()
{
    String sourceFile =
        FileChooser.getMediaPath("KatieFancy.jpg");
    Picture sourcePicture = new Picture(sourceFile);
    Pixel sourcePixel = null;
    Pixel targetPixel = null;

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

To try this method create the target picture from the blank paper-sized picture file, invoke the method on it, and show the result. The picture of Katie will be copied with the upper left corner at (100,100).

> Picture targetPicture = new Picture(Chooser.getMediaPath("7inx95in.jpg"));
> targetPicture.copyKatieMidway();
> targetPicture.show();

Similarly, we don’t have to copy a whole picture. Cropping is taking only part of a picture out of the whole picture. Digitally, that’s just a matter of changing your start and end coordinates. To grab just Katie’s face out of the picture, we only have to figure out the upper left corner of a rectangle enclosing her face and use that as the starting values for sourceX and sourceY. We also need to determine the bottom right corner of the rectangle enclosing her face and use that as the stopping x and y values. (Figure 5.9). We can use the picture explorer to determine these values. The upper left corner of the rectangle enclosing the face is at (70,3) and the bottom right corner is at (135,80).
Chapter 5 Modifying Pixels in a Matrix

Program 25: Cropping a picture onto a canvas

```java
/**
 * Method to copy just Katie’s face to the current picture
 */
public void copyKatie’sFace()
{
    String sourceFile =
       FileChooser.getMediaPath("KatieFancy.jpg");
    Picture sourcePicture = new Picture(sourceFile);
    Pixel sourcePixel = null;
    Pixel targetPixel = null;

    // loop through the columns
    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 = this.getPixel(targetX, targetY);
            // Copy the pixel
            targetPixel = sourcePixel;
        }
    }
}```
Section 5.2  Copying and Transforming Pictures  153

```java
    targetPixel.setColor(sourcePixel.getColor());
    }
}
```

To try this method create the target picture from the blank paper-sized picture file, invoke the method on it, and show the result. Just Katie’s face will be copied to the target picture with the upper left corner at (100, 100).

```java
> Picture targetPicture = new Picture(FileChooser.getMediaPath("7inx95in.jpg"));
> targetPicture.copyKatieFace();
> targetPicture.show();
```

Try to copy part of another picture to the blank paper-sized picture file. What do you need to change and what can stay the same?

**How Does That Work?**

Let’s look at a small example to see what’s going on in the copying program. We start out with a source and a target, and copy from x=0, y=0 to x=3, y=1.

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

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

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

In the mediasources folder are a couple images of flowers (Figure 5.10), each 100 pixels wide. Let’s make a collage of them, by combining several of our effects to create different flowers. We’ll copy them all into the blank image 640x480.jpg. All we really have to do is to copy the pixel colors to the right places.

**Figure 5.10: Flowers in the mediasources folder**

```java
/**
 * Method to copy flower pictures to create a collage.
 * All the flower pictures will be lined up near the
 * bottom of the current picture (5 pixels from the bottom)
 */
public void copyFlowers() {
    // create the flower pictures
    Picture flower1Picture = new Picture(FileChooser.getMediaPath("flower1.jpg"));
    Picture flower2Picture = new Picture(FileChooser.getMediaPath("flower2.jpg"));

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

    // save the heights of the two pictures
    int flower1Height = flower1Picture.getHeight();
```
int flower2Height = flower2Picture.getHeight();

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

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

// copy the flower1 negated to x = 200
flower1Picture.negate();
for (int sourceX = 0, targetX = 200;
     sourceX < flower1Picture.getWidth();
     sourceX++, targetX++)
{
    for (int sourceY = 0,
         targetY = this.getHeight() - flower1Height - 5;
         sourceY < flower1Picture.getHeight();
         sourceY++, targetY++)
    {
        sourcePixel = flower1Picture.getPixel(sourceX, sourceY);
        targetPixel = this.getPixel(targetX, targetY);
        targetPixel.setColor(sourcePixel.getColor());
    }
}
Chapter 5  Modifying Pixels in a Matrix

```java
} // clear the blue in flower 2 picture and add at x=300
flower2Picture.clearBlue();
for (int sourceX = 0, targetX = 300;
    sourceX < flower2Picture.getWidth();
    sourceX++, targetX++)
{ for (int sourceY = 0,
    targetY = this.getHeight() - flower2Height - 5;
    sourceY < flower2Picture.getHeight();
    sourceY++, targetY++)
{ sourcePixel = flower2Picture.getPixel(sourceX, sourceY);
targetPixel = this.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 = this.getHeight() - flower1Height - 5;
    sourceY < flower1Picture.getHeight();
    sourceY++, targetY++)
{ sourcePixel = flower1Picture.getPixel(sourceX, sourceY);
targetPixel = this.getPixel(targetX, targetY);
targetPixel.setColor(sourcePixel.getColor());
}
}
```
**Method that will copy all of the passed source picture into the current picture object starting with the left corner given by xStart. It will put the sourcePicture at 5 pixels from the bottom of this picture**

```java
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());
        }
    }
}
```
Chapter 5  Modifying Pixels in a Matrix

```java
public void copyFlowersBetter() {
    // create the three pictures
    Picture flower1Picture = new Picture(FileChooser.getMediaPath("flower1.jpg"));
    Picture flower2Picture = new Picture(FileChooser.getMediaPath("flower2.jpg"));

    // copy the first flower picture to near the bottom left corner of the canvas
    this.copyPictureTo(flower1Picture, 0);

    // copy the flower2 picture starting with x = 100 in the canvas
    this.copyPictureTo(flower2Picture, 100);

    // copy the flower1 negated to x = 200 in the canvas
    flower1Picture.negate();
    this.copyPictureTo(flower1Picture, 200);

    // clear the blue in flower 2 picture and add at x=300 in the canvas
    flower2Picture.clearBlue();
    this.copyPictureTo(flower2Picture, 300);

    // copy the negated flower 1 to x=400
    this.copyPictureTo(flower1Picture, 400);
}
```

The method `copyFlowersBetter` is much easier to read and understand now. And, we now have a method `copyPictureTo` which is easy to reuse.

We can even make a more general copy method which takes both the starting x and starting y values for the target picture and copies the passed source picture into the current picture with the source pictures upper left corner at the passed starting x and y values in the target.

```java
/**
 * Method that will copy all of the passed source picture into the current picture object starting with the left corner given by xStart, yStart
 * @param sourcePicture the picture object to copy
 * @param xStart the x position to start the copy into on the
*/
```
Section 5.2 Copying and Transforming Pictures

```java
public void copyPictureTo(Picture sourcePicture, int xStart, int yStart)
{
    Pixel sourcePixel = null;
    Pixel targetPixel = null;

    // loop through the columns
    for (int sourceX = 0, targetX = xStart;
        sourceX < sourcePicture.getWidth();
        sourceX++, targetX++)
    {
        // loop through the rows
        for (int sourceY = 0, targetY = yStart;
            sourceY < sourcePicture.getHeight();
            sourceY++, targetY++)
        {
            sourcePixel = sourcePicture.getPixel(sourceX, sourceY);
            targetPixel = this.getPixel(targetX, targetY);
            targetPixel.setColor(sourcePixel.getColor());
        }
    }
}
```

Notice that you can have two methods with the same names (like `copyPictureTo`) and you don’t have any trouble when you compile. How can that be? Java allows you to have many methods with the same method name as long as the parameters are different. The first `copyPictureTo` method took a `Picture` object and an int. The second `copyPictureTo` method took a `Picture` object, and two int values. So the two methods have a different number of parameters. Having more than one method with the same name but different parameters is called overloading. It doesn’t really matter what you name the parameters. What matters in the types. Two methods with the same name are allowed if the number of parameters is different, or the types of the parameters are different, or the order of the parameter types is different.

5.2.3 Blending Pictures

When we create collages by copying, any overlap typically means that one picture shows over another. The last picture painted on is the one that appears. But it doesn’t have to be that way. We can blend pictures by multiplying their colors and adding them. This gives us the effect of transparency.

We know that 100% of something is the whole thing. 50% of one and 50% of another would also add up to 100%. In the program below, we blend a picture of
the two sisters with an overlap of some 50 (the width of Katie minus 150) columns of pixels (Figure 5.12) onto the current picture.

Program 27: Blending two pictures

```java
/**
 * Method to blend two sisters together onto the current picture
 */
public void blendPictures()
{

    // create the sister pictures
    Picture katiePicture =
        new Picture(FileChooser.getMediaPath("KatieFancy.jpg"));
    Picture jennyPicture =
        new Picture(FileChooser.getMediaPath("JenParty.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 = this.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++)
    {
```
Section 5.2 Copying and Transforming Pictures

for (int sourceY=0, targetY=0; 
    sourceY < katiePicture.getHeight(); 
    sourceY++, targetY++)
{
    katiePixel = katiePicture.getPixel(sourceX, sourceY);
    jennyPixel = jennyPicture.getPixel(sourceX - 150, sourceY);
    targetPixel = this.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++)
    {
        jennyPixel = jennyPicture.getPixel(sourceX, sourceY);
        targetPixel = this.getPixel(targetX, targetY);
        targetPixel.setColor(jennyPixel.getColor());
    }
}

To try this out create a picture object using the blank 640 by 480 file and invoke the method on that. Show the result.

> Picture picture = new Picture(FileChooser.getMediaPath("640x480.jpg"));
> picture.blendPictures();
> picture.show();
5.2.4 Rotation

Transformations to the image occur by using the index variables differently or incrementing them differently, but otherwise keeping the same program. Let’s rotate Katie 90 degrees to the left. What does that mean? Let’s try it with something simple first. You can write some numbers in a table on a piece of paper and then rotate it left and then read the new table to see where the old numbers were moved to (Figure 5.13). Notice that the columns become the rows and the rows the columns but it isn’t as simple as just using the source x for the target y and the source y for the target x.

Value (0,0) in the source moves to (0,2) in the target. Value (0,1) in the source moves to (1,2) in the target. Value (1,0) in the source moves to (0,1) in the target. Value (1,1) in the source moves to (1,1) in the target. Value (2,0) in the
source moves to (0,0) in the target. Value (2,1) in the source moves to (1,0) in the target. So the first column values move into the bottom row and the last column values move into the top row. Also notice that the target x value is the same as the source y value.

We will do the rotation by looping through the pixels in the usual way and getting the source pixel in the usual way but the target pixel’s x value will be the source y and the target pixel’s y value will be width of the source picture - 1 - the source x (Figure 5.14).

![Figure 5.13: Rotating some numbers in a table to the left 90 degrees](image)

Program 28: Rotating a picture left 90 degrees

```java
/**
 * Method to copy the picture of Katie but rotate her left 90 degrees on the current picture
 */
public void copyKatieLeftRotation() {
    String sourceFile = FileChooser.getMediaPath("KatieFancy.jpg");
    Picture sourcePicture = new Picture(sourceFile);
    Pixel sourcePixel = null;
    Pixel targetPixel = null;

    // loop through the columns
    for (int sourceX = 0; sourceX < sourcePicture.getWidth(); sourceX++) {
        // loop through the rows
        for (int sourceY = 0; sourceY < sourcePicture.getHeight(); sourceY++) {
            // set the target pixel color to the source pixel color
```
Chapter 5  Modifying Pixels in a Matrix

1. `sourcePixel = sourcePicture.getPixel(sourceX, sourceY);`
2. `targetPixel = this.getPixel(sourceY, sourcePicture.getWidth() - 1 - sourceX);`
3. `targetPixel.setColor(sourcePixel.getColor());`

To try this out create a picture from the blank paper-sized file and then invoke the method on it. Show the result.

> Picture picture = new Picture(Chooser.getMediaPath("7inx95in.jpg"));
> picture.copyKatieLeftRotation();
> picture.show();

![FIGURE 5.14: Copying a picture to a blank page rotated to the left 90 degrees](image)

**How Does That Work?.**

Rotating starts with the same source and target, and even the same variable values, but since we use the target X and Y differently, we get a different effect.

Now, as we increment the Y variables, we’re moving down the source, but across the target from left to right. As we increment the X variables we’re moving across the source but up the target.
The source x and source y are both 0. The target x is equal to the source y so it is also 0. But, the target y is equal to the width of the source picture minus 1 minus the source x. The width of the source picture is 3 so the target y is 3 - 1 - 0 which is 2. So we copy the color of the source pixel at (0,0) to the target pixel at (0,2).

The source y is incremented by the inner loop to 1 and tested against the height of the source picture (2). Since it is less than the height we do the body of the inner loop. So now the source x is 0 and the source y is 1. The target x is equal to the source y so it is 1. The target y is equal to the width of the source picture minus 1 minus the source y. The width of the source picture is 3 so the target y is 3 - 1 - 0 which is 2. So we copy the color of the source pixel at (0,1) to the target pixel at (1,2).
The source y is incremented by the inner loop to 2 and tested against the height of the source picture (2). Since it is not less than the height the inner loop finishes and the source x is incremented to 1 by the outer loop. The inner loop starts and sets the source y to 0. So, the source x is 1 and the source y is 0. The target x is equal to the source y so it is 0. The target y is equal to the width of the source picture minus 1 minus the source y. The width of the source picture is 3 so the target y is 3 - 1 - 1 which is 1. So we copy the color of the source pixel at (1,0) to the target pixel at (0,1).

The inner loop will increment source y and so the next color will be copied from (1,1) to (1,1). Then, the inner loop will stop again and source x will be incremented by 1 to 2. The next color will be copied from (2,0) to (0,0). The inner loop will increment source y so the next color will be copied from (2,1) to (1,0). At this point source x will be incremented to 3 which is not less than the width of the source picture (3) and the nested loop will stop.

### 5.2.5 Scaling

A very common transformation for pictures is to scale them. Scaling up means to make them larger, and scaling them down makes them smaller. It’s common to scale a 1-megapixel or 3-megapixel picture down to a smaller size to make it easier to use on the Web. Smaller pictures require less disk space, and thus less network bandwidth, and thus are easier and faster to download.

Scaling a picture requires the use of **sampling** which we’ll also use with sounds later. To scale a picture **smaller** we are going to take every other pixel when copying from the source to the target. To scale a picture **larger** we are going to take every pixel twice.

Scaling the picture down is the easier function. We will use the passion-Flower.jpg picture which is 640 (width) by 480 (height). Instead of incrementing the source X and Y variables by 1, we simply increment by 2. We divide the amount of space by 2, since we’ll fill half as much room–our width will be 640/2 and the height will be 480/2. The result is a smaller flower on the blank 640 by 480 picture (Figure 5.15).

#### Program 29: Scaling a picture down (smaller)

```java
// **
* Method to copy the flower but smaller (half as big)
* on to the current picture
*/
public void copyFlowerSmaller() {
    Picture flowerPicture =
        new Picture(FileChooser.getMediaPath("passionFlower.jpg"));
    Pixel sourcePixel = null;
    Pixel targetPixel = null;

    // loop through the columns
```
Section 5.2  Copying and Transforming Pictures  167

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 = this.getPixel(targetX, targetY);  
        targetPixel.setColor(sourcePixel.getColor());  
    }  
}  

To try this out create a picture object using the blank 640 by 480 file and invoke the method on that. Show the result.

> Picture picture = new Picture(FileChooser.getMediaPath("640x480.jpg"));  
> picture.copyFlowerSmaller();  
> picture.show();

![Figure 5.15: Scaling the picture down](image)

Scaling up the picture (making it larger) is a little trickier. We want to take every pixel twice. What we’re going to do is to increment the source index variables by 0.5. Now, we can’t reference pixel 1.5. But if we reference (int) 1.5 we’ll get 1 again, and that’ll work. The sequence of 1, 1.5, 2, 2.5... will become 1, 1.2, 2... The result is a larger form of the picture (Figure 5.16). Let’s try this on rose.jpg which is 320 by 240 so scaling it up will result in a picture that is 640 by 480.

Program 30: Scaling the picture up (larger)
/**
 * Method to copy a flower but scaled to 2x normal size
 * onto the current picture
 */

public void copyFlowerLarger()
{
    Picture flowerPicture =
        new Picture(FileChooser.getMediaPath("rose.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 = this.getPixel((int) targetX, (int) targetY);
            targetPixel.setColor(sourcePixel.getColor());
        }
    }
}

STo try this out create a picture object using the blank 640 by 480 file and invoke the method on that. Show the result.

> Picture picture = new Picture(FileChooser.getMediaPath("640x480.jpg"));
> picture.copyFlowerLarger();
> picture.show();

How Did That Work?

We start from the same place as the original code for copying a picture. Say we are copying from the source picture starting at (0,0) and copying to the target picture starting at (3,1). First we will copy the color of the pixel at (0,0) in the source picture to (3,1) in the target picture.
Section 5.2  Copying and Transforming Pictures  169

FIGURE 5.16: Scaling up a picture

When we increment \texttt{sourceY} by 0.5, the actual value will be 0.5 but the (int) value is 0 so we end up referring to the same pixel in the source, but the target has moved on to the next pixel. So we will copy the color of the pixel at (0,0) to (3,2).

When we increment \texttt{sourceY} a second time by 0.5 it will now equal 1.0, so we now move on to the next pixel in the source. So we will copy the color of the pixel at (0,1) to (3,3).

Again when the \texttt{sourceY} is incremented by 0.5 the actual value will be 1.5 but the (int) of that is 1 so we will copy from (0,1) to (3,4).

And eventually, we cover every pixel. Notice that the end result is degraded–
it’s choppier than the original. Each pixel is copied four times: twice in the x direction and twice in the y direction.

You might want to be able to scale a picture to a particular size, instead of always using the canvas pictures. There is a constructor that takes a width and height `new Picture(width, height)` and creates a blank picture of the passed width and height (both specified in pixels). `new Picture(640, 480)` would create a picture object that is 640 pixels wide by 480 pixels tall—just like the canvas.

Here is a more general method that will scale up the current picture object by some passed number of times. It uses the constructor that specifies a width and height for the new `Picture` object.

Since this method creates a picture object that it changes we want to be able to return this object so that we can refer to it again. How do we return something from a method? Well we need to say what type of thing we are returning and then we have to actually return it. You specify the type of thing that the method returns in the method declaration. Up to now we have been using the keyword `void` to say that the method doesn’t return anything. So, change the `void` to be the type of thing that you are returning. Since this method will return an object of the `Picture` class the type is `Picture`. At the end of the method use the keyword `return` followed by what you want to return. The compiler will check that the type of the thing you actually return matches the type you used in the method declaration. If it doesn’t you will get a compile error.

```
/*
 * Method to create a new picture that is scaled up by the passed number of times.
 * @return the new scaled up picture
 */
public Picture scaleUp(int numTimes) {
    Picture targetPicture =
        new Picture(this.getWidth() * numTimes,
                    this.getHeight() * numTimes);
    double factor = 1.0 / numTimes;
    Pixel sourcePixel = null;
    Pixel targetPixel = null;

    // loop through the columns
```
for (double sourceX = 0, targetX = 0;
    sourceX < this.getWidth();
    sourceX = sourceX + factor, targetX++)
{
    // loop through the rows
    for (double sourceY = 0, targetY = 0;
        sourceY < this.getHeight();
        sourceY = sourceY + factor, targetY++)
    {
        sourcePixel = this.getPixel((int) sourceX, (int) sourceY);
        targetPixel = targetPicture.getPixel((int) targetX,
            (int) targetY);
        targetPixel.setColor(sourcePixel.getColor());
    }
}
return targetPicture;

Since the method scaleUp returns the resulting scaled Picture object we had
better save a reference to the Picture object to be able to refer to it again.

> Picture p = new Picture(FileChooser.getMediaPath("flower1.jpg");
> p = p.scaleUp(2);
> p.explore();

Since this method create a new Picture object and copies the scaled picture
into that new Picture object and then returns the new Picture object if you want
to see the result you will have to save a reference to the resulting picture. You can
reuse variables like p, but realize that you will no longer have a reference to the
original Picture object. Of course, you could have declared a new variable to hold
the scaled picture. It would also be of type Picture.

> Picture origPicture = new Picture(FileChooser.getMediaPath("flower1.jpg");
> Picture scaledPicture = origPicture.scaleUp(2);
> scaledPicture.show();
> origPicture.show();

5.3 CONCEPTS SUMMARY
This chapter introduced two-dimensional arrays, nested loops, working with mul-
tiple variables in a for loop, returning a value from a method, and method over-
loading.

5.3.1 Two-dimensional Arrays
Pixels are stored in a two-dimensional array. A two-dimensional array is similar
to seating in an auditorium. You can find your seat based on the row and chair
number. You can access a location in a two-dimensional array by specifying an x
and y. All indices start with 0.
5.3.2 Nested Loops

To process all of the pixels in a picture and track the x and y location of each pixel you need to use a nested loop. Nested loops are loops inside of loops. You can either loop through the rows and then the columns (y and then x) or loop through the columns and then the rows (x and then y).

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

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

To restrict the area that you are looping through use different values for starting and stopping the loop. To loop through a rectangular area starting with the pixel at (startX, startY) at the upper left corner of the rectangular area and ending with the pixel at (endX,endY) as the bottom right corner of the rectangular area use:

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

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

You can declare more than one variable in a loop. This is useful when you copy from one picture to another. Use variables to represent the source picture x and y values and use other variables to represent the target picture x and y values. You can change how the source and target pixel are used in order to rotate the picture.

```java
// 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++)
    {
        sourcePixel = sourcePicture.getPixel(sourceX, sourceY);
        targetPixel = this.getPixel(targetX, targetY);
        targetPixel.setColor(sourcePixel.getColor());
    }
}

By changing the initial and ending values of sourceX, sourceY, targetX and
targetY you can change what part of the source picture you want to copy and
where you want it to go on the target picture. Using this you can clip and create
collages.

If you change the amount you increment or decrement a loop variable by you
can scale a picture up or down.

5.3.3 Returning a Value from a Method

To declare a method you specify the visibility for the method, the type of thing it
returns, the name of the method, and the parameter list inside parentheses. This
is followed by the body of the method which is inside of an open and close curly
brace.

visibility returnType name( parameterList )
{
    // statements in method

    // return a value
    return valueToReturn;
}

Methods that do not return any value use the keyword void as the returnType.
Methods that do return a value use the type of that value for the returnType and
then have a return keyword in them that is followed by the thing to return. Re-
member that a type is any of the primitive types or the name of a class.

Here is an example public method declaration that doesn’t return anything
and the name of the method is mirrorVertical and it doesn’t take any parameters.

public void mirrorVertical()

Here is an example public method declaration that returns an object of the
class Picture.

public Picture scaleUp (int numTimes)

Notice that it gives a return type of Picture. The body of the method must
have the keyword return in it and it must return an object that is an instance of
the class Picture.
5.3.4 Method Overloading

A class can have more than one method with the same name as long as the parameter list is different. The methods can take a different number of parameters, or the types of the parameters can be different, or the order of the types can be different. You can't have two methods with the same name and the same number of parameters with the same types in the same order.

```java
public void copyPictureTo (Picture sourcePicture, int xStart)

public void copyPictureTo (Picture sourcePicture, int xStart, int yStart)
```

Notice that there are two method declarations with the same name but one takes 2 parameters and one takes 3. The compiler will check that a method exists that takes the same number and type of parameters. If the compiler can't find a method with the same number, type, and order of parameters it will report that the method doesn't exist.

```java
> p.copyPictureTo();
Error: No 'copyPictureTo' method in 'Picture'
```

**OBJECTS AND METHODS SUMMARY**

Here are the functions used or introduced in this chapter:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>new Picture(int width, int height)</td>
<td>Creates a new <code>Picture</code> object with the given width and height. All pixels are white.</td>
</tr>
<tr>
<td>getMediaPath(String fileName)</td>
<td>Returns the full path name with the media directory followed by the passed file name. This is a class method on the <code>FileChooser</code> class. The default media directory is &quot;c:/intro-prog-java/mediasources/&quot;.</td>
</tr>
<tr>
<td>setMediaPath(String directory)</td>
<td>Sets the media directory to use when getting a full path using <code>getMediaPath(String fileName)</code>. This is a class method on the <code>FileChooser</code> class.</td>
</tr>
</tbody>
</table>

**PROBLEMS**

5.1. Write the code to mirror a picture around a horizontal line from (0, height-1) to (width-1, height-1). Be aware that this will double the height of the picture.

5.2. Try to mirror a picture around a diagonal line from (0, 0) to (width-1, height-1). Try to mirror a picture around a diagonal line from (0, height-1) to (width-1, 0).

5.3. Write the code to rotate a picture to the right by 90 degrees.

5.4. We've seen that if you increment the source picture index by 2 while incrementing the target picture index by 1 for each copied pixel, you end up with the source being scaled down onto the target. What happens if you increment the target picture index by 2 as well? What happens if you increment both the source and target by 0.5 and use `int` to get just the integer part?

5.5. Write a method named `createCollage` to create a collage of the same image at least four times onto the `7x96in.jpg` blank JPEG. (You are welcome to add additional images, too.) One of those four copies can be the original picture. The other three should be modified forms. You can do any of scaling, cropping,
or rotating the image; creating a negative of the image; shifting or altering colors on the image; and making it darker or lighter.

After composing your image, mirror it. You can do it vertically or horizontally (or otherwise), in any direction—just make sure that your four base images are visible still after mirroring.

Your single method should make all of this happen— all of the effects and compositing must occur from the single function `createCollage`. Of course, it is perfectly okay to use other functions, but make it so that a tester of your program need only to call `setMediaPath()` and put all your input pictures in her `mediasources` directory, create a `Picture` object from the blank paper-sized file, and then execute `createCollage()`—and will expect to have a collage generated and returned.

*5.6. 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

5.4 COLOR FIGURES

FIGURE 5.17: Merging red, green, and blue to make new colors

FIGURE 5.18: Color: RGB triplets in a matrix representation
Section 5.4  Color Figures  177

FIGURE 5.19: Color: The original picture (left) and red-decreased version (right)

FIGURE 5.20: Color: Overly blue (left) and red increased by 30% (right)

FIGURE 5.21: Color: Original (left) and blue erased (right)
Chapter 5  Modifying Pixels in a Matrix

FIGURE 5.22: Original beach scene (left) and at (fake) sunset (right)

FIGURE 5.23: Color: Lightening and darkening the original picture

FIGURE 5.24: Color: Negative of the image
FIGURE 5.25: Color: Color picture converted to grayscale

FIGURE 5.26: Color: Increasing reds in the browns
FIGURE 5.27: Color: Increasing reds in the browns, within a certain range

FIGURE 5.28: Finding the range where Jenny’s eyes are red, then changing them to black
Section 5.4  Color Figures  181

FIGURE 5.29: Frames from the slow sunset movie

FIGURE 5.30: Frames from the slow fade-out movie
FIGURE 5.31: Frames from the Mommy watching Katie movie

FIGURE 5.32: Frames from the original too dark movie
FIGURE 5.33: Frames from the modified lighter movie

FIGURE 5.34: Frames from the original movie with kids crawling in front of a blue screen
FIGURE 5.35: Frames from the kids on the moon movie
CHAPTER 6

Conditionally Modifying Pixels

6.1 CONDITIONAL PIXEL CHANGES
6.2 SIMPLE EDGE DETECTION: CONDITIONALS WITH TWO OPTIONS
6.3 SEPIA-TONED AND POSTERIZED PICTURES: USING MULTIPLE CONDITIONALS TO CHOOSE THE COLOR
6.4 HIGHLIGHTING EXTREMES
6.5 COMBINING PIXELS: BLURING
6.6 BACKGROUND SUBTRACTION
6.7 CHROMAKEY
6.8 CONCEPTS SUMMARY

Chapter Learning Objectives

The media learning goals for this chapter are:

- To replace one color with another in a picture.
- To do simple edge detection.
- To replace several colors in a picture: sepia-toned.
- To replace a range of colors with one color: posterizing.
- To average nearby pixels when scaling up for a smoother result: blur.
- To replace the background in a picture.

The computer science goals for this chapter are:

- To conditionally execute a statement or block of statements using if.
- To use a conditional with two possible results: if and else.
- To use a conditional with greater than two possible results: if, else if, and else.
- To combine boolean expressions with and and or.
6.1 CONDITIONAL PIXEL CHANGES

So far we have been processing all of the pixels in the same way. But, what if we want to process the pixels in different ways? For example, we might want to turn someone's hair a different color, or get rid of 'red-eye' in a picture, or even reduce the number of colors in a picture.

We need something that executes a block of code only if some condition (expression) is true. We know that computers can compare values to see if they are equal, less than, or greater than (and combinations of these) and return true or false. We have used this with loops by continuing a loop while the index is less than the length of the array as in: while(index < pixelArray.length). This will execute the loop while the expression (index < pixelArray.length) is true.

Try the following in the interactions pane:

```java
> 0 < 20
true
> 30 < 20
false
> 20 < 20
false
> 20 <= 20
true
```

One way to conditionally execute code in Java is with an if (expression). The if is a keyword that means that if the expression in the parentheses evaluates to true then execute the following statement or block of statements. If it is false just skip the execution of that statement or block of statements (Figure 6.1). So, we can check if the color at the current pixel is close to a particular color and if so execute a statement or block of statements (in curly braces).

```
if (expression)
```

Try the following in the interactions pane:

```java
> int x = 30;
> if (x < 40) System.out.println("x is less than 40");
```
x is less than 40
> if (x > 40) System.out.println("x is greater than 40");
> System.out.println(x);
30

Notice that since x is less than 40 the string saying so was output. However, since x is not greater than 40 the string saying x is greater than 40 was not output. We do see the output from the next statement System.out.println(x) since execution jumps to the statement following an if when the expression is false.

6.1.1 Comparing Colors

What does it mean to compare two colors? How can the computer tell if the color at the current pixel is “red”? The distance between two colors is the Cartesian distance between the colors as points in a three-dimensional space, where red, green, and blue are the three dimensions. Recall that the distance between two points \((x_1, y_1)\) and \((x_2, y_2)\) is:
\[
\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}
\]
The similar measure for two colors \((\text{red}_1, \text{green}_1, \text{blue}_1)\) and \((\text{red}_2, \text{green}_2, \text{blue}_2)\) is:
\[
\sqrt{(\text{red}_1 - \text{red}_2)^2 + (\text{green}_1 - \text{green}_2)^2 + (\text{blue}_1 - \text{blue}_2)^2}
\]
However, you won’t have to code this. The \texttt{Pixel} class has an object method \texttt{colorDistance(Color color)} which returns the distance between the color in the current \texttt{Pixel} object and the passed color. The hard part is determining what “close enough” is for two colors.

6.1.2 Replacing Colors

Here’s a program that tries to replace the brown color with red. Mark used the picture explorer to figure out roughly what the RGB values were for Katie’s brown hair, then wrote a program to look for colors close to that, and then increase the redness of those pixels. Mark played a lot with the value that he used for distance (here, 50.0) and the amount of redness increase (here, 100% increase). However, this approach turned part of the couch and carpet red too. (Figure 6.2 and Figure 5.26).

Program 32: Color replacement: Turn brown into red

```java
/**
 * Method to turn to the brown in a picture into red
 */
public void turnBrownIntoRed()
{
    Color brown = new Color(42, 25, 15);
    Pixel[] pixels = this.getPixels();
    Pixel pixel = null;

    // loop through the pixels
```
Conditionally Modifying Pixels

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

    // check if in distance to brown and if so double the red
    if (pixel.colorDistance(brown) < 50.0)
    {
        pixel.setColor(new Color((int)(pixel.getRed() * 2.0),
                                   pixel.getGreen(),
                                   pixel.getBlue()));
    }
}
```

To use this method to turn Katie into a redhead first create a `Picture` object from the file “KatieFancy.jpg”. Then invoke the method `turnBrownIntoRed` on that `Picture` object and show the result.

```java
> Picture picture = new Picture(FileChooser.getMediaPath("KatieFancy.jpg"));
> picture.turnBrownIntoRed();
> picture.explore();
```

![Image](Image.png)

**FIGURE 6.2:** Increasing reds in the browns

Notice that we can use a simple `for` loop through the one-dimensional array of `pixels` for this. We don’t care where the pixels are in the two-dimensional array in this method. Of course, we could have used nested `for` loops instead to loop through all the pixels.
Section 6.1 Conditional Pixel Changes

What this method is doing is looping through all the pixels in the current picture and for each pixel checking if the distance between the color in the current pixel is less than 50 away from the color brown (defined as red=42, green=25, blue=15). If the distance between the current color and the defined brown is less than 50, the red value at the current pixel is doubled. If the distance is equal or greater than 50, the pixel color is not changed.

With the picture explorer we can also figure out the coordinates just around Katie’s face, and then just do the browns near her face. The effect isn’t too good, though it’s clear that it worked. The line of redness is too sharp and rectangular (Figure 6.3 and Figure 5.27).

Program 33: Color replacement in a rectangular area

```java
/**
 * Method to turn brown to red inside of
 * a rectangular area
 */
public void turnBrownToRedInRectangle()
{
    Color brown = new Color(42, 25, 15);
    Pixel pixel = null;

    // loop through the x values
    for (int x=63; x < 125; x++)
    {
        for (int y=6; y < 76; y++)
        {
            // get the current pixel
            pixel = this.getPixel(x, y);

            // check if in distance to brown and if so double the red
            if (pixel.colorDistance(brown) < 50.0)
            {
                pixel.setColor(new Color((int) (pixel.getRed() * 2.0),
                                           pixel.getGreen(),
                                           pixel.getBlue()));
            }
        }
    }
}
```

To use this method to turn Katie’s hair red first create a Picture object from the file “KatieFancy.jpg”. Then invoke the method turnBrownIntoRedInRectangle on that Picture object and show the result.

```java
> Picture picture = new Picture(FileChooser.getMediaPath("KatieFancy.jpg"));
> picture.turnBrownIntoRedInRectangle();
> picture.explore();
```
Chapter 6  Conditionally Modifying Pixels

FIGURE 6.3: On left the couch color changes, on right the couch color doesn’t change

We put the values for the range right inside the method `turnBrownIntoRedInRectangle()`. This meant that we didn’t need to pass any parameters to specify the range but it makes the method less reusable. If we want to use it to change a different picture we would probably have to edit the method to change the range and then recompile. The method would be easier to reuse if we specified the range when we invoke the method.

Program 34: Color replacement with passing in the range

```java
/** *
 * Method to turn brown to red in a rectangular area
 * specified
 * by startX, endX−1, startY, endY−1
 * @param startX the starting location to check in x
 * @param endX the last pixel checked is one less than this in x
 * @param startY the starting location to check in y
 * @param endY the last pixel checked is one less than this in y
 */
public void turnBrownToRedInRectangle(int startX, int endX, int startY, int endY, double distance)
{
```
Section 6.1 Conditional Pixel Changes

Color brown = new Color(42,25,15);
Pixel pixel = null;

// loop through the x values
for (int x=startX; x < endX; x++)
{
    for (int y=startY; y < endY; y++)
    {
        // get the current pixel
        pixel = this.getPixel(x,y);

        /* check if in distance to brown is less than
         * the passed distance and if so double the red
         */
        if (pixel.colorDistance(brown) < distance)
            pixel.setColor(new Color((int)(pixel.getRed() * 2.0),
                                      pixel.getGreen(),
                                      pixel.getBlue()));
    }
}

That certainly is easier to reuse now for other ranges. But, what if we want
to change the distance to use between brown and the current color? We could pull
out the distance and pass that in as well.

Program 35: Color replacement with passing in the range and distance

/**
 * Method to turn brown to red in a rectangular area
 * specified
 * by startX, endX-1, startY, endY-1
 * @param startX the starting location to check in x
 * @param endX the last pixel checked is one less than
 * this in x
 * @param startY the starting location to check in y
 * @param endY the last pixel checked is one less than
 * this in y
 * @param distance if the current color is within
 * this distance to brown then change it
 */
public void turnBrownToRedInRectangle(int startX, int endX,
                                      int startY, int endY,
                                      double distance)
{
    Color brown = new Color(42,25,15);
    Pixel pixel = null;
// loop through the x values
for (int x=startX ; x < endX ; x++)
{
    for (int y=startY ; y < endY ; y++)
    {
        // get the current pixel
        pixel = this . getPixel(x,y);
        
        /* check if in distance to brown is less than
         * the passed distance and if so double the red
         */
        if (pixel . colorDistance(brown) < distance)
        {
            pixel . setColor(new Color((int)(pixel . getRed() * 2.0),
                                         pixel . getGreen(),
                                         pixel . getBlue()));
        }
    }
}

Can you think of any other things that you could do to make this method easier to reuse? What if I want to change something other than brown? What if I want to change the old color by increasing the green?

6.1.3 Reducing Red-Eye

“Red-eye” is the effect where the flash from the camera bounces off the back of the subject’s eyes. Reducing red-eye is a really simple matter. We find the pixels that are “pretty close” (a distance from red of 165 works well) to red, then change those pixels’ color to a replacement color.

We probably don’t want to change the whole picture. In the Figure 6.4, we can see that Jenny is wearing a red dress—we don’t want to wipe out that red, too. We’ll fix that by only changing the area where Jenny’s eyes are. Using the picture explorer, we find the upper left and lower right corners of her eyes. Those points were (109,91) and (202,107).

Program 36: Remove red-eye

/**
 * Method to remove red-eye from the current picture object
 * in the rectangle defined by startX, startY, endX, endY.
 * The red will be replaced with the passed newColor
 * @param startX the top left corner x value of a rectangle
 * @param startY the top left corner y value of a rectangle
 * @param endX the bottom right corner x value of a rectangle
 * @param endY the bottom right corner y value of a rectangle
 */
Section 6.1  Conditional Pixel Changes  193

FIGURE 6.4: Finding the range of where Jenny’s eyes are red

* @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);
        }
    }
}

We call this function with:

> Picture jennyPicture = new Picture("c:/intro-prog-java/mediasources/jenny-red.jpg");
> jennyPicture.removeRedEye(109,91,202,107,java.awt.Color.black);
> jennyPicture.explore();

to replace the red with black—certainly other colors could be used for the
replacement color. The result was good, and we can check that the eye really does now have all-black pixels (Figure 6.5). (See also Figure 5.28.)

FIGURE 6.5: After fixing red-eye.

6.2 SIMPLE EDGE DETECTION: CONDITIONALS WITH TWO OPTIONS

What if we want to look for areas of high contrast between a pixel and the pixel below it? If the contrast is high we can make the pixel black and if the difference is low we can make the pixel white. This is a simple form of edge detection. It results in a picture that looks like a pencil sketch.

What does high contrast mean? It means that the difference between the two colors is high. One way to calculate this is to average the red, green, and blue values in the top pixel and subtract this from the average of the red, green, and blue values in the bottom pixel. If the absolute value of the difference is greater than some amount, then there is high contrast. If there is high contrast set the top pixel color to black and if not set it to white.

6.2.1 Negation

You have seen a way to execute a statement or block of statements if some condition is true using an `if`. But, how do you execute a statement or block of statements if a condition is false? One way is to use an `if` but negate the condition using the `!` operator.

```
> !true
false
> !false
true
```

So if we wanted to check if a value was less than 20 and if it is print out "Yes,
the value is less than 20" and if it isn’t print out "No, this value is not less than 20" we could test for the value being less than 20 and also test for the value being not less than 20 using the '! ' operator.

```java
> int x = 30;
> if (x < 20) System.out.println("Yes, the value is less than 20");
> if ( ! (x < 20)) System.out.println("No, this value is not less than 20");
```

No, this value is not less than 20

### 6.2.2 Testing for Both Conditions
Another way to test that a way isn’t less than 20 is to test if it is greater or equal to 20.

```java
> int x = 30;
> if (x < 20) System.out.println("Yes, the value is less than 20");
> if (x >= 20) System.out.println("No, this value is not less than 20");
```

No, this value is not less than 20

The problem with this approach is that we have to run both tests every time. But, if the first test is true there is no reason to run the second test (x can’t be both less than 20 and greater or equal to 20). We would like a way to say if the first test is true execute one statement or block of statements and if it is false then execute a different statement or block of statements.

### 6.2.3 Conditionals with Two Options
The way to do this is with `if (expression)` and `else`. As before, the `if` part will execute the statement or block of statements following the `if` when the expression is true. But, if we add a `else` this will also execute the statement or block of statements following the `else` when the expression following the `if` is false (Figure 6.8). And it will also mean that the `else` is skipped if the `if` test was true.

![Flowchart of an if with an else](image)

**FIGURE 6.6:** Flowchart of an if with an else

To enter the following in DrJava’s interaction pane use **Shift-Enter** after the if statement to let DrJava know that there will be a following `else`.
> int x = 30;
> if (x < 20) System.out.println("Yes, the value is less than 20");
else System.out.println("No, this value is not less than 20"); No, this value is not less than 20

As you can see the statement following the if didn’t execute, but the statement following the else did. Try the same thing but this time give x a value less than 20.

> x = 15;
> if (x < 20) System.out.println("Yes, the value is less than 20");
else System.out.println("No, this value is not less than 20");
Yes, the value is less than 20

Notice that this time the statement following the if did execute but the statement following the else did not.

6.2.4 Simple Edge Detection

To do simple edge detection we will loop through the pixels with the outer loop incrementing y from 0 to less than picture height-1. Usually we have been looping while y is less than the height, so why stop at height-1? We are comparing the value at a y position with the pixel below it so the last y value that still has another row below it is at height-1.

The inner loop will increment x from 0 to less than the picture width. The top pixel will be set to the pixel at the current x and y location. The bottom pixel will be set to the pixel at the current x but y+1. We will compare the absolute value of the difference between the intensities (average of color values) and if it is less than some passed limit we will change the top pixel color to white and otherwise we will set the top pixel color to black. The Pixel class has an object method getAverage() that returns the average of the three color values.

```
Program 37: Edge Detection

/**
 * Method to do a simple edge detection by comparing the
 * absolute value of the difference between the color
 * intensities (average of the color values) between a
 * pixel and the pixel below it. If the absolute value
 * of the difference between the color intensities is
 * less than a passed amount the top pixel color
 * will be set to white. Otherwise it is set to black.
 * @param amount if the absolute value of the differences
 * in the color average is less than this
 * set the color to white, else black
 */
public void edgeDetection(double amount) {
    Pixel topPixel = null;
```
Section 6.2 Simple Edge Detection: conditionals with two options

```
Pixel bottomPixel = null;
double topAverage = 0.0;
double bottomAverage = 0.0;
int endY = this.getHeight() - 1;

/* loop through y values from 0 to height – 1
   *(since compare to below pixel) */
for (int y = 0; y < endY; y++) {
    // loop through the x values from 0 to width
    for (int x = 0; x < this.getWidth(); x++) {
        // get the top and bottom pixels
        topPixel = this.getPixel(x, y);
        bottomPixel = this.getPixel(x, y + 1);

        // get the color averages for the two pixels
        topAverage = topPixel.getAverage();
        bottomAverage = bottomPixel.getAverage();

        /* check if the absolute value of the difference
           * is less than the amount */
        if (Math.abs(topAverage - bottomAverage) < amount) {
            topPixel.setColor(Color.WHITE);
        } else {
            topPixel.setColor(Color.BLACK);
        }
    }
}
```

To execute this method use:

```java
> Picture p = new Picture(FileChooser.getMediaPath("butterfly1.jpg"));
> p.explore();
> p.edgeDetection(10);
> p.explore();
```
Making it Work Tip: Using Curly Braces
You may have noticed that the method `edgeDetection` shows the starting curly braces at the end of the line instead of on a new line as shown in the previous methods. Java doesn’t care if the curly braces are at the end of a line or on a new line. Some programmers prefer one to another. It is often easier to see that you forgot a curly brace if it is by itself on a new line. However, the Java guidelines say to put the opening curly brace at the end of a line and the closing one on a new line.
You may also notice that we are using curly braces after the `if` and `else` even though there is only one statement to be executed, so they aren’t really needed. It is good practice to have them even if they aren’t needed because the code is easier to read and change.

6.3 SEPIA-TONED AND POSTERIZED PICTURES: USING MULTIPLE CONDITIONALS TO CHOOSE THE COLOR

We handled the case of having two different ways to process the pixels using an `if` and `else`. What if we have more than two ways that we want to process some pixels? For example, what if we wanted to do one thing if a value is less than some number, another thing if it is equal and yet a third if it is greater than the number?
Section 6.3  Sepia-Toned and Posterized Pictures: Using multiple conditionals to choose the color

We could check for each of these conditions with an if as shown below:

```java
> int y = 10;
> if (y < 10) System.out.println("y is less than 10");
> if (y == 10) System.out.println("y is equal to 10");
> if (y > 10) System.out.println("y is greater than 10");
```

This works but results in some unnecessary checking. Notice that y was equal to 10 and so that was printed out but it still executed the next statement which checked if y was greater than 10. But, can y be equal to 10 and greater than 10? What would have happened if y was less than 10? It would have printed out a string saying that y is less than 10 and then still checked if y was equal or greater than 10. We need something to say if the previous test was true only execute that and then skip to the end of all the checks. We have seen a way to do this for two possibilities (true or false) using if and else. One way to handle more three or more possibilities is with if, else if and finally else. You can use as many else if statements as needed. You are not required to have a final else.

![Flowchart of an if, else if, and an else](image)

```java
> int y = 2;
> if (y < 10) System.out.println("Y is less than 10");
> else if (y == 10) System.out.println("y is equal to 10");
> else System.out.println("y is greater than 10");
```
y is less than 10

So far, we’ve done color modification by simply saying “This color replaces that color.” We can be more sophisticated in our color swapping. We can look for a range of colors, by using if, else if, and else, and replace the color with some function of the original color or a specific color. The results are quite interesting.

For example, we might want to generate sepia-toned prints. Older prints sometimes have a 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, middle ranges, and shadows—and treating them differently, we can get a better effect (Figure 6.9).

FIGURE 6.9: Original scene (left) and using our sepia-tone program

The way we do this is to first convert the picture to gray, both because older prints were in shades of gray, and because it makes it a little easier to work with. We then look for high, middle, and low ranges of color, and change them separately. We want to make the shadows (darkest grays) a bit darker. We want to make most of the picture (middle grays) into a brownish color. We want to the highlights (lightest grays) a bit yellow. Recall that yellow is a mixture of red and green so one way to make things yellow is to increase the red and green. Another way is to reduce the amount of blue. The advantage to reducing the blue is that you don’t have to worry about increasing a value past 255 which is the maximum.

Program 38: Convert a picture to sepia-tones

```c
/**
 * 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
 */
```
Section 6.3  Sepia-Toned and Posterized Pictures: Using multiple conditionals to choose the color

```java
/*
 * public void sepiaTint()
 * {
 *     Pixel pixel = null;
 *     double redValue = 0;
 *     double greenValue = 0;
 *     double blueValue = 0;
 *
 *     // first change the current picture to grayscale
 *     this.grayscale();
 *
 *     // loop through the pixels
 *     for (int x = 0; x < this.getWidth(); x++)
 *     {
 *         for (int y = 0; y < this.getHeight(); y++)
 *         {
 *             // get the current pixel and color values
 *             pixel = this.getPixel(x,y);
 *             redValue = pixel.getRed();
 *             greenValue = pixel.getGreen();
 *             blueValue = pixel.getBlue();
 *
 *             // tint the shadows darker
 *             if (redValue < 60)
 *             {
 *                 redValue = redValue * 0.9;
 *                 greenValue = greenValue * 0.9;
 *                 blueValue = blueValue * 0.9;
 *             }
 *
 *             // tint the midtones a light brown
 *             // by reducing the blue
 *             else if (redValue < 190)
 *             {
 *                 blueValue = blueValue * 0.8;
 *             }
 *
 *             // tint the highlights a light yellow
 *             // by reducing the blue
 *             else
 *             {
 *                 blueValue = blueValue * 0.9;
 *             }
 *
 *             // set the colors
 *             pixel.setRed((int) redValue);
 *             pixel.setGreen((int) greenValue);
 *             pixel.setBlue((int) blueValue);
 *         }
 *     }
 * }
 */
```
Try this method out by:

```java
> Picture picture = new Picture(Picture.getMediaPath("gorge.jpg"));
> picture.show();
> picture.sepiaTint();
> picture.repaint();
```

Posterizing is a process of converting a picture to a smaller number of colors. We’re going to do that by looking for specific ranges of color, then setting the color to one value in that range. The result is that we reduce the number of colors in the picture (Figure 6.10).

![Figure 6.10: Reducing the colors (right) from the original (left)](image)

**Program 39: Posterizing a picture**

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

What’s really going on here, though, is setting up (a) a bunch of levels then (b) setting the value of red, green, or blue to the midpoint of that level. We can do
this more generally using mathematics to compute the ranges for a desired number of levels and picking the midpoint. We need to check if the current value is in the range and if so set it to the midpoint of the range.

How do we check if a value is in a range? If we call the bottom of the range $\text{bottomValue}$ and the top of the range $\text{topValue}$ then we could use this math notation $\text{bottomValue} \leq \text{testValue} < \text{topValue}$. However in Java we need to write it $\text{bottomValue} \leq \text{testValue} \text{ && testValue} < \text{topValue}$. The two ampersands (‘&&’) mean ‘and’. If I say you have to set the table and sweep the floor, how many jobs do you have to do? The answer is two, or both of them. If I say you can set the table or sweep the floor, how many jobs do you have to do then? The answer is one, or just one of the two. Similarly if in Java you have if (expression $\text{&& expression}$) then both expressions must be true for the body of the if to be executed. And, if you have if (expression $\text{|| expression}$) then only one of the two expressions must be true for the body of the if to be executed. The $\text{||}$ means ‘or’.

Below is the program for a flexible number of levels, and Figure 6.11 shows a couple of examples.

**Program 40: Posterize by levels**

```java
// Method to posterize (reduce the number of colors) in the picture
// @param numLevels the number of color levels to use
*/
public void posterize(int numLevels) {
    Pixel pixel = null;
    int redValue = 0;
    int greenValue = 0;
    int blueValue = 0;
    int increment = (int) (256.0 / numLevels);
    int bottomValue, topValue, middleValue = 0;

    // loop through the pixels
    for (int x = 0; x < this.getWidth(); x++) {
        for (int y = 0; y < this.getHeight(); y++) {
            // get the current pixel and colors
            pixel = this.getPixel(x, y);
            redValue = pixel.getRed();
            greenValue = pixel.getGreen();
            blueValue = pixel.getBlue();

            // loop through the number of levels
            for (int i = 0; i < numLevels; i++) {
                // compute the bottom, top, and middle values
```
Section 6.3 Sepia-Toned and Posterized Pictures: Using multiple conditionals to choose the color

```java
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);
```

FIGURE 6.11: Pictures posterized to two levels (left) and four levels (right)
Chapter 6 Conditionally Modifying Pixels

Computer Science Idea: Overloading
Notice that we now have two methods called posterize. One method doesn’t take any parameters and the other takes the number of levels. In Java you are allowed to have more than one method with the same name as long as the parameters are different. This is called overloading. You can have a different number of parameters as in this case. You can also use different types and/or orders for the parameters. It means that you can have several methods that have the same behavior but operate on different parameters. It certainly is easier than coming up with new names for methods that do that same thing.

6.4 HIGHLIGHTING EXTREMES

What if we want to highlight the lightest and darkest areas of a picture? Would we highlight areas that are less than some amount from white and less than the same amount from black? Is there any color that is both close to white and black? No, we would want to replace the color at all pixels that have a distance from white or a distance from black less than some amount. We used ‘&&’ to mean ‘and’ in the last program. In this program we will use ‘∥’ to mean ‘or’.

Program 41: Highlight extremes

```java
/**
   * Method to replace the pixel colors in the current picture object that have a color distance less than the passed amount to white or black with the passed replacement color
   * @param replacementColor the new color to use
   */
  public void highlightLightAndDark(double amount, Color replacementColor) {
      Pixel pixel = null;

      // loop through all the pixels in the x direction
      for (int x = 0; x < getWidth(); x++) {
          // loop through all the pixels in the y direction
          for (int y = 0; y < getHeight(); y++) {
              // get the current pixel
              pixel = getPixel(x, y);

              // if the distance from white or black is less than the
```


You can use this as follows:

```java
> Picture picture = new Picture(Picture.getMediaPath("butterfly1.jpg"));
> picture.explore();
> picture.highlightLightAndDark(50.0, java.awt.Color.yellow);
> picture.explore();
```

![FIGURE 6.12: Original picture (left) and light or dark areas highlighted (right)](image)

### 6.5 COMBINING PIXELS: BLURRING

When we make pictures larger (scaling them up), we usually get rough edges: Sharp steps to lines, which we call **pixelation**. We can reduce pixelation by **blurring** the image. What we do is set each pixel to an average of pixels around it. In this example, we go through all pixels (note the large loop that surrounds everything) and then in the X and Y dimensions, compute the average of the pixels to either side of the pixel. It takes a picture, and a number (**size**) of pixels to compute the average.

Of course we need to be careful not to try and access pixels beyond the allowed values of the two-dimensional array of pixels. Try this in the interactions pane:
Chapter 6  Conditionally Modifying Pixels

```java
> Picture p = new Picture(FileChooser.getMediaPath("caterpillar.jpg"));
> System.out.println(p.getWidth());
329
> System.out.println(p.getHeight());
150
> p.getPixel(330,160);
java.lang.ArrayIndexOutOfBoundsException: Coordinate out of bounds!
at sun.awt.image.ByteInterleavedRaster.getDataElements(Unknown Source)
at java.awt.image.BufferedImage.getRGB(Unknown Source)
at SimplePicture.getBasicPixel(SimplePicture.java:247)
at Pixel.setValuesFromPictureAndLocation(Pixel.java:137)
at Pixel.<init>(Pixel.java:57)
at SimplePicture.getPixel(SimplePicture.java:270)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
at java.lang.reflect.Method.invoke(Unknown Source)
```

The `java.lang.ArrayIndexOutOfBoundsException` tells us that we tried to access an array element that was outside the allowed indices. If this happens when our program is executing the program will stop and report the exception.

So how do we check that the index values are acceptable? We know that the x indices range from 0 to width - 1 and the y indices range from 0 to height - 1. So we can use 

```
x >= 0 && x < this.getWidth() && y >= 0 && y < this.getHeight()
```

Program 42: 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);
            redValue = redValue + samplePixel.getRed();
            greenValue = greenValue + samplePixel.getGreen();
            blueValue = blueValue + samplePixel.getBlue();
            count = count + 1;
        }
    }
}

// use average color of surrounding pixels
Color newColor = new Color(redValue / count,
                             greenValue / count,
                             blueValue / count);

    pixel.setColor(newColor);
}

Here is how to use this method:

> Picture p = new Picture(FileChooser.getMediaPath("flower1.jpg"));
> p = p.scaleUp(2);
> p.explore();
> p.blur(2);
> p.explore();

Figure 6.13 shows the flower from the collage made bigger, then blurred. You can see the pixilation in the bigger version—the sharp, blocky edges. With the blur,
some of that pixelation goes away. More careful blurs take into account regions of colors (so that edges between colors are kept sharp), and thus are able to reduce pixelation without removing sharpness.

![Figure 6.13: Making the flower bigger, then blurring to reduce pixellation](image)

### 6.6 BACKGROUND SUBTRACTION

Let's imagine that you have a picture of someone, and a picture of where they stood without them there (Figure 6.14). Could you subtract the background of the person (i.e., figure out where the colors are close), and then replace another background? Say, of the moon (Figure 6.15)?

![Figure 6.14: A picture of a child (Katie), and her background without her](image)

**Program 43: Subtract the background and replace it with a new one**

```c
/**
 * Method to replace the background in the current picture
```
FIGURE 6.15: A new background, the moon

```java
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 distance between the current pixel color
               * and the old background pixel color is less
               * than the 15 then swap in the new background pixel */
            if (currPixel.colorDistance(oldPixel.getColor()) < 15.0)
            {
            }
        }
    }
}
```
newPixel = newBackground.getPixel(x, y);
currPixel.setColor(newPixel.getColor());
}
}
}

To test if we can replace an old background with a new background try:

> Picture p = new Picture(chooser.getMediaPath("kid-in-frame.jpg"));
> Picture oldBg = new Picture(chooser.getMediaPath("bgframe.jpg"));
> Picture newBg = new Picture(chooser.getMediaPath("moon-surface.jpg"));
> p.swapBackground(oldBg, newBg);
> p.show();

We can, but the effect isn’t as good as we would like (Figure 6.16). Our daughter’s shirt color was too close to the color of the wall. And though the light was dim, the shadow is definitely having an effect here.

![Figure 6.16: Katie on the moon](image)

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 6.17) weren't all that comparable. The background swap (again with the jungle scene) hardly did anything at all! We changed the threshold value to 50, and finally got some swapping (Figure 6.18).

> Picture p = new Picture(chooser.getMediaPath("wall-two-people.jpg"));
> Picture oldBg = new Picture(chooser.getMediaPath("wall.jpg"));
> Picture newBg = new Picture(chooser.getMediaPath("beach.jpg"));
> p.swapBackground(oldBg, newBg);
> p.show();
Making it Work Tip: Add an input parameter to generalize a method

Notice that we changed the threshold from 15.0 to 50.0 for the second test of the `swapBackground(oldBG,newBG)` method. A better thing to do would be to change the method to take the threshold distance as another input parameter `swapBackground(oldBG,newBG,threshold)`. This means we won’t have to keep changing the method each time we want to change the threshold, which means the method can be used in more situations.

Program 44: Better Swap Background

```java
/**
 * Method to replace the background in the current picture
 * with the background from another picture
 * @param oldBackground a picture with the old background
 * to replace
 * @param newBackground a picture with the new background
 * to use
 * @param threshold if the distance between the current
 * pixel color and the background pixel color is less
 * than this amount use the new background pixel color
 */
public void swapBackground(Picture oldBackground,
                           Picture newBackground,
                           double threshold)
{
    Pixel currPixel = null;
    Pixel oldPixel = null;
    Pixel newPixel = null;

    // loop through the columns
    for (int x=0; x<getWidth(); x++)
    {
        // loop through the rows
        for (int y=0; y<getHeight(); y++)
        {
            // get the current pixel and old background pixel
            currPixel = this.getPixel(x,y);
            oldPixel = oldBackground.getPixel(x,y);

            /* if the distance between the current pixel color
            */
```
214 Chapter 6 Conditionally Modifying Pixels

```java
* and the old background pixel color is less than
* the threshold then swap in the new background
* pixel
*/
if (currPixel.colorDistance(oldPixel.getColor()) < threshold)
{
    newPixel = newBackground.getPixel(x, y);
    currPixel.setColor(newPixel.getColor());
}
}
```

To make this work pass the threshold too when invoking `swapBackground`:

```java
> Picture p = new Picture(FileChooser.getMediaPath("wall-two-people.jpg"));
> Picture oldBg = new Picture(FileChooser.getMediaPath("wall.jpg"));
> Picture newBg = new Picture(FileChooser.getMediaPath("beach.jpg"));
> p.swapBackground(oldBg,newBg,50);
> p.show();
```

![Image of two people in front of a wall, and a picture of the wall](image)

**FIGURE 6.17:** Two people in front of a wall, and a picture of the wall

### 6.7 CHROMAKEY

The way that weatherpersons appear to be in front of a weather map that changes, is that they actually stand before a background of a fixed color (usually blue or green), then subtract that color. This is called chromakey. Mark took our son’s blue sheet, attached it to the entertainment center, then took a picture of himself in front of it, using a timer on a camera (Figure 6.19).

Mark tried a new way to test for “blueness”. If the blue value was greater than the sum of the red and green values then it the color was “blue”.

Program 45: Chromakey: Replace all blue with the new background
FIGURE 6.18: Swapping a beach for the wall, using background subtraction, with a threshold of 50

FIGURE 6.19: Mark in front of a blue sheet

```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++)
        {
            // code here
        }
    }
}
```
Chapter 6

Conditionally Modifying Pixels

```java
{
    // get the current pixel
    currPixel = this.getPixel(x, y);

    /* if the color at the current pixel mostly blue
     * (blue value is greater than red and green
     * combined, then use new background color
     */
    if (currPixel.getRed() + currPixel.getGreen() < currPixel.getBlue())
    {
        newPixel = newBg.getPixel(x, y);
        currPixel.setColor(newPixel.getColor());
    }
}
```

The effect is really quite striking (Figure 6.20). Do note the “folds” in the lunar surface, though. The really cool thing is that this program works for any background that’s the same size as the image (Figure 6.21). To put Mark on the moon and on the beach try this:

```java
> Picture mark = new Picture(FileChooser.getMediaPath("blue-mark.jpg");
> Picture newBg = new Picture(FileChooser.getMediaPath("moon-surface.jpg");
> mark.chromakey(newBg);
> mark.explore();
> mark = new Picture(FileChooser.getMediaPath("blue-mark.jpg");
> newBg = new Picture(FileChooser.getMediaPath("beach.jpg");
> mark.chromakey(newBg);
> mark.explore();
```

There's another way of writing this code, which is shorter but does the same thing.

```
// Program 46: Chromakey, shorter

/ **
* Method to do chromakey using a blue background
* @param newBg the new background image to use to replace
* the blue from the current picture
*/
public void chromakeyBlue(Picture newBg)
{
    Pixel[] pixelArray = this.getPixels();
    Pixel currPixel = null;
    Pixel newPixel = null;
```
// loop through the pixels
for (int i = 0; i < pixelArray.length; i++)
{
    // get the current pixel
    currPixel = pixelArray[i];

    /* if the color at the current pixel mostly blue
     * (blue value is greater than green and red
     * combined, then use new background color
     */
    if (currPixel.getRed() + currPixel.getGreen() < currPixel.getBlue())
    {
        newPixel = newBg.getPixel(currPixel.getX(), currPixel.getY());
        currPixel.setColor(newPixel.getColor());
    }
}

FIGURE 6.20: Mark on the moon
Making it Work Tip: When do you need a different method name?
Notice that we used a different name for the new shorter method. We couldn’t have used the same name and had both methods in our Picture class since the parameters are the same. Methods can be overloaded (use the same name) as long as the parameters are different (in number, or order, or type).

You don’t really want to do chromakey with a common color, like red—something that there’s a lot of in your face. Mark tried it with the two pictures in Figure 6.22—one with the flash on, and one with it off. We changed the test to if (currPixel.getRed() > currPixel.getGreen() + currPixel.getBlue()). The one without a flash was terrible—the student’s face was replaced with the background. The one with the flash was better, but the flash is still clear after the swap (Figure 6.23). It’s clear why moviemakers and weather people use blue or green backgrounds.

6.8 CONCEPTS SUMMARY
We have covered boolean expressions, conditionally executing code using if and else, combining boolean expressions using “and” (&&) and “or” (||) and method overloading.

6.8.1 Boolean Expressions
A boolean expression is one that results in true or false. The values true and false are reserved words in Java. Here are some example boolean expressions.
FIGURE 6.22: Student in front of a red background, and with flash on

FIGURE 6.23: Using chromakey program with a red background

> int x = 20;
> System.out.println(x <= 30);
  true
> System.out.println(x > 30);
  false
> System.out.println(x == 20);
  true
> System.out.println(x != 20);
  false

Notice that to check for a variable having a value we use == not =. The = is used to assign a value to a variable, not check for equality. To check for inequality use !=.

6.8.2 Combining Boolean Expressions

You can combine boolean expressions with the Java operators && to represent “and” and || to represent “or”. When you use && both boolean expressions must be true in order for it to return true. If the first boolean expression isn’t true the second won’t even be tested. This is called short-circuit evaluation. When you use || only one boolean expression must be true in order for it to return true. With “or” if the
first boolean expression is true the second won’t be evaluated. If the first boolean expression is false the second one will still be evaluated.

```java
> int x = 3;
> int y = 5;
> System.out.println(x < 5 && y < 6);
true
> System.out.println(x > 5 && y < 6);
false
> System.out.println(x < 5 && y > 6);
false
> System.out.println(x > 5 && y > 6);
false
> System.out.println(x > 5 || y < 6);
true
```

### 6.8.3 Conditional Execution

To conditionally execute one statement use the `if` keyword followed by a boolean expression inside of an open and close parenthesis. Put the statement that you only want executed if the boolean expression is true on a new line and indent it. If the boolean expression is false then execution will continue with the next statement.

```
if (boolean expression)
    // statement to execute if the boolean expression is true
    statement
    // next statement
```

To conditionally execute a block of statements use the `if` keyword followed by a boolean expression inside of an open and close parenthesis. Put the statements to be executed when the boolean expression is true inside of an open and close curly brace. Indent the statements to make it easier to visually see that these statements will only be executed if the boolean expression is true. If the boolean expression is false execution will continue with the statement following the close curly brace.

```
if (boolean expression) {
    statements
}
```

If you want to execute one block of statements if the boolean expression is true and another if it is false use the `else` keyword as well. Put the statements that you want to execute when the boolean expression is true inside of an open and close curly brace after the `if(booleanExpression)`. Next, add the keyword `else` and put the statements that you want executed when the boolean expression is false inside of an open and close curly brace.

```
if (boolean expression) {
    statements
}
else {
    statements
}
```
Section 6.8 Concepts Summary

If you have 3 or more options use nested if and else.

```java
if (boolean expression) {
    statements
} else if (boolean expression) {
    statements
} else {
    statements
}
```

If you have 4 options start with an if (boolean expression) and have two else if (boolean expression) and a final else. The last else is optional.

PROBLEMS

6.1. Try doing chromakey in a range—grab something out of its background where the something is only in one part of a picture. For example, put a halo around someone’s head, but don’t mess with the rest of their body.

6.2. Write a method to copy all but the white pixels from one picture to another. Use this to put the robot in robot.jpg on the moon in moon-surface.jpg.

6.3. Start with a picture of someone you know, and make some specific color changes to it:

- Turn the skin green
- Turn the eyes red.
- Turn the hair orange.

Of course, if you’re friend’s skin is already green, or eyes red, or hair orange—choose a different target color.

6.4. Which of the methods below removes all the blue from every pixel of a picture that already has a blue value of more than 100?

1. A only
2. D only
3. B and C
4. C and D
5. None
6. All

What do the other ones do?

A.

```java
public void blueOneHundred() {
    Pixel p = null;

    for (int x = 0; x < 100; x++) {
        for (int y = 0; y < 100; y++) {
            p = getPixel(x, y);
```

```java
```
6.5. Convert the method `copyFlowerLarger()` to a method that can scale any picture up. It should return a new picture object created using `new Picture(this.getWidth() * 2, this.getHeight() * 2)`.

6.6. Write the method to turn the lightest areas of a picture gray to simulate a fog.

6.7. Write other edge detection methods. Try comparing the current pixel intensity with the one on the right. Try comparing the current pixel to the average of the pixels to the right and below.

6.8. What would the output from the following be:

B.

```java
public void blueChange() {
    Pixel[] pixelArray = getPixels();
    Pixel pixel = null;

    for (int i = 0; i < pixelArray.length; i++) {
        pixel = pixelArray[i];
        if (pixel.getBlue() > 0) {
            pixel.setBlue(100);
        }
    }
}
```

C.

```java
public void clearSomeBlue() {
    Pixel[] pixelArray = getPixels();
    Pixel pixel = null;

    for (int i = 0; i < pixelArray.length; i++) {
        pixel = pixelArray[i];
        if (pixel.colorDistance(Color.BLUE) > 100) {
            pixel.setBlue(0);
        }
    }
}
```

D.

```java
public void setBlue() {
    Pixel[] pixelArray = getPixels();
    Pixel pixel = null;

    for (int i = 0; i < pixelArray.length; i++) {
        pixel = pixelArray[i];
        if (pixel.getBlue() > 100) {
            pixel.setBlue(0);
        }
    }
}
```
Section 6.8 Concepts Summary 223

```java
int x = 30;
for (int i=x; i < 40; i++) {
    if (i < 35) System.out.println("i is less than 35");
    else if (i == 35) System.out.println("i is 35");
    else System.out.println("i is greater than 35");
}
```

6.9. Write a method to do green or red chromakey.
6.10. Write a method to do chromakey in a specified rectangular region.
CHAPTER 7

Drawing

7.1 DRAWING ON IMAGES USING THE GRAPHICS CLASS
7.2 PROGRAMS AS SPECIFYING DRAWING PROCESS
7.3 USING GRAPHICS2D FOR ADVANCED DRAWING
7.4 CONCEPT SUMMARY

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

• To be able to draw simple shapes (lines, ovals, rectangles, arcs) on existing pictures.
• To create pictures by drawing them.
• To use the Java2D API for more complicated drawing.

The computer science goals for this chapter are:

• To be able to choose between using vector and bitmapped image formats.
• To be able to choose when one should write a program for a task versus using existing applications software.
• To promote reuse by working with existing classes.
• To introduce the concept of inheritance.
• To introduce the concept of an interface.

7.1 DRAWING ON IMAGES USING THE GRAPHICS CLASS

Professional photographers have a problem in that it is easy for people to scan pictures and print many copies of them. How can they allow people to see proofs of pictures, but discourage people from using their pictures without paying for them? One way that professional photographers protect their images is to put some text on them. People can still scan the picture and make copies, but it is obvious that the picture is stolen. The text also hides some of the picture so people are more likely to pay for pictures without the text.

How can you add text to a picture? 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. Here’s an example that creates a grid of
Section 7.1 Drawing on Images Using the Graphics Class 225

lines on a picture (Figure 7.1). It works by simply setting all the pixels in a line to black! The gap between the lines is 20 pixels.

Program 47: Draw lines by setting pixels

```java
/**
 * Method to draw a grid on a picture
 */
public void drawGrid() {
    Pixel pixel = null;

    // Draw the horizontal lines
    for (int y = 20; y < this.getHeight(); y+=20) {
        for (int x = 0; x < this.getWidth(); x++) {
            pixel = this.getPixel(x, y);
            pixel.setColor(Color.black);
        }
    }

    // draw the vertical lines
    for (int x = 20; x < this.getWidth(); x+=20) {
        for (int y = 0; y < this.getHeight(); y++) {
            pixel = this.getPixel(x, y);
            pixel.setColor(Color.black);
        }
    }
}
```

To test this method create a Picture object and then invoke the method on the Picture.

```java
> Picture pictureObj = new Picture(FileChooser.getMediaPath("barbara.jpg"));
> pictureObj.show();
> pictureObj.drawGrid();
> pictureObj.repaint();
```

This method first draws the horizontal lines by setting \(y\) to a value of 20 and incrementing \(y\) by 20, while \(x\) starts at 0 and is incremented by 1. Next the method draws the vertical lines by setting \(x\) to 20 and incrementing \(x\) by 20 while \(y\) starts at 0 and is incremented by 1. To draw more lines decrease the start and increment values and to draw less lines increase the start and increment values.
FIGURE 7.1: Adding a grid of lines to a picture (barbara.jpg)

Making it Work Tip: Working with Color Objects

You may notice that this program is using the pre-defined Color object (object of the Color class) Color.black. Remember that 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. You can also create a Color object by providing the red, green, and blue values (between 0 and 255) using new Color(red,green,blue). For example to create a pure black color you would use new Color(0,0,0) and to create a pure white color you would use new Color(255,255,255).

We can imagine drawing anything we want like this, by simply setting individual pixels to whatever colors we want. 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. While we could do it, it would involve a lot of work to do all the math for all the different shapes and letters. That’s work that lots of people want done, so instead, the basic drawing has been built into Java for you.
7.1 Drawing with Graphics methods

Most modern programming languages with graphics libraries provide ways to draw a variety of shapes and text. In Java the Graphics class in the java.awt package has a number of methods that allow you to draw simple shapes and text. These graphics methods work similar to painting. First you pick a color to use to paint with and then you can paint several shapes using that color. If one shape is drawn over another it will cover the shape underneath.

Computer Science Idea: Packages
A package is a collection of related classes. Java defines a large number of classes and it would be overwhelming if they weren’t grouped into packages. Some common packages are java.lang which has the basics of the Java language in it, java.awt which has the original graphics classes in it, and java.io which contains classes used for input and output.

- setColor(Color color) sets the color to use for drawing.
- drawLine(int x1, int y1, int x2, int y2) draws a line from position (x1,y1) to (x2,y2) using the current color.
- drawRect(int x1, int y1,int w, int h) draws the outline of a rectangle with the upper left corner at (x1,y1), a width of w, and a height of h using the current color.
- fillRect(int x1, int y1, int w, int h) draws a filled (solid) rectangle with the upper left corner at (x1,y1), a width of w, and a height of h using the current color.
- drawOval(int x1, int y1, int w, int h) draws the outline of an oval with the upper left corner of the enclosing rectangle at (x1,y1), the width of the enclosing rectangle is w and the height of the enclosing rectangle is h using the current color.
- fillOval(int x1, int y1, int w, int h) draws a filled (solid) oval with the upper left corner of an enclosing rectangle at (x1,y1), the width of the enclosing rectangle is w and the height of the enclosing rectangle is h using the current color.
- drawArc(int x1, int y1, int w, int h, int startAngle, int arcAngle) draws an outline of an arc that is part of an oval that fits in the enclosing rectangle at (x1,y1), the width of the enclosing rectangle is w and the height of the enclosing rectangle is h. The arc starts at the given startAngle and extends arcAngle degrees (where 0 degrees is at the 3 o'clock position on a clock and 45 degrees goes through the upper right corner of the enclosing rectangle). The ending angle is the startAngle plus arcAngle.
• fillArc(int x1, int y1, int w, int h, int startAngle, int arcAngle) draws a filled arc that is part of an oval that fits in the enclosing rectangle at \((x1, y1)\), the width of the enclosing rectangle is \(w\) and the height of the enclosing rectangle is \(h\). The arc starts at the given \(startAngle\) and extends \(arcAngle\) degrees (where 0 degrees is at the 3 o’clock position on a clock and 45 degrees goes through the upper right corner of the enclosing rectangle). The ending angle is the \(startAngle\) plus \(arcAngle\).

• drawPolygon(int[] xArray, int[] yArray, int numPoints) draws the outline of a closed polygon using the x values in \(xArray\) and the y values in \(yArray\) using the current color.

• fillPolygon(int[] xArray, int[] yArray, int numPoints) draws a filled closed polygon using the x values in \(xArray\) and the y values in \(yArray\) using the current color.

Making it Work Tip: Use the Java API
Java is a large language and it is nearly impossible to know every method for every class. Use the application program interface (API) documentation to see the methods that are available for a class. If you look at the documentation for the Graphics class in the java.awt package you will see all of the methods defined for that class. We are only showing some of the most commonly used methods here. To view the API documentation go to: http://java.sun.com/. Find the specification for the version of the language that you are using. Click on the package in the top left window frame and then click on the class name in the bottom left window frame. The documentation for that class will appear on the right. Scroll down to “Method Summary” (Figure 7.2). This gives an alphabetical listing of all of the methods in defined in that class.

We can use these commands to add simple shapes to existing pictures. What would it look like if a mysterious red box washed up on the shore of a beach? We will need to get a Graphics object to use for the drawing. We can get one from a Picture object using the method getGraphics(). When you draw you set the color using setColor(Color color) and then do any drawing commands. We can make a box appear on a picture of a beach with this method (Figure 7.3).

Program 48: Adding a box

```java
/** *
 * Method to add a solid red rectangle to the current picture *
 */
public void addBox()
```
Section 7.1 Drawing on Images Using the Graphics Class

List of packages

List of classes in this package

Documentation for the class: Graphics

FIGURE 7.2: Viewing the Java API for java.awt.Graphics

FIGURE 7.3: A box washed up on the shore of the beach

```java
{
    // get the graphics context from the picture
    Graphics g = this.getGraphics();

    // set the color to red
    g.setColor(Color.red);

    // draw the box as a filled rectangle
    g.fillRect(150, 200, 50, 50);
}
```

Execute this method using:

```bash
> Picture p = new Picture(Chooser.getMediaPath("beach-smaller.jpg"));
```
Chapter 7 Drawing

> p.addBox();
> p.show();

This method isn’t very reusable. The only way to change it to work for other rectangles is to modify the color and rectangle information and then recompile. If we want this to work on any rectangle then we will want to pass parameters to make the method more general.

Program 49: General draw box

```java
/**
 * Method to draw a filled box on the current picture
 * @param color the color to draw the box with
 * @param topLeftX the top left x coordinate of the box
 * @param topLeftY the top left y coordinate of the box
 * @param width the width of the box
 * @param height the height of the box
 */
public void drawBox(Color color, int topLeftX, int topLeftY, int width, int height)
{
    // get the graphics context for drawing
    Graphics g = this.getGraphics();

    // set the current color
    g.setColor(color);

    // draw the filled rectangle
    g.fillRect(topLeftX, topLeftY, width, height);
}
```

We could use this more general method to generate the same picture by:

> Picture p = new Picture(FileChooser.getMediaPath("beach-smaller.jpg"));
> p.drawBox(java.awt.Color.red,150,200,50,50);
> p.show();

The advantage of the method `drawBox` over the method `addBox` is that it can be used to draw any rectangle of any color on any picture.

Below is another example of using the simple drawing commands (Figure 7.4).

Program 50: An example of using drawing commands

```java
/**
 * Method to show different drawing capabilities. It will draw a string, a line, a filled rectangle, the outline of a rectangle, the outline of an oval,
 */
```
* and a filled arc.
*/

public void drawExample() {
    // get the graphics object to use for drawing
    Graphics graphics = this.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 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
    */
Chapter 7 Drawing

FIGURE 7.4: An example drawn picture

```java
    graphics.fillArc(400,300,200,150,0,180);
}
```

To try this out create a picture from the blank 640 by 480 file. Then invoke the method `drawExample()` on the picture.

```java
> Picture p = new Picture(FileChooser.getMediaPath("640x480.jpg"));
> p.drawExample();
> p.show();
```

How would you draw a simple face (Figure 7.5)? You could draw an oval for the head. You could use filled ovals for the eyes. You could use arcs for the mouth and eyebrows.

Program 51: An example of using oval and arc drawing commands

```java
/**
 * Method to draw a face to demonstrate drawing
 * ovals and arcs
 */
public void drawFace()
{
    // get the graphics object to use for drawing
    Graphics graphics = this.getGraphics();

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

    // draw the oval for the face
```
Section 7.1  Drawing on Images Using the Graphics Class  233

FIGURE 7.5: A drawn face (left) and the face with enclosing rectangles (right)

```java
graphics.drawOval(130,50,380,380);
// draw the ovals for the eyes
graphics.fillOval(225,155,40,40);
graphics.fillOval(375,155,40,40);
// draw the arcs for the eyebrows
graphics.drawArc(225,145,40,40,45,90);
graphics.drawArc(375,145,40,40,45,90);
// draw the arc for the mouth
graphics.drawArc(190,85,255,255,-45,-90);
}
```

To try this out create a picture from the blank 640 by 480 file. Then invoke the method `drawFace()` on the picture.

```java
> Picture p = new Picture(FileChooser.getMediaPath("640x480.jpg"));
> p.drawFace();
> p.show();
```

**Making it Work Tip: Drawing Ovals and Arcs**

Notice that to draw an oval or arc you give the upper left corner of the enclosing rectangle, not the upper left corner of the oval or arc (Figure 7.5). Arcs specify the starting angle, but not the ending angle. The ending angle is the starting angle added to the arc angle. Graph paper and a protractor can help you plan your drawing.

7.1.2  Vector and bitmap representations

Here’s a thought: Which of these is smaller—the picture (Figure 7.4) or the method Program 50 (page 230)? The picture, on my disk, is about 31 kilobytes (a kilobyte is a thousand bytes). The program is less than 2 kilobytes. 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 Program 7.4 (page 232). 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 the 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.

What does *compression* mean? It means that various techniques have been used to make the file smaller. Some compression techniques are *lossy compression*—some detail is lost, but hopefully the least significant (perhaps even invisible to the human eye, or ear) detail. Other techniques, *lossless compression* lose no detail, but still scrunch the file. One of the lossless techniques is *run length encoding* (RLE).

Imagine that you’ve got a long line of yellow pixels in a picture, surrounded by some blue pixels. Something like this:

```
B B Y Y Y Y Y Y Y Y B B
```

What if you encoded this, not as a long line of pixels, but as something like:

```
B B 9 Y B B
```

In words, you encode “blue, blue, then 9 yellows, then blue and blue.” Since each of those yellow pixels takes 24 bits (3 bytes for red, green, and blue), but recording “9” takes just a single byte, there’s a huge savings. We say that we’re encoding the *length* of the *run* of yellows—thus, run length encoding. That’s just one of the compression methods that gets used to make pictures smaller.

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—in a sense, vector notation is already compressed. 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
Section 7.1  Drawing on Images Using the Graphics Class

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 information as 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.

7.1.3 Drawing Text (Strings)

In order to add text to an image we can use the `java.awt.Graphics` method `drawString(String str, int x, int y)`. This will draw the passed string at the passed x and y position. However, the x and y position are not the top left corner as is usual with the drawing methods. The x and y position give the baseline of the string which is the line on which you would write the string if you were using lined paper (Figure 7.6).

When you use the `drawString` method the string will be drawn in the current color and *font*. The font specifies what the characters in the string will look like, what style will be used (bold, italic, plain), and how big the characters will be. You have seen that you can set the color using `setColor(Color colorObj)`. You can also set the font using `setFont(Font fontObj)`.

To create a *Font* object you specify the font name, style, and size: `new Font(String name, int style, int size)`. You can always use any of Dialog,
DialogInput, Monospaced, Serif, or SansSerif for font names. You can get an array of all of the available font names using:

```java
> import java.awt.*;
> GraphicsEnvironment env = GraphicsEnvironment.getLocalGraphicsEnvironment();
> String[] nameArray = env.getAvailableFontFamilyNames();
> for (int i=0; i < nameArray.length; i++)
    System.out.println(nameArray[i]);
```

The method to draw a passed string on the current picture at the passed x and y is:

### Program 52: Draw a string on a picture

```java
/**
 * Method to draw a string on the current picture
 * @param text the string to draw
 * @param x the x location to start at
 * @param y the y location of the baseline
 */
public void drawString(String text, int x, int y)
{
    // get the graphics object
    Graphics g = getGraphics();

    // set the color
    g.setColor(Color.black);

    // set the font
    g.setFont(new Font("Arial", Font.BOLD, 24));

    // draw the string
    g.drawString(text,x,y);
}
```

To use this program you can use the picture explorer to determine where you want the baseline of the string to be and then use the `drawString` method to draw the string on the picture. Our son, Matthew, took a picture of a kitten on a trip to Greece. Let’s add a string that explains the picture near the bottom of the picture (Figure 7.7).

```java
> Picture p = new Picture(FileChooser.getMediaPath("kitten.jpg");
> p.explore();
> p.drawString("Matt’s picture of a kitten in Greece",67,283);
> p.explore();
```
The string isn’t quite in the center of the picture. What if we want it to be in the center? How could we calculate the starting x position for the string such that the resulting string is centered? We know that the center of the picture horizontally is at the half the width of the picture (\texttt{int}) \texttt{(picture.getWidth() \times 0.5)}. If we subtract half the length of the string in pixels from the center of the picture that would center the string. How do we calculate the length of the string in pixels? The length of the string depends on the number of characters in the string but also on the font used to draw the string.

To get information about the length of the string in the number of pixels drawn we can use the \texttt{FontMetrics} class which is in package \texttt{java.awt}. To get a \texttt{FontMetrics} object use \texttt{g.getFontMetrics()} where \texttt{g} is a \texttt{Graphics} object. The \texttt{FontMetrics} class contains methods for getting information about the display of a font. For example, we can get the length in pixels of a string using the method \texttt{stringWidth(String str)}. We could get the height in pixels of a string drawn in the current font using the method \texttt{getHeight()}. We could get the length of the descent (part of character like ‘p’ below the baseline) using \texttt{getDescent()}.

**Program 53: Drawing a string centered horizontally on a picture**

```java
/**
 * Method to draw a horizontally centered string
 * on the current picture
 * @param text the string to draw
 * @param x the x location to start at
 * @param y the y location of the baseline
 */
public void drawHorizontalCenteredString(String text, int y)
```
FIGURE 7.8: Drawing a string centered on a picture

```java
{  
    // get the graphics object
    Graphics g = getGraphics();

    // create the font object
    Font font = new Font("Arial", Font.BOLD, 24);

    // set the color
    g.setColor(Color.black);

    // set the font
    g.setFont(font);

    // get the font metrics
    FontMetrics fontMetrics = g.getFontMetrics();

    // get the width of the string
    int strWidth = fontMetrics.stringWidth(text);

    // calculate the center of the picture
    int center = (int) (this.getWidth() * 0.5);

    // draw the string centered in x
    g.drawString(text,
                 center - (int) (strWidth * 0.5),
                 y);
}
Another thing we can do with drawing is create pictures that are exactly specified—things that might be too hard to do by hand. Take, for example, Figure 7.9. This is a rendering of a famous optical illusion, and it’s not as effective as the famous ones—but it’s simple to understand how this version 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 7.9 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 specific exactly what that picture should be.

Program 54: Draw the gray effect

```java
/**
 * Method to draw a gray effect picture on the current picture
 */
public void drawGrayEffect()
{
   // 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++)
         this.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++)
      for (int y=0; y < 100; y++)
         this.getPixel(x,y).setColor(new Color(grayLevel,grayLevel,grayLevel));
```

FIGURE 7.9: A programmed gray scale effect
Chapter 7  Drawing

```java
    this.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++)
            this.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++)
            this.getPixel(x,y).setColor(medGray);
```

To use this method create a picture of the file that has a blank 640 by 480 picture in it. Invoke the method on that picture.

```java
> Picture p = new Picture(FileChooser.getMediaPath("640x480.jpg"));
> p.drawGrayEffect();
> p.show();
```

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 mathematical relationships.

**Program 55: Draw the picture in Figure 7.10**

```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 on the current picture
   */
public void drawFilledRectangles()
{
    Graphics g = this.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);
    }
}
```

To use this method create a picture of the file that has a blank 640 by 480 picture in it. Invoke the method on that picture.
Section 7.2 Programs as Specifying Drawing Process

241

> Picture p = new Picture(chooser.getMediaPath("640x480.jpg"));
> p.drawFilledRectangles();
> p.show();

FIGURE 7.10: Nested colored rectangles

Program 56: Draw the picture in Figure 7.11

/**
 * Method to draw a picture with a succession of
 * rectangles on the current picture
 */
public void drawRectangles()
{
    Graphics g = this.getGraphics();
    Color color = null;

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

To use this method create a picture of the file that has a blank 640 by 480 picture in it. Invoke the method on that picture.

> Picture p = new Picture(chooser.getMediaPath("640x480.jpg"));
> p.drawRectangles();
> p.show();
7.2.1 Why do we write programs?

Why do we write programs, especially to draw pictures? Couldn’t we draw pictures like these in Photoshop or Visio? 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—Photoshop isn’t simple.

But if I give you these methods (programs), you can create the picture anytime you want. What’s more, by giving you the methods, 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, write a program and give the people the program.

7.3 USING GRAPHICS2D FOR ADVANCED DRAWING

We have used the `java.awt.Graphics` object to draw simple shapes like lines, rectangles, ovals, arcs, and polygons. The Graphics object is like a painter who picks up a brush with the color you set and paints a shape with it. If you ask the painter to paint another shape over a previous one the new shape will cover the original shape. But, what if we want to use a thicker ‘brush’ when we draw? What if we want shapes to combine rather than occlude? What if we want to treat our shapes as objects?
The `java.awt.Graphics` class is okay for simple drawing but lacks many advanced features. However, the class `java.awt.Graphics2D` which is part of the Java 2D API can be used for more advanced drawing. Some of the capabilities of a `Graphics2D` object are:

- You can set the width of the 'brush' (pen). You can also set the style of the 'brush' to do different types of dashed lines.
- You can rotate, translate, scale, or shear what you are drawing.
- You can fill a shape with more than just a solid color. You can fill a shape with a gradient or a texture.
- You can change what happens when objects overlap.
- You can clip objects so that only the part visible inside the clipping area is drawn. This is like using a stencil.
- You can set rendering hints to make your curves smoother using anti-aliasing if it is available.

### 7.3.1 Setting the Brush Width

To use these advanced features you will need a `Graphics2D` object. To get a `Graphics2D` object you must cast the `Graphics` object to a `Graphics2D` object using `Graphics2D g2 = (Graphics2D) graphics`. All `Graphics` objects that you get are really `Graphics2D` objects, so this is allowed.

You can set the 'brush' options using `setStroke`. You set the color, gradient, or texture to use for painting with the method `setPaint(Paint p)`. Instead of methods that draw shapes the `Graphics2D` class has you create objects and either `draw` them to draw the outline or `fill` them to fill with the current paint information. Here is a simple program that draws a red x on the current picture using a brush (stroke) with the specified width.

```java
/**
 * Method to add two crossed lines to a picture.
 * One line will go from the top left corner to the
 * bottom right corner. The other will go from the
 * bottom left corner to the top right corner.
 */
public void drawWideX(Color color, float width)
{
    // get the Graphics2D object
    Graphics graphics = getGraphics();
    Graphics2D g2 = (Graphics2D) graphics;

    // set the color and brush width
```
Chapter 7  Drawing

```java
    g2.setPaint(color);
g2.setStroke(new BasicStroke(width));
    // get the max x and y values
    int maxX = getWidth() - 1;
    int maxY = getHeight() - 1;
    // draw the lines
    g2.draw(new Line2D.Double(0, 0, maxX, maxY));
g2.draw(new Line2D.Double(0, maxY, maxX, 0));
```

You can use this program to add a wide red 'X' to a picture as shown in Figure 7.12.

```java
> Picture p = new Picture(Chooser.getMediaPath("grayMotorcycle.jpg"));
> p.drawLineX(java.awt.Color.red, 5);
> p.show();
```

Making it Work Tip: Creating and Drawing Shapes with Graphics2D
Notice that we created a Line2D.Double object and then asked the Graphics2D object named g2 to draw this object. This is different from how we drew shapes using the Graphics class. With the Graphics2D class you create geometric objects and either draw or fill them. The Line2D.Double probably looks strange to you. This is actually the name of a class in the java.awt.geom package. Even though java.awt.geom and java.awt both start the same they are different packages. If you imported all classes in the package java.awt using the wildcard "*" you still wouldn't have imported the classes in the java.awt.geom package. You need at least two import statements if you are using classes from both of these packages.

7.3.2 Copying Pictures by Drawing Images
Both the Graphics and Graphics2D classes have methods for drawing images. You can use these methods to copy a picture to the current picture object. You don't have to copy the colors pixel by pixel. The method to do this is drawImage.

There are many methods with the name drawImage. Recall that method overloading allows methods with the same name as long as the parameters are different. Look at the API for the Graphics class to see all the drawImage methods. Which method would you use to copy all the pixels in one image (picture) to another at a particular location?
Section 7.3 Using Graphics2D for Advanced Drawing

FIGURE 7.12: Drawing a red X on a picture

Notice that the `drawImage` methods take an `Image` object not a `Picture` object. The `Picture` class is one we created, but it contains a method that will get you an `Image` object: `getImage()`.

So, to copy a picture to the current picture object you need to get the `Graphics` object to use to draw on the current picture. You can do this using the method `getGraphics()`. Next draw the `Image` that you get from the passed `Picture` (using the method `getImage()`) at the passed x and y location using the method `drawImage(Image img, int x, int y, ImageObserver observer)`. What is an `ImageObserver`? It is an object that wants to be notified as the image is changed. It can be `null` to say that no object wants to be notified.

Program 58: Copy a picture to this picture

```java
/**
 * Method to copy the passed picture into the current picture at the given x and y position in the current picture
 * @param source the picture to copy
 * @param x the x of the upper left corner to copy to
 * @param y the y to the upper left corner to copy to
 */
public void copy(Picture source, int x, int y) {
    Graphics g = this.getGraphics();
    g.drawImage(source.getImage(), x, y, null);
}
```
You can use this program to copy a turtle to the beach using:

```java
> Picture p1 = new Picture(Chooser.getMediaPath("beach.jpg"));
> p1.explore();
> Picture p2 = new Picture(Chooser.getMediaPath("turtle.jpg"));
> p1.copy(p2,194,304);
> p1.show();
```

**FIGURE 7.13:** Drawing a turtle on a beach

Now change the `copy` method to use a `Graphics2D` object. Will it still compile? Try it and see. Does the `Graphics2D` class have the `drawImage` method?

**Program 59: Copy a picture to this picture using Graphics2D**

```java
/**
 * Method to copy the passed picture into the current picture at the given x and y position in the current picture
 * @param source the picture to copy
 * @param x the x of the upper left corner to copy to
 * @param y the y to the upper left corner to copy to
 */
public void copy2D(Picture source, int x, int y) {
    // get the graphics object
    Graphics g = this.getGraphics();
    Graphics g2 = (Graphics2D) g;
    // copy the passed picture into this picture
    g2.drawImage(source.getImage(), x, y, null);
    // restore the graphics object
    g2.dispose();
    g = this.getGraphics();
}
```
Section 7.3 Using Graphics2D for Advanced Drawing

Why does this work? The class `Graphics2D` inherits from the class `Graphics`. What does that mean? Have you heard of children inheriting features or abilities from their parents like eye color, hair color, or musical ability? A child class inherits from a parent class. What can it inherit? Classes don’t have eye color, but they do define data (fields) and behavior (methods). A child class inherits data (fields) and behavior (methods).

Unlike people, classes inherit all of the fields and methods of the parent class and they can only have one parent. An object of a child class can invoke public methods that are defined in a parent class just as if they were defined in the child class. An object of a child class can also access any public fields in a parent class as if they were defined in the child class.

What about private fields and methods? While these are inherited they cannot be directly accessed by a child object, nor should they be. The standard way to work with private fields is to provide public methods that can modify or access the private fields, but keep the data in a valid state. Private methods are only meant to be used inside the class they are defined in so you wouldn’t want a child directly using those.

How can we tell that the class `Graphics2D` inherits from the class `Graphics`? If you look at the API for `Graphics2D` you will see that the API starts with the package name. On the next line is the class name. Following that is the ancestor tree of the class. It shows all the ancestors of the class starting with `java.lang.Object`. The class `java.awt.Graphics2D` class inherits from the class `java.awt.Graphics` which inherits from the class `java.lang.Object`. You can also tell who the parent class is by finding the class name given after the extends keyword in the class declaration.

FIGURE 7.14: Documentation for Graphics2D
7.3.3 General Scaling

We have shown how to scale an image up by copying the same pixel more than once. We have shown how to scale an image down by skipping every other pixel. The methods that we wrote would double the size of the original image or reduce it by half. But, what if we want to scale up in x and down in y? What if we want to scale to a specific size? The Graphics2D class provides ways to handling scaling using an AffineTransform object in package java.awt.geom to handle the transformation. Here is a program that will scale a picture by the passed x and y factors.

Program 60: General Scale Method

```java
/**
 * Method to create a new picture by scaling the current picture by the given x and y factors
 * @param xFactor the amount to scale in x
 * @param yFactor the amount to scale in y
 * @return the resulting picture
 */
public Picture scale(double xFactor, double yFactor) {
    // set up the scale transform
    AffineTransform scaleTransform = new AffineTransform();
    scaleTransform.scale(xFactor, yFactor);

    // create a new picture object that is the right size
    Picture result = new Picture((int) (getWidth() * xFactor),
                                (int) (getHeight() * yFactor));

    // get the graphics 2d object to draw on the result
    Graphics graphics = result.getGraphics();
    Graphics2D g2 = (Graphics2D) graphics;

    // draw the current image onto the result image scaled
g2.drawImage(this.getImage(), scaleTransform, null);

    return result;
}
```
You can use the `scale` method to create a new picture from the original picture by scaling up or down by any amount in x and/or y.

```java
> Picture p = new Picture(Chooser.getMediaPath("mattDoor.jpg"));
> Picture p1 = p.scale(2.0, 0.5);
> p1.show();
```

FIGURE 7.15: Original picture and picture scaled up 2 times in x and down by half in y

Making it Work Tip: Build from Working Parts

Earlier we wrote a method to scale a picture by copying colors from a source picture to a target picture pixel by pixel. Here we are using classes defined as part of the Java language to do that work for us. There are many classes in Java available for you to use. When you start solving a problem you don’t have to create everything you need from scratch. Instead you can assemble your solution from classes that already exist. This is one of the goals of object-oriented programming: make things reusable by encapsulating them in objects defined by classes.

7.3.4 Shearing

One of the effects that is easy to do with an `java.awt.geom.AffineTransform` object is to shear the image. Shearing by 1.0 in x moves each row over by the y index amount. So the first row, with a y index of 0, starts at (0, 0). The second row, with a y index of 1, starts at (1, 1). The third row, with a y index of 2, starts at (2, 2). The last row’s x will start at \((height - 1, height - 1)\). To figure out how big our new picture will need to be no matter what the amounts are we are shearing by use the `getTranslationEnclosingRectangle(AffineTransform trans)` method which takes an `AffineTransform` object and returns a rectangle that will enclose
the picture that results from applying that AffineTransform object to the current Picture object.

Program 61: General Shear Method

```java
/**
 * Method to create a new picture by shearing the current picture by the given x and y factors
 * @param xFactor multiplier to use to shift in x direction based on y index
 * @param yFactor multiplier to use to shift in y direction based on x index
 * @return the resulting picture
 */
public Picture shear(double xFactor, double yFactor)
{
    // set up the shear transform
    AffineTransform shearTransform = new AffineTransform();
    shearTransform.shear(xFactor, yFactor);
    Rectangle2D rect =
        getTranslationEnclosingRectangle(shearTransform);

    /* create a new picture object big enough to hold the result */
    Picture result = new Picture(
        (int) (Math.ceil(rect.getWidth())),
        (int) (Math.ceil(rect.getHeight())));

    // get the graphics 2d object from the result
    Graphics graphics = result.getGraphics();
    Graphics2D g2 = (Graphics2D) graphics;

    /* save the current transformation and set-up to center the new image */
    AffineTransform savedTrans = g2.getTransform();
    AffineTransform centerTrans = new AffineTransform();
    centerTrans.translate(0 - rect.getX(), 0 - rect.getY());
    g2.setTransform(centerTrans);

    // draw the current image onto the result image sheared
    g2.drawImage(this.getImage(), shearTransform, null);

    // reset g2 transformation to the saved one
    g2.setTransform(savedTrans);

    return result;
}
```
You can use the `shear` method to create a new picture from the original picture by shearing by any amount in x and/or y.

```java
> Picture p = new Picture(FileChooser.getMediaPath("mattDoor.jpg"));
> Picture p1 = p.shear(1.0,0.0);
> p1.show();
```

![Picture sheared by 1.0 in x](image)

**FIGURE 7.16:** Picture sheared by 1.0 in x

### 7.3.5 Drawing with a GradientPaint

Instead of just filling with a solid color you can fill with a blend of two colors (a gradient). You have to specify the two colors and the rectangular area where they transition from one color to another in the coordinates of the shapes you are drawing.

What if you want to add a sun that changes color from yellow at the top to red at the bottom to a picture? You can create a `java.awt.GradientPaint` object and set that to be the paint using the method `setPaint(Paint p)`. A `GradientPaint` object is a kind of `Paint` object because the class `GradientPaint` implements the `Paint` interface.

### 7.3.6 Interfaces

What is an interface? An interface in Java is a special kind of class that only has abstract methods (methods with just a method declaration and no body) and perhaps some constants defined in it. Even though a class can only inherit from one parent class a class can implement many interfaces. An object of a class that
Chapter 7 Drawing

Computer Science Idea: Constants

Constants are variables that are declared with the keyword final which means that the value of the variable can’t change. Constants are usually also declared with the keyword static so that they are allocated in the object that defines the class and not in each object of the class. You don’t need each object of a class to have a copy of a constant. The convention in Java is to name constants with all capital letters and use '_' between words (like AlphaComposite.SRC_OVER). Notice that if a constant is defined with the keyword static it can be referred to using ClassName.CONSTANT_NAME.

So even though the class GradientPaint inherits from java.lang.Object it implements the interface java.awt.Paint and so objects of GradientPaint can be passed to methods that expect objects of the type Paint. The classes Color and TexturePaint also implement the Paint interface which is why they can also be passed to the method setPaint(Paint p).

Computer Science Idea: Interface

An interface is a boundary between two things. You may be familiar with the USB interface. It allows several different types of devices to communicate with a computer. You can hook up a mouse, a camera, or a disk drive and the computer will know how to communicate with the device because they understand the same interface.

Program 62: Overlap Pictures Using AlphaComposite

```java
/**
 * Method to add a gradient painted sun to the current picture
 * @param x the x location for the upper left corner of the rectangle enclosing the sun
 * @param y the y location for the upper left corner of the rectangle enclosing the sun
 * @param width the width of the enclosing rectangle
 * @param height the height of the enclosing rectangle
 */
public void drawSun(int x, int y, int width, int height) {
    // get the graphics2D object for this picture
    Graphics g = this.getGraphics();
    Graphics2D g2 = (Graphics2D) g;
```
Section 7.3 Using Graphics2D for Advanced Drawing

```java
// create the gradient for painting from yellow to red with
// yellow at the top of the sun and red at the bottom
float xMid = (float) (width / 0.5 + x);
GradientPaint gPaint = new GradientPaint(xMid, y,
    Color.yellow,
    xMid, y + height,
    Color.red);

// set the gradient and draw the ellipse
g2.setPaint(gPaint);
g2.fill(new Ellipse2D.Double(x, y, width, height));
```

You can use this program to add a sun to the beach.

```java
> Picture p = new Picture(FileChooser.getMediaPath("beach.jpg"));
> p.drawSun(201,80,40,40);
> p.show();
```

FIGURE 7.17: A beach with a sun that is filled with a gradient from yellow to red

7.3.7 Blending Pictures Using AlphaComposite

In the `blendPictures` method we blended two pictures by multiplying the pixel color values by 0.5 and adding them. It is also possible to use an `AlphaComposite` object from package `java.awt` to do blending or transparency effects. The default is to replace the color at overlapping pixels with the new color. To modify this default you can get an `AlphaComposite` object that specifies how you want to handle overlapping pixels in the source and destination using the class method `AlphaComposite.getInstance(int rule, float alpha)` that returns an `AlphaComposite` object to handle the overlap. For the rule you can use one of the
predefined constants such as `SRC_OVER` which draws the new pixels on top of the old pixels. The alpha value can be between 0.0 and 1.0 where 0.0 is fully transparent (invisible) and 1.0 is fully opaque. If we use an alpha value of 0.5 we will achieve the same effect as the method `blendPictures` without having to loop through all the pixels and calculate the resulting color.

To overlap two pictures we will draw the part of the first picture before a given overlap point using one of the `drawImage` methods inherited from `Graphics` that specifies the rectangular area to draw to and the rectangular area to draw from. We will draw the first part using an `AlphaComposite` that uses an alpha value of 1.0 (opaque) so that it replaces any previous pixel color. Next we will draw the parts of the two pictures that overlap using an alpha value of 0.5 so that it uses half the color from the image being drawn and half the color from the current image. Finally we will draw the part of the second picture that is after the overlapping area using an alpha value of 1.0 (opaque).

**Program 63: Overlap Pictures Using AlphaComposite**

```java
/**
 * Method to overlap one picture with another horizontally on top of the current picture. First
 * the part of the first picture before the overlap will be displayed, next to that will be the overlapping
 * region up to the end of the first picture, after that is the remainder of the second picture
 * @param p1 the first picture to display
 * @param p2 the second picture to display
 * @param startOverlap the x position where the overlap begins
 */
public void overlapPictures(Picture p1,
                             Picture p2,
                             int startOverlap)
{
    int amountOverlap = p1.getWidth() - startOverlap;
    // get the Graphics2D object
    Graphics g = this.getGraphics();
    Graphics2D g2 = (Graphics2D) g;
    // draw p1 up to overlap point
    g2.setComposite(
        AlphaComposite.getInstance(AlphaComposite.SRC_OVER,
                                 (float) 1.0f));
    g2.drawImage(p1.getImage(),
                 0,0,startOverlap ,p1.getHeight(),
                 0,0,startOverlap ,p1.getHeight(),
                 null);
    // draw p1 in the overlap area (replace background)
```
g2.drawImage(p1.getImage(),
    startOverlap,0,p1.getWidth(),p1.getHeight(),
    startOverlap,0,p1.getWidth(),p1.getHeight(),null);

    // set the composite to blend the old and new pixels
    // 50%
g2.setComposite(AlphaComposite.getInstance(AlphaComposite.SRC_OVER,
        0.5f));

    g2.drawImage(p2.getImage(),
        startOverlap,0,p1.getWidth(),p2.getHeight(),
        0,0,amountOverlap,p2.getHeight(),null);

    // draw p2 after the overlap
    g2.setComposite(AlphaComposite.getInstance(AlphaComposite.SRC_OVER,
        (float)1.0f));

    g2.drawImage(p2.getImage(),
        p1.getWidth(),0,p2.getWidth()+startOverlap,
        p2.getHeight(),amountOverlap,0,p2.getWidth(),
        p2.getHeight(),null);
}

To use this program to overlap two pictures do:

> Picture p1 = new Picture(Chooser.getMediaPath("KatieFancy.jpg"));
> Picture p2 = new Picture(Chooser.getMediaPath("JenParty.jpg"));
> Picture p3 = new Picture(Chooser.getMediaPath("640x480.jpg"));
> p3.overlapPictures(p1,p2,150);
> p3.show();

You can experiment with the alpha amount that you use in the method getInstance. If you specify 0.5 then half of the source color values will be combined with half of the target color values. If you specify 1.0 then the source color will replace the target color. If you use 0.25 then only 0.25 of the source color will be combined with 0.75 of the target color.

7.3.8 Clipping

You can create one shape and then use that shape as a stencil to limit what is shown when you draw other shapes or images. Only the area inside of the stencil will be seen. This is called clipping.

Let’s create a stencil from an ellipse and then draw the beach using that stencil to clip the image of the beach. The only part of the beach that will be
FIGURE 7.18: Two pictures with a horizontal overlap

visible is the part inside the ellipse (Figure 7.19).

Program 64: Clip an Image to an Ellipse

```java
/**
 * Method to clip the picture to an ellipse
 * @return a new picture with the image clipped to an ellipse
 */
public Picture clipToEllipse()
{
    int width = this.getWidth();
    int height = this.getHeight();
    Picture result = new Picture(width, height);

    // get the graphics2D object for this picture
    Graphics g = result.getGraphics();
    Graphics2D g2 = (Graphics2D) g;

    // create an ellipse to use for clipping
```
Ellipse2D.Double ellipse =
    new Ellipse2D.Double(0,0,width,height);

    // use the ellipse for clipping
    g2.setClip(ellipse);

    // draw the image
    g2.drawImage(this.getImage(),0,0,width,height,null);

    // return the result
    return result;
}

To use this program try the following:

> Picture p = new Picture(FileChooser.getMediaPath("beach.jpg"));
> Picture p2 = p.clipToEllipse();
> p2.show();

FIGURE 7.19: Clip picture using an ellipse

7.4 CONCEPT SUMMARY

In this chapter we discussed packages, using predefined Java classes, inheritance, and interfaces.

7.4.1 Packages

A package is a group of related classes. Classes that are basic to the Java language are in java.lang like the System class. Classes that are used for graphics are in java.awt like the Color class. Classes that are used for input and output are in java.io like the File class.
Chapter 7 Drawing

The full name for a class is `packageName.ClassName`. You don’t have to use the full name for any class in the package java.lang. You don’t have to use the full name for classes in packages other than java.lang if you use a import statement. You can import a class by using the keyword `import` and then the full name of the class.

```java
> import java.awt.Color;
```

You can also import all classes in a package.

```java
> import java.awt.*;
```

When you use classes in packages other than java.lang in methods you will either need to use the full class name or add import statements before the class definition.

```java
import java.awt.*;
import java.awt.font.*;
import java.awt.geom.*;
import java.text.*;
```

```java
/**
 * A class that represents a picture. This class inherits from 
 * SimplePicture and allows the student to add functionality to 
 * the Picture class.
 *
 * @author Barbara Ericson ericson@cc.gatech.edu
 */
public class Picture extends SimplePicture
```

7.4.2 Predefined Java Classes

There are a large number of predefined classes in Java. The API contains the documentation for all the classes in a version of Java. The documentation tells you important information about each class such as what methods it has. The documentation is organized by packages. There are classes for doing graphical user interfaces (in packages javax.swing and java.awt). There are classes for doing input and output (in package java.io). There are classes for doing work with databases (in package java.sql). There are classes for doing work with sound (in package javax.sound). There are classes for working with lists, sets, and maps (java.util).

It is easier and faster to build on what exists rather than write your own classes. Even if what you need isn’t part of Java someone else may have written something similar.

7.4.3 Inheritance

When one class inherits from another it gets all the data (fields) and behavior (methods) from that class. This means that if `Graphics2D` inherits from `Graphics` it understands the same public messages as `Graphics`. The API for a class shows what class it inherits from as well as all methods that are inherited and what class they are inherited from.
7.4.4 Interfaces

A Java class can inherit from only one class but it can implement several interfaces. An interface defines how classes will communicate without worrying about what types they actually are. An object of a class that implements an interface can be declared with the interface name as the type. So if a GradientPaint object implements the Paint interface it can be passed as a parameter to methods that take objects of the type Paint. The API for a class shows what interfaces it implements.

OBJECTS AND METHODS SUMMARY

In this chapter we have talked about several kinds of objects.

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Package</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AffineTransform</td>
<td>java.awt.geom</td>
<td>An object that knows how to do transformations such as rotation, scaling, and shearing.</td>
</tr>
<tr>
<td>AlphaComposite</td>
<td>java.awt</td>
<td>An object that knows how to combine overlapping pixels.</td>
</tr>
<tr>
<td>BasicStroke</td>
<td>java.awt</td>
<td>An object that knows how to draw the outlines of shapes in different widths, with different kinds of joins between lines, and possibly using dashes.</td>
</tr>
<tr>
<td>Ellipse2D.Double</td>
<td>java.awt.geom</td>
<td>An object that represents an ellipse with coordinates that can be of the type double.</td>
</tr>
<tr>
<td>FontRenderContext</td>
<td>java.awt.font</td>
<td>An object that is used for getting accurate text rendering information.</td>
</tr>
<tr>
<td>Graphics</td>
<td>java.awt</td>
<td>An object that knows how to draw or fill simple shapes.</td>
</tr>
<tr>
<td>Graphics2D</td>
<td>java.awt</td>
<td>An object that can handle more complicated drawing.</td>
</tr>
<tr>
<td>Line2D.Double</td>
<td>java.awt.geom</td>
<td>An object that represents a line with coordinates that can be of the type double.</td>
</tr>
</tbody>
</table>

**Picture** methods used or introduced in this chapter:

- `getGraphics()` Returns a `Graphics` object that can be used to draw on the current picture.
- `getImage()` Returns an `Image` object that can be used by `drawImage` methods in `Graphics` or `Graphics2D`.

**Graphics** methods introduced in this chapter:
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>drawArc(int x1, int y1, int w, int h, int startAngle, int arcAngle)</code></td>
<td>Draws the outline of an arc which is part of an oval that fits in an enclosing rectangle with an upper left corner at ((x1, y1)), a width of (w), and a height of (h). The arc will start at the given (startAngle) and end at (startAngle + arcAngle).</td>
</tr>
<tr>
<td><code>drawImage(Image image, int x, int y, ImageObserver observer)</code></td>
<td>Draws the passed image with the top left corner at ((x, y)). The observer is the object to be notified as more of the image is drawn.</td>
</tr>
<tr>
<td><code>drawImage(Image image, int dx1, int dy1, int dx2, int dy2, int sx1, int sy1, int sx2, int sy2, ImageObserver observer)</code></td>
<td>Draws the area from the passed image that is inside the rectangle defined by ((sx1, sy1)) and ((sx2, sy2)) into the rectangle defined by ((dx1, dy1)) and ((dx2, dy2)) in the destination. The observer is the object to be notified as more of the image is drawn.</td>
</tr>
<tr>
<td><code>drawLine(int x1, int y1, int x2, int y2)</code></td>
<td>Draws a line from position ((x1, y1)) to ((x2, y2)).</td>
</tr>
<tr>
<td><code>drawOval(int x1, int y1, int w, int h)</code></td>
<td>Draws the outline of an oval that fits in an enclosing rectangle with an upper left corner at ((x1, y1)), a width of (w), and a height of (h).</td>
</tr>
<tr>
<td><code>drawPolygon(int[] xArray, int[] yArray, int numPoints)</code></td>
<td>Draws the outline of a closed polygon with the points of the polygon given in the (x) and (y) arrays.</td>
</tr>
<tr>
<td><code>drawRect(int x1, int y1, int w, int h)</code></td>
<td>Draws the outline of a rectangle with the upper left corner at ((x1, y1)), a width of (w), and a height of (h).</td>
</tr>
<tr>
<td><code>fillArc(int x1, int y1, int w, int h, int startAngle, int arcAngle)</code></td>
<td>Draws a filled arc which is part of an oval that fits in an enclosing rectangle with an upper left corner at ((x1, y1)), a width of (w), and a height of (h). The arc will start at the given (startAngle) and end at (startAngle + arcAngle).</td>
</tr>
<tr>
<td><code>fillOval(int x1, int y1, int w, int h)</code></td>
<td>Draws a filled oval that fits in an enclosing rectangle with an upper left corner at ((x1, y1)), a width of (w), and a height of (h).</td>
</tr>
<tr>
<td><code>fillPolygon(int[] xArray, int[] yArray, int numPoints)</code></td>
<td>Draws a filled closed polygon with the points of the polygon given in the (x) and (y) arrays.</td>
</tr>
<tr>
<td><code>fillRect(int x1, int y1, int w, int h)</code></td>
<td>Draws a rectangle filled with the current color with an upper left corner at ((x1, y1)), a width of (w), and a height of (h).</td>
</tr>
<tr>
<td><code>setClip(Shape clip)</code></td>
<td>Sets the shape to use as a stencil to limit what is shown.</td>
</tr>
<tr>
<td><code>setColor(Color colorObj)</code></td>
<td>Sets the color to draw with.</td>
</tr>
<tr>
<td><code>setFont(Font fontObj)</code></td>
<td>Sets the font to use for drawing strings.</td>
</tr>
</tbody>
</table>

**Graphics2D** methods introduced in this chapter:
Section 7.4  Concept Summary  261

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>draw(Object obj)</td>
<td>Draw the outline of the passed object.</td>
</tr>
<tr>
<td>drawImage(Image image, AffineTransform xform, ImageObserver observer)</td>
<td>Draws the passed Image object after applying the passed AffineTransform object to it and notifies the ImageObserver as more of the rendered image is available.</td>
</tr>
<tr>
<td>getFontRenderContext()</td>
<td>Returns a FontRenderContext object that contains rendering hints and device information such as the dots-per-inch.</td>
</tr>
<tr>
<td>getTransform()</td>
<td>Returns the current AffineTransform object that represents all current transformations (rotations, translations, scaling, etc).</td>
</tr>
<tr>
<td>fill(Object obj)</td>
<td>Draw the passed object filled with the current paint type.</td>
</tr>
<tr>
<td>setComposite(Composite comp)</td>
<td>Set the Composite object that specifies how to handle overlapping pixels.</td>
</tr>
<tr>
<td>setPaint(Paint paint)</td>
<td>Sets the color, gradient, or texture to using when painting.</td>
</tr>
<tr>
<td>setStroke(Stroke s)</td>
<td>Set the brush (pen) to the one defined by the passed Stroke object.</td>
</tr>
<tr>
<td>setTransform(AffineTransform transform)</td>
<td>Sets the current transformation object to the passed AffineTransform object.</td>
</tr>
</tbody>
</table>

PROBLEMS

7.1. Using the drawing tools presented here, draw a house—just go for the simple child’s house with one door, two windows, walls, and a roof.

7.2. Put a cabana on that beach. Draw the house from the previous exercise on the beach where we put the mysterious box previously.

7.3. Now use your house to draw a town with dozens of houses at different sizes. You’ll probably want to modify your house function to draw at an input coordinate, then change the coordinate where each house is drawn.

7.4. Draw a rainbow—use what you know about colors, pixels, and drawing operations to draw a rainbow. Is this easier to do with our drawing functions or by manipulating individual pixels? Why?

7.5. Modify the method drawFace to take the width and height of the desired face and calculate the positions based on the desired width and height.

7.6. Draw a string on a picture at the top of the picture and centered horizontally. You will need to use the FontMetrics class to get the height of the string in pixels in order to determine where the baseline should be so that the string is visible. You should also subtract the descent from the height.

7.7. Create another method that takes the text, x, y, font and color to use when you draw a string. Rewrite the old drawString() method to call this new method.

7.8. Create a method that draws an X across the current picture using dashed lines.

7.9. Write a general rotate method that takes the degrees to rotate the current Picture object and returns a new Picture object. Use the AffineTransform.rotate(double radians) method to do this. You will need to use the general Math.toRadians(int degree) method to translate the degrees to radians.

7.10. Write a method to draw a text string rotated 90 degrees to the right down the right side of the picture.
262  Chapter 7  Drawing

7.11. Write a method to overlap two pictures vertically using an AlphaComposite object.

7.12. Write a method to clip an image to a triangle or star shape.
PART THREE

SOUNDS

Chapter 8  Modifying all Samples in a Sound
Chapter 9  Modifying Samples using Ranges
Chapter 10 Combining and Creating Sounds
CHAPTER 8

Modifying all Samples in a Sound

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

• To understand how we digitize sounds, and what the limitations are of human hearing that allow us to digitize sounds.

• To use the Nyquist theorem to determine the sampling rate necessary for digitizing a desired sound.

• To manipulate the volume of a sound.

• To create (and avoid) clipping.

The computer science goals for this chapter are:

• To understand and use a one-dimensional array as a data structure.

• To use the formula that \( n \) bits result in \( 2^n \) possible patterns in order to figure out the number of bits needed to save values.

• To debug sound programs (methods).

• To use iteration (with `while` and `for` loops) for manipulating sounds.

• To use conditionals when manipulating sounds.

8.1 HOW SOUND IS ENCODED

There are two parts to understanding how sound is encoded and manipulated.

• First, what are the physics of sound? How is it that we hear a variety of sounds?

• Next, how can we then map these sounds into the numbers of a computer?
8.1.1 The Physics of Sound

Physically, sounds are waves of air pressure. When something makes a sound, it makes ripples in the air just like stones or raindrops dropped into a pond cause ripples in the surface of the water (Figure 8.1). Each drop causes a wave of pressure to pass over the surface of the water, which causes visible rises in the water, and less visible but just as large depressions in the water. The rises are increases in pressure and the lows are decreases in pressure. Some of the ripples we see are actually ones that arise from combinations of ripples—some waves are the sums and interactions from other waves.

![Figure 8.1: Raindrops causing ripples in the surface of the water, just as sound causes ripples in the air](image)

In the air, we call these increases in pressure *compressions* and decreases in pressure *rarefactions*. It’s these compressions and rarefactions that lead to our hearing. The shape of the waves, their *frequency*, and their *amplitude* all impact how we perceive the sound.

The simplest sound in the world is a *sine wave* (Figure 10.5). In a sine wave, the compressions and rarefactions arrive with equal size and regularity. In a sine wave, one compression plus one rarefaction is called a *cycle*. At some point in the cycle, there has to be a point where there is zero pressure, just between the compression and the rarefaction. The distance from the zero point to the greatest pressure (or least pressure) is called the *amplitude*.

Formally, amplitude is measured in *Newtons per meter-squared* (N/m²). That’s a rather hard unit to understand in terms of perception, but you can get a sense of the amazing range of human hearing from this unit. The smallest sound that humans typically hear is 0.0002 N/m², and the point at which we sense the vibrations in our entire body is 200 N/m²! In general, amplitude is the most important factor in our perception of *volume*: If the amplitude rises, we perceive the sound as being louder typically. Other factors like air pressure factor into our perception of increased volume, too. Ever noticed how sounds sound differently on very humid days as compared with very dry days?

When we perceive an increase in volume, we say that we’re perceiving an increase in the *intensity* of sound. Intensity is measured in watts per meter-squared (W/m²). (Yes, those are watts just like the ones you’re referring to when you get a 60-watt light bulb—it’s a measure of power.) The intensity is proportional to
Human perception of sound is not a direct mapping from the physical reality. The study of the human perception of sound is called \textit{psychoacoustics}. One of the odd facts about psychoacoustics is that most of our perception of sound is \textit{logarithmically} related to the actual phenomena. Intensity is an example of this. A change in intensity from $0.1 W/m^2$ to $0.01 W/m^2$ sounds the \textit{same} to us (as in the same amount of volume change) as a change in intensity of $0.001 W/m^2$ to $0.0001 W/m^2$.

We measure the change in intensity in \textit{decibels} (dB). That’s probably the unit that you most often associate with volume. A decibel is a logarithmic measure, so it does match the way we perceive volume. It’s always a ratio, a comparison of two values. $10 \times \log_{10}(I_1/I_2)$ is change in intensity in decibels between $I_1$ and $I_2$. If two amplitudes are measured under the same conditions, we can express the same definition as amplitudes: $20 \times \log_{10}(A_1/A_2)$. If $A_2 = 2 \times A_1$ (i.e., the amplitude doubles), the difference is roughly 6dB.

When decibel is used as an absolute measurement, it’s in reference to the threshold of audibility at \textit{sound pressure level} (SPL): 0 dB SPL. Normal speech has intensity about 60 dB SPL. Shouted speech is about 80 dB SPL.

How often a cycle occurs is called the \textit{frequency}. If a cycle is short, then there can be lots of them per second. If a cycle is long, then there are fewer of them. As the frequency increases we perceive the \textit{pitch} to increase. We measure frequency in \textit{cycles per second} (cps) or \textit{Hertz} (Hz).

All sounds are periodic: there is always some pattern of rarefaction and compression that leads to cycles. In a sine wave, the notion of a cycle is easy. In natural waves, it’s not so clear where a pattern repeats. Even in the ripples in a pond, the waves aren’t as regular as you might think. The time between peaks in waves isn’t always the same: it varies. This means that a cycle may involve several peaks-and-valleys until it repeats.

Humans hear between 2 Hz and 20,000 Hz (or 20 kilohertz, abbreviated 20 kHz). The sound of a sine wave is produced when a change in intensity is matched to the square of the amplitude. For example, if the amplitude doubles, intensity quadruples.
kHz). Again, as with amplitudes, that’s an enormous range! To give you a sense of where music fits into that spectrum, the note A above middle C is 440 Hz in traditional, equal temperament tuning (Figure 8.3).

FIGURE 8.3: The note A above middle C is 440 Hz

Like intensity, our perception of pitch is almost exactly proportional to the log of the frequency. We really don’t perceive absolute differences in pitch, but the ratio of the frequencies. If you heard a 100 Hz sound followed by a 200 Hz sound, you’d perceive the same pitch change (or pitch interval) as a shift from 1000 Hz to 2000 Hz. Obviously, a difference of 100 Hz is a lot smaller than a change of 1000 Hz, but we perceive it to be the same.

In standard tuning, the ratio in frequency between the same notes in adjacent octaves is 2:1. Frequency doubles each octave. We told you earlier that A above middle C is 440 Hz. You know then that the next A up the scale is 880 Hz.

How we think about music is dependent upon our cultural standards, but there are some universals. Among these universals are the use of pitch intervals (e.g., the ratio between notes C and D remains the same in every octave), the relationship between octaves remains constant, and the existence of four to seven main pitches (not considering sharps and flats here) in an octave.

What makes the experience of one sound different from another? Why is it that a flute playing a note sounds so different than a trumpet or a clarinet playing the same note? We still don’t understand everything about psychoacoustics and what physical properties influence our perception of sound, but here are some of the factors that lead us to perceiving different sounds (especially musical instruments) as distinct.

- Real sounds are almost never single frequency sound waves. Most natural sounds have several frequencies in them, often at different amplitudes. These additional frequencies are sometimes called overtones. When a piano plays the note C, for example, part of the richness of the tone is that the notes E and G are also in the sound, but at lower amplitudes. Different instruments have different overtones in their notes. The central tone, the one we’re trying to play, is called the fundamental.

- Instrument sounds are not continuous with respect to amplitude and frequency. Some come slowly up to the target frequency and amplitude (like
wind instruments), while others hit the frequency and amplitude very quickly and then the volume fades while the frequency remains pretty constant (like a piano).

- Not all sound waves are represented well by sine waves. Real sounds have funny bumps and sharp edges. Our ears can pick these up, at least in the first few waves. We can do a reasonable job synthesizing with sine waves, but synthesizers sometimes also use other kinds of wave forms to get different kinds of sounds (Figure 8.4).

![Figure 8.4: Some synthesizers using triangular (or sawtooth) or square waves.](image)

### 8.1.2 Exploring Sounds

On your CD, you will find the MediaTools application with documentation for how to get it started. The MediaTools application contains tools for sound, graphics, and video. Using the sound tools, you can actually observe sounds as they’re coming into your computer’s microphone to get a sense of what louder and softer sounds look like, and what higher and lower pitched sounds look like.

The basic sound editor looks like Figure 8.5. You can record sounds, open WAV files on your disk, and view the sounds in a variety of ways. (Of course, assuming that you have a microphone on your computer!)

![Figure 8.5: Sound editor main tool](image)

To view sounds, click the View button, then the Record button. (Hit the Stop button to stop recording.) There are three kinds of views that you can make of the sound.

The first is the signal view (Figure 8.6). In the signal view, you’re looking at the sound raw—each increase in air pressure results in an rise in the graph, and each decrease in sound pressure results in a drop in the graph. Note how rapidly...
the wave changes! Try some softer and louder sounds so that you can see how the sounds’ look changes. You can always get back to the signal view from another view by clicking the Signal button.

![Viewing the sound signal as it comes in](image)

**FIGURE 8.6:** Viewing the sound signal as it comes in

The second view is the *spectrum view* (Figure 8.7). The spectrum view is a completely different perspective on the sound. In the previous section, you read that natural sounds are often actually composed of several different frequencies at once. The spectrum view shows these individual frequencies. This view is also called the *frequency domain*.

Frequencies increase in the spectrum view from left to right. The height of a column indicates the amount of energy (roughly, the volume) of that frequency in the sound. Natural sounds look like Figure 8.8 with more than one *spike* (rise in the graph). (The smaller rises around a spike are often seen as *noise*.)

The technical term for how a spectrum view is generated is called a *Fourier transform*. A Fourier transform takes the sound from the *time domain* (rises and falls in the sound over time) into the frequency domain (identifying which frequencies are in a sound, and the energy of those frequencies, over time). The specific technique being used in the MediaTools signal view is a *Fast Fourier Transform* (or *FFT*), a very common way to do Fourier transforms quickly on a computer so that we can get a real time view of the changing spectra.

The third view is the *sonogram view* (Figure 8.9). The sonogram view is very much like the spectrum view in that it’s describing the frequency domain, but it presents these frequencies over time. Each column in the sonogram view, sometimes called a *slice* or *window (of time)*, represents all the frequencies at a given moment in time. The frequencies increase in the slice from lower (bottom) to higher (top). The darkness of the spot in the column indicates the amount of energy of that frequency in the input sound at the given moment. The sonogram view is great for studying how sounds change over time, e.g., how the sound of a piano key being struck changes as the note fades, or how different instruments differ in their sounds, or in how different vocal sounds differ.
8.1.3 Encoding Sounds

You just read about how sounds work physically and how we perceive them. To manipulate these sounds on a computer and to play them back on a computer, we have to digitize them. To digitize sound means to take this flow of waves and turn it into numbers. We want to be able to capture a sound, perhaps manipulate it, and then play it back (through the computer’s speakers) and hear what we captured—as exactly as possible.

The first part of the process of digitizing a sound is handled by the computer’s hardware—the physical machinery of the computer. If a computer has a microphone and the appropriate sound equipment (like a SoundBlaster sound card on Wintel computers), then it’s possible, at any moment, to measure the amount of air pressure against that microphone as a single number. Positive numbers correspond to rises in pressure, and negative numbers correspond to rarefactions. We call this an analog-to-digital conversion (ADC)—we’ve moved from an analog signal (a continuously changing sound wave) to a digital value. This means that we can get an instantaneous measure of the sound pressure, but it’s only one step along the way. Sound is a continuous changing pressure wave. How do we store that in our computer?

By the way, playback systems on computers work essentially the same in reverse. The sound hardware does a digital-to-analog conversion (DAC), and the analog signal is then sent to the speakers. The DAC process also requires numbers
Section 8.1  How Sound is Encoded

FIGURE 8.8: Viewing a sound in spectrum view with multiple “spikes” representing pressure.

If you’ve had some calculus, you’ve got some idea of how we might do that. You know that we can get close to measuring the area under a curve with more and more rectangles whose height matches the curve (Figure 8.10). With that idea, it’s pretty clear that if we capture enough of those microphone pressure readings, we capture the wave. We call each of those pressure readings a sample—we are literally “sampling” the sound at that moment. But how many samples do we need? In integral calculus, you compute the area under the curve by (conceptually) having an infinite number of rectangles. While computer memories are growing larger and larger all the time, we still can’t capture an infinite number of samples per sound.

Mathematicians and physicists wondered about these kinds of questions long before there were computers, and the answer to how many samples we need was actually computed long ago. The answer depends on the highest frequency you want to capture. Let’s say that you don’t care about any sounds higher than 8,000 Hz. The Nyquist theorem says that we would need to capture 16,000 samples per second to completely capture and define a wave whose frequency is less than 8,000 cycles per second.

Computer Science Idea: Nyquist Theorem
To capture a sound of at most \( n \) cycles per second, you need to capture \( 2n \) samples per second.

This isn’t just a theoretical result. The Nyquist theorem influences applications in our daily life. It turns out that human voices don’t typically get over 4,000 Hz. That’s why our telephone system is designed around capturing 8,000 samples per second. That’s why playing music through the telephone doesn’t really work...
very well. The limits of (most) human hearing is around 22,000 Hz. If we were to capture 44,000 samples per second, we would be able to capture any sound that we could actually hear. CD’s are created by capturing sound at 44,100 samples per second—just a little bit more than 44 kHz for technical reasons and for a fudge factor.

We call the rate at which samples are collected the **sampling rate**. Most sounds that we hear in daily life are well within the range of the limits of our hearing. You can capture and manipulate sounds in this class at a sampling rate of 22 kHz (22,000 samples per second), and it will sound quite reasonable. If you use too low of a sampling rate to capture a high-pitched sound, you’ll still hear something when you play the sound back, but the pitch will sound strange.

Typically, each of these samples are encoded in two bytes (16 bits). Though there are larger *sample sizes*, 16 bits works perfectly well for most applications. CD-quality sound uses 16 bit samples.
Section 8.1 How Sound is Encoded

In 16 bits, the numbers that can be encoded range from -32,768 to 32,767. These aren’t magic numbers—they make perfect sense when you understand the encoding. These numbers are encoded in 16 bits using a technique called two’s complement notation, but we can understand it without knowing the details of that technique. We’ve got 16 bits to represent positive and negative numbers. Let’s set aside one of those bits (remember, it’s just 0 or 1) to represent whether we’re talking about a positive (0) or negative (1) number. We call that the sign bit. That leaves 15 bits to represent the actual value. How many different patterns of 15 bits are there? We could start counting:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000000000000</td>
<td>-32,768</td>
</tr>
<tr>
<td>000000000000001</td>
<td>-32,767</td>
</tr>
<tr>
<td>000000000000010</td>
<td>-32,766</td>
</tr>
<tr>
<td>000000000000011</td>
<td>-32,765</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>111111111111110</td>
<td>32,758</td>
</tr>
<tr>
<td>111111111111111</td>
<td>32,759</td>
</tr>
</tbody>
</table>

That looks forbidding. Let’s see if we can figure out a pattern. If we’ve got two bits, there are four patterns: 00, 01, 10, 11. If we’ve got three bits, there are eight patterns: 000, 001, 010, 011, 100, 101, 110, 111. It turns out that $2^2$ is four, and $2^3$ is eight. Play with four bits. How many patterns are there? $2^4 = 16$ It turns out that we can state this as a general principle.

Computer Science Idea: $2^n$ patterns in $n$ bits

If you have $n$ bits, there are $2^n$ possible patterns in those $n$ bits.

$2^{15} = 32,768$. Why is there one more value in the negative range than the positive? Zero is neither negative nor positive, but if we want to represent it as bits, we need to define some pattern as zero. We use one of the positive range values (where the sign bit is zero) to represent zero, so that takes up one of the 32,768 patterns.

The sample size is a limitation on the amplitude of the sound that can be captured. If you have a sound that generates a pressure greater than 32,767 (or a rarefaction greater than -32,768), you’ll only capture up to the limits of the 16 bits. If you were to look at the wave in the signal view, it would look like somebody took some scissors and clipped off the peaks of the waves. We call that effect clipping for that very reason. If you play (or generate) a sound that’s clipped, it sounds bad—it sounds like your speakers are breaking.

There are other ways of digitizing sound, but this is by far the most common. The technical term for this way of encoding sound is pulse coded modulation (PCM). You may encounter that term if you read further in audio or play with audio software.

What this means is that a sound in a computer is a long list of numbers, each of which is a sample in time. There is an ordering in these samples: If you played the samples out of order, you wouldn’t get the same sound at all. The most efficient
way to store an ordered list of data items on a computer is with an array. An array is literally a sequence of bytes right next to one another in memory. We call each value in an array an element.

We can easily store the samples that make up a sound in an array. Think of each two bytes as storing a single sample. The array will be large—for CD-quality sounds, there will be 44,100 elements for every second of recording. A minute long recording will result in an array with 26,460,000 elements.

Each array element has a number associated with it, called its index. The index numbers start at 0 and increase sequentially. The first one is 0, the second one is 1, and so on. It may sound strange to say the the index for the first array element is 0 but this is basically a measure of the distance from the first element in the array. Since the distance from the first element to itself is 0, the index is 0. You can think about an array as a long line of boxes, each one holding a value and each box having an index number on it (Figure 8.11).

![Array Example](image)

FIGURE 8.11: A depiction of the first five elements in a real sound array

Using the MediaTools, you can graph a sound file (Figure 8.12) and get a sense of where the sound is quiet (small amplitudes), and loud (large amplitudes). This is actually important if you want to manipulate the sound. For example, the gaps between recorded words tend to be quiet—at least quieter than the words themselves. You can pick out where words end by looking for these gaps, as in Figure 8.12.

![Sound Recording](image)

FIGURE 8.12: A sound recording graphed in the MediaTools

You will soon read about how to read a file containing a recording of a sound into a sound object, view the samples in that sound, and change the values of the sound array elements. By changing the values in the array, you change the sound. Manipulating a sound is simply a matter of manipulating elements in an array.

### 8.2 MANIPULATING SOUNDS

Now that we know how sounds are encoded, we can manipulate sounds using Java programs. Here’s what we’ll need to do.
Section 8.2 Manipulating Sounds

1. We’ll need to get a filename of a WAV file, and make a Sound object from it. You already saw how to do that in a previous chapter.

2. You will often get the samples of the sound as an array. Sample objects are easy to manipulate, and they know that when you change them, they should automatically change the original sound. You’ll read first about manipulating the samples to start with, then about how to manipulate the sound samples from within the sound itself.

3. Whether you get the sample objects out of a sound, or just deal with the samples in the sound object, you will then want to do something to the value at the sample.

4. You may then want to write the sound back out to a new file, to use elsewhere. (Most sound editing programs know how to deal with audio.)

8.2.1 Opening Sounds and Manipulating Samples

You have already seen how to pick a file with FileChooser.pickAFile() and then make a Sound object (object of the class Sound) with new Sound(fileName). Here’s an example of doing that in DrJava.

```java
String fileName = FileChooser.pickAFile();
Sound sound1 = new Sound(fileName);
System.out.println(sound1);
```

What new Sound(fileName) does is to scoop up all the bytes from the file name provided as input, dump them into memory, and place a big sign on them saying, “This is a sound object (an object of the class Sound)!” When you execute `Sound sound1 = new Sound(fileName)`, you are saying, “Create an object variable called sound1 that refers to the Sound object created from the information in the file with a file name given by the variable fileName.” When you use sound1 as input to functions, you are saying “Use that sound object over there (yeah, the one referred to by the variable sound1) as input to this function.”

You can get the samples from a sound using getSamples(). The method getSamples() must be invoked on a Sound object and returns an array of all the samples as SoundSample objects (objects of the class SoundSample). When you execute this function, it may take quite a while before it finishes—longer for longer sounds, shorter for shorter sounds.

```java
SoundSample[] sampleArray = sound1.getSamples();
System.out.println(sampleArray.length);
```

The method getSamples() is making an array of SoundSample objects out of the samples in the Sound object. An object is more than just a simple value—for example, a SoundSample object knows what Sound object it came from and what its index is. You will read more about objects later, but take it at face value now that getSamples() provides you with a bunch of sample objects that you can manipulate—and in fact, makes manipulation pretty easy. You can get the
value of a `SoundSample` object by using `getValue()`, and you set the value of a `SoundSample` object with `setValue(value)`.

But before we get to the manipulations, let’s look at some other ways to get and set samples. We can ask the sound to give us the value of a specific sample at a specific index, by using the method `getSampleValueAt(index)` on a `Sound` object.

```java
> System.out.println(sound1.getSampleValueAt(0));
36
> System.out.println(sound1.getSampleValueAt(1));
29
```

What numbers can we use as index values? Anything between 0 and the number of samples minus 1. We can get the number of samples using `getLength()`.

Notice the error that we get below if we try to get a sample past the end of the array.

```java
> System.out.println(sound1.getLength());
421110
> sound1.getSampleValueAt(500000);
You are trying to access the sample at index: 500000, but the last valid index is at 421109
```

We can similarly change sample values in a `Sound` object with `setSampleValueAt(index)`. This method changes the value of the sample at the passed index. We can then check it again with `getSampleValueAt()`.

```java
> System.out.println(sound1.getSampleValueAt(0));
36
> sound1.setSampleValueAt(0,12);
> System.out.println(sound1.getSampleValueAt(0));
12
```

**Common Bug: Mistyping a name**

You just saw a whole bunch of method names, and some of them are pretty long. What happens if you type one of them wrong? DrJava will complain that it doesn’t know what you mean, like this:

```java
> sound1.setSampleVAlueAt(0,12);
Error: No 'setSampleVAlueAt' method in 'Sound'
```

It’s no big deal. Use the up arrow on the keyboard to bring up the last thing you typed into DrJava and then use the left arrow to get to the place with the error and then fix it.

You can use the up arrow to get to any of the commands you have typed in the interactions pane since you started DrJava.

What do you think would happen if we then played this sound? Would it really sound different than it did before, now that we’ve turned the first sample from the
number 36 to the number 12? Not really. To explain why not, let’s find out what
the sampling rate is for this sound, by using the method `getSamplingRate()`.

```java
> Sound aSound = new Sound(FileChooser.getMediaPath("preamble.wav"));
> System.out.println(aSound.getSamplingRate());
22050.0
```

To make some of our manipulations easier, we’re going to be using
`FileChooser.setMediaPath(String directory)`
and
`FileChooser.getMediaPath(String baseFileName)`

Using `setMediaPath(String directory)` will set a media directory (folder),
and then `getMediaPath(String baseFileName)` will reference media files within
that directory. This makes it much easier to reference media files—you don’t have
to spell out the whole path. The method `getMediaPath` takes a base file name as an
argument, and will return the directory set by `setMediaPath` with the passed file
name added to the end of the directory name. The default for the media directory is
c:/intro-prog-java/mediasources/. If you have your media in another directory
you will need to use `setMediaPath` before you can use `getMediaPath`.

```java
> FileChooser.setMediaPath("c:/intro-prog-java/mediasources/");
The media directory is now c:/intro-prog-java/mediasources/
> System.out.println(FileChooser.getMediaPath("barbara.jpg"));
c:/intro-prog-java/mediasources/barbara.jpg
> System.out.println(FileChooser.getMediaPath("croak.wav"));
c:/intro-prog-java/mediasources/croak.wav
```

**Common Bug: It’s not a file, it’s a string**

Just because `getMediaPath` returns something that looks
like a path doesn’t mean that a file really exists there.
You have to know the right base name, but if you do, it’s
easier to use in your code. But if you put in a non-existent
file, you’ll get a path to a non-existent file. The method
`getMediaPath` will warn you if the file doesn’t exist.

```java
> FileChooser.getMediaPath("blahblah.wav");
There is no file named blahblah.wav in directory
c:/intro-prog-java/mediasources/
```

The sound that we’re manipulating in this example (a recording of Mark
reading part of the U.S. Constitution’s preamble) has a sampling rate of 22,050
samples per second. Changing one sample changes $1/22050$ of the first second of
that sound. If you can hear that, you have amazingly good hearing—and I will
have my doubts about your truthfulness!

Obviously, to make a significant manipulation to the sound, we have to ma-
nipulate hundreds if not thousands of samples. We’re certainly not going to do that
by typing thousands of lines like this:
We need to take advantage of the computer executing our program, by telling it to go do something hundreds or thousands of times. That’s the topic for the next section.

But we will end this section by talking about how to write your results back out to a file. Once you’ve manipulated your sound and want to save it out to use elsewhere, you use `write(String fileName)` which takes a filename as input. Be sure that your file ends with the extension “.wav” if you’re saving a sound so that your operating system knows what to do with it (what type of data is in it)!

```java
Sound aSound = new Sound(FileChooser.pickAFile());
System.out.println(aSound.getFileName());
c:\intro-prog-java\mediasources\preamble.wav
aSound.write("c:\\intro-prog-java\\mediasources\\preamble.wav");
```

### Making it Work Tip: Slashes and Backslashes in Java

Even though the file name is output with backslashes (`\`) in it, a Java string that wants to have a backslash in it must double it since the backslash character is used in Java strings to indicate special characters like tab. Or, you can use a forward slash (`/`) instead of double backslashes.

### Common Bug: Saving a file quickly—and how to find it again!

What if you don’t know the whole path to a directory of your choosing? You don’t have to specify anything more than the base name.

```java
aSound.write("new-preamble.wav")
```

The problem is finding the file again! In what directory did it get saved? This is a pretty simple bug to resolve. The default directory (the one you get if you don’t specify a path) is wherever DrJava is. You can also use `FileChooser.getMediaPath("new-preamble.wav")` to get the full name of the media directory and base file name.

You’ll probably figure out when playing sounds a lot that if you use `play()` a couple times in quick succession, you’ll mix the sounds. How do you make sure that the computer plays only a single sound and then waits for that sound to end?
Section 8.2  Manipulating Sounds  279

You use something called `blockingPlay()`. That works the same as `play()`, but it waits for the sound to end so that no other sound can interfere while it’s playing.

8.2.2 Using MediaTools for Looking at Sounds

The MediaTools for manipulating sounds that you read about earlier can also be used to study sound files. Any WAV file on your computer can be opened and studied within the sound tools.

Using the MediaTools Application.

From the basic sound editor tool in the MediaTools application, click on File to get the option to open a WAV file (Figure 8.13). The MediaTools’ open file dialog will then appear. Find a WAV file by clicking on the directories on the left until you find one that contains the WAV files you want on the right (Figure 8.14), then click OK.

Your CD contains a `mediasources` directory on it. Most of the examples in the book use the media in this directory. You’ll probably want to drag the `mediasources` folder onto your hard disk so that you can use it there. The default media directory is `c:/intro-prog-java/mediasources` so that would be a good place to put it. If you put it somewhere else just use `FileChooser.setMediaPath(directory)` to set the directory to use to a different directory.

![FIGURE 8.13: The sound editor open menu in MediaTools application](image)

You will then be shown the file in the sound editor view (Figure 8.15). The sound editor lets you explore a sound in many ways (Figure 8.16). As you scroll through the sound and change the sound cursor (the red/blue line in the graph) position, the index changes to show you which sound array element you’re currently looking at, and the value shows you the value at that index. You can also fit the whole sound into the graph to get an overall view (but currently breaks the index/value displays). You can even “play” your recorded sound as if it were an instrument—try pressing the piano keys across the bottom of the editor. You can also set the cursor (via the scrollbar or by dragging in the graph window) and play the sound before the cursor—a good way to hear what part of the sound corresponds to what index positions. Clicking the <> button provides a menu of options which includes getting an FFT view of the sound.
8.2.3 Introducing Loops

The problem of wanting to do something similar a great many times is a common one in computing: How do we get the computer to do something over-and-over again? We need to get the computer to loop or iterate. Java has commands especially for looping (iterating).

Starting with Java version 1.5 there is a new way of looping through all members of an array using a for each loop. The syntax is

```
for (Type variableName : array)
```

You can read this as, "for each element in the array execute the body of the loop." The first time through the loop the `variableName` will refer to the first element of the array (the one at index 0). The second time through the loop the `variableName` will refer to the second element of the array (the one at index 1). The last time through the loop the `variableName` will refer to the last element of the array (the one at index (length - 1)). The code to loop through all the `SoundSample` objects in an array of `SoundSample` objects and set the value of each
FIGURE 8.16: Exploring the sound in the editor in MediaTools application

sample to its original value (no change) is below:

```java
public void doNothing()
{
    SoundSample[] sampleArray = this.getSamples();
    int value = 0;

    // loop through all the samples in the array
    for (SoundSample sample : sampleArray)
    {
        value = sample.getValue();
        sample.setValue(value);
    }
}
```

As you can see the for each loop makes it easy to loop through all the elements of an array. You can use the for each loop whenever you want to process all the elements of an array and you don’t need to know the current index in the body of the loop.

However, not everyone is using Java 1.5 yet. If you are not using Java 1.5 you may want to start with a while loop. A while loop executes some commands while a test returns true. In order for the loop to stop there but be some way for the test to end up false.

The way that we will manipulate a sound is to change the values in the samples that make up the sound. We want to loop through all the samples in a sound and do something to each value. One way to do that is to loop through all the elements of the array of samples. We are going to use the getSamples() function we saw earlier to provide our array.

For example, here is the while loop that simply sets each sample to its own value (a particularly useless exercise, but it’ll get more interesting in just a couple pages).

```java
public void doNothing()
{
    SoundSample[] sampleArray = this.getSamples();
    SoundSample sample = null;
    int index = 0;
    ```
```java
int value = 0;

// loop through all the samples in the array
while (index < sampleArray.length)
{
    sample = sampleArray[index];
    value = sample.getValue();
    sample.setValue(value);
    index++;
}
```

Let’s talk through this code.

- The first statement gets the array of SoundSample objects from the current Sound object using the method `getSamples()` and declares a variable `sampleArray` which refers to it. The “this” means the current Sound object that was implicitly passed to the method. You could leave off the “this” and the compiler would add it.

- The next statement declares a variable `sample` that can refer to a SoundSample object but is set to `null` to show that it doesn’t reference any object yet.

- Next a primitive variable `index` is declared and initialized to 0.

- The primitive variable `value` is declared and initialized to 0.

- The code `while (index < sampleArray.length)` tests if the value of `index` is less than the length of the array of SoundSample objects. If it is, the body of the loop will be executed. If not, execution will continue with the first statement following the body of the loop.

- The first statement in the body of the loop sets the variable `sample` to refer to the SoundSample object at the value of `index` in the array `sampleArray`. Since index starts off with a value of 0 this will refer to the first SoundSample object in the array the first time through the loop.

- The space reserved for the variable `value` is set to the value of the SoundSample object referred to by `sample`.

- Next, the value of the SoundSample object referred to by `sample` is set to the contents of `value`. Since the contents of the variable `value` is the value for this SoundSample object there will be no change. It just sets the sample value to the original value.

- Finally, the value in `index` is incremented by one. Then execution will jump back to the `while` test again.

Here’s the exact same code (it would work exactly the same), but with different variable names.
Section 8.2  Manipulating Sounds  283

SoundSample[] a = this.getSamples();  SoundSample s = null;  int i = 0;  int v = 0;

// loop through all the samples in the array  while (i < a.length)
{
  s = a[i];
  v = s.getValue();
  s.setValue(v);
  i++;
}

What’s the difference? These are slightly easier to confuse variable names. a and s are not as obvious as to what they are naming as sampleArray and sample. Java doesn’t care which we use, and the single character variable names are clearly easier to type. But the longer variable names make it easier to understand your code. It is best to try to make your code easier for humans to understand.

You may have wondered do we need the variable v? We could combine the two statements into one.

SoundSample[] a = this.getSamples();  SoundSample s = null;  int i = 0;

// loop through all the samples in the array  while (i < a.length)
{
  s = a[i];
  s.setValue(s.getValue());
  i++;
}

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.

**Common Bug: Keep sounds short**
Longer sounds take up more memory and will process more slowly.

**Common Bug: Windows and WAV files**
The world of WAV files isn’t as compatible and smooth as one might like. WAV files created with other applications (such as Windows Recorder) may not play in DrJava, and DrJava WAV files may not play in all other applications (e.g., WinAmp 2). Some tools like Apple QuickTime Player Pro (http://www.apple.com/quicktime) are good at reading any WAV file and being able to export a new one that most any other application can read. Some WAV files are encoded using MP3 which means they are really MP3 files. You can convert these using Sound.convert(origFileName, convertedFileName) where origFileName and convertedFileName are the full names (include path information).

### 8.3 CHANGING THE VOLUME OF SOUNDS

Earlier, we said that the amplitude of a sound is the main factor in the volume. This means that if we increase the amplitude, we increase the volume. Or if we decrease the amplitude, we decrease the volume.

Don’t get confused here—changing the amplitude doesn’t reach out and twist up the volume knob on your speakers. If your speaker’s volume (or computer’s volume) is turned down, the sound will never get very loud. The point is getting the sound itself louder. Have you ever watched a movie on TV where, without changing the volume on the TV, sound becomes so low that you can hardly hear it? (Marlon Brando’s dialogue in the movie The Godfather comes to mind.) That’s what we’re doing here. We can make sounds shout or whisper by tweaking the amplitude.

#### 8.3.1 Increasing Volume

Here’s a function that doubles the amplitude of an input sound.

**Program 65: Increase an input sound’s volume**

```java
/**
 * Method to double the volume (amplitude) of the sound
 */
```
Section 8.3 Changing the Volume of Sounds 285

public void increaseVolume()
{
    SoundSample[] sampleArray = this.getSamples();
    SoundSample sample = null;
    int value = 0;
    int index = 0;

    // loop through all the samples in the array
    while (index < sampleArray.length)
    {
        sample = sampleArray[index];
        value = sample.getValue();
        sample.setValue(value * 2);
        index++;
    }
}

Go ahead and type the above into your DrJava definitions pane before the last curly brace in the Sound.java class. Click Compile All to get DrJava to compile it. Follow along the example below to get a better idea of how this all works.

To use this program, you have to create a sound first and invoke this method on it. Don’t forget that you can’t type this code in and have it work as-is: Your path names may be different than what is shown here!

> String f = "c:/intro-prog-java/mediasources/gettysburg10.wav";
> Sound s = new Sound(f);
> s.play();
> s.explore();
> s.increaseVolume();
> s.play();
> s.explore();

In the interactions pane we create a variable f which refers to a String object that holds the name of a file. We create the variable s which refers to a Sound object created from the file using new Sound(f). We ask this Sound object to play using s.play(). We then open an explorer on the sound to see what it looks like graphically using s.explore(). We next increase its volume using s.increaseVolume(). This implicitly passes the Sound object to the method increaseVolume(). So the code this.getSamples() in the method increaseVolume() means to get them
from the implicitly passed Sound object (the one referred to by variable \( s \)).

Computer Science Idea: Changing memory doesn’t change the file
If you create another Sound object from the same file will you get the original sound or the sound with volume increased? You will get the original sound. The Sound object \( s \) was created by reading the file data into memory. The change to the Sound object was done in memory, but the file wasn’t changed. If you want to save your changes write them out to a file using the method
\[
\text{soundObj.write(String fileName)}; \quad \text{where soundObj is the name of the Sound object and fileName is the full path name of the file. So to save the changed Sound object above use } s.write("gettyLouder.wav");.
\]

8.3.2 Did that Really Work?
Now, is it really louder, or does it just seem that way? We can check it in several ways. You could always make the sound even louder by evaluating \text{increaseVolume} on our sound a few more times—eventually, you’ll be totally convinced that the sound is louder. But there are ways to test even more subtle effects.

If you compare graphs of the two sounds using the sound explorer, you will find that the graph of the sound does have greater amplitude after increasing it using our function. Check it out in Figure 8.17.

FIGURE 8.17: Comparing the graphs of the original sound (left) and the louder one (right)

Maybe you’re unsure that you’re really seeing a larger wave in the second picture. You can use a sound explorer to check the individual sample values. You can actually already see that in Figure 8.17—see that the first value (index number 0) is 59 in the original sound and 118 in the second sound. You can also check the value at any index using the sound explorer. Just click on a location and the value will be displayed for that location. To check the same location in the second explorer just type in the desired current index and it will show the value at that index. You’ll see that the louder sound really does have double the value of the
Section 8.3 Changing the Volume of Sounds 287

FIGURE 8.18: Comparing specific samples in the original sound (left) and the louder one (right)

same sample in the original sound.

Finally, you can always check for yourself from within DrJava. If you’ve been following along with the example\(^1\), then the variable \(s\) is the now louder sound. \(f\) should still be the filename of the original sound. Go ahead and make a new sound object which is the original sound—that is named below as \(s_{\text{Orig}}\) (for sound original). Check any sample that you want—it’s always true that the louder sound has twice the value than the the original sound.

```java
> System.out.println(s);
Sound file: c:/intro-prog-java/mediasources/gettysburg10-louder.wav number of samples: 220568
> System.out.println(f);
c:/intro-prog-java/mediasources/gettysburg10.wav
> Sound sOrig = new Sound(f);
> System.out.println(s.getSampleValueAt(0)); 118
> System.out.println(sOrig.getSampleValueAt(0)); 59
> System.out.println(s.getSampleValueAt(1)); 78
> System.out.println(sOrig.getSampleValueAt(1)); 39
> System.out.println(s.getSampleValueAt(999)); -80
> System.out.println(sOrig.getSampleValueAt(999)); -40
```

You can see from the last value that even negative values become more negative. That’s what’s meant by “increasing the amplitude.” The amplitude of the wave goes in both directions. We have to make the wave larger in both the positive and negative dimensions.

\(^1\)What? You haven’t? You should! It’ll make much more sense if you try it yourself!
It’s important to do what you just read in this chapter: *Doubt* your programs. Did that *really* do what I wanted it to do? The way you check is by *testing*. That’s what this section is about. You just saw several ways to test:

- By looking at the result overall (like with the graphs created by the explorer),
- By checking pieces of the results (like with the explorer or MediaTools), and
- By writing additional code statements that check the results of the original program.

**Figuring out how it Worked.**

Let’s walk through the code, slowly, and consider how this program worked.

```java
/**
 * Method to *double* the volume (amplitude) of the sound
 */
public void increaseVolume()
{
    SoundSample [] sampleArray = this.getSamples();
    SoundSample sample = null;
    int value = 0;
    int index = 0;

    // loop through all the samples in the array
    while (index < sampleArray.length)
    {
        sample = sampleArray[index];
        value = sample.getValue();
        sample.setValue(value * 2);
        index++;
    }
}
```

Recall our picture of the samples in a sound array.

```
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>59</td>
<td>39</td>
<td>16</td>
<td>10</td>
<td>-1</td>
</tr>
</tbody>
</table>
```

This is what `sound.getSamples()` would return: An array of `SoundSample` objects. The `while` loop allows us to walk through each sample, one at a time. The name (variable) `sample` will refer to each `SoundSample` object in turn.

The variable `index` starts out with a value of 0. This is less than the length of the array `sampleArray` so the body of the loop is executed. The variable `sample` is changed to refer to the first `SoundSample` object (the one at index 0).
Section 8.3  Changing the Volume of Sounds 289

The variable \texttt{value} will take on the value of 59 when \texttt{value=sample.getValue()} is executed. The value stored at that \texttt{SoundSample} object will be set to \texttt{value} times 2 (59 \times 2 = 118).

The value in variable \texttt{index} will be incremented by 1 (0 + 1 = 1). That's the end of the first pass through the body of the \texttt{while} loop. The loop will then start over. The test that \texttt{index} is less than the length of the array of samples will happen again. Since, it is still less the body of the loop will be executed (statements inside the open and close curly braces). The variable \texttt{sample} will be changed to refer to the second item in the array (the one at index 1).

Again, the variable \texttt{value} is set to the value of the \texttt{SoundSample} object. The value of the \texttt{SoundSample} object is set to twice the amount held in the variable \texttt{value}. 
This is what it will look like after 5 times through the loop.

8.3.3 Decreasing Volume

Decreasing volume, then, is the reverse of the previous process.

Program 66: Decrease an input sound’s volume

```java
/**
 * Method to halve the volume (amplitude) of the sound.
 */
public void decreaseVolume()
{
```
Section 8.3  Changing the Volume of Sounds

```java
SoundSample[] sampleArray = this.getSamples();
SoundSample sample = null;
int value = 0;
int index = 0;

// loop through all the samples in the array
while (index < sampleArray.length)
{
    sample = sampleArray[index];
    value = sample.getValue();
    sample.setValue(((int)(value * 0.5)));
    index++;
}
```

- Our method is called on a Sound object. The Sound object is implicitly passed to the method and is accessed using the keyword this. You can leave off the this on this.getSamples() since it is understood to be invoked on the current object.

- The variable sample will refer to a different SoundSample object each time through the loop.

- Each time sample refers to a new SoundSample object, we will get the value of that SoundSample object. We put that in the variable value.

- We then set the value held by the SoundSample object to 50% of its current value, by multiplying value by 0.5, and setting the sample value to that. However, because the value is an integer and the result of a computation with a floating point value (0.5) is a floating point number we must cast to integer using (int)(value * 0.5) to let the compiler know we realize that we will be throwing away the fractional part.

We can use it like this.

```java
> String f = FileChooser.pickAFile();
> System.out.println(f);
C:\intro-prog-java\mediasources\gettysburg10-louder.wav
> Sound sound1 = new Sound(f);
> System.out.println(sound1);
Sound file:
C:\intro-prog-java\mediasources\gettysburg10-louder.wav number of
samples: 220568
> sound1.play();
> sound1.decreaseVolume();
> sound1.play();

We can even do it again, and lower the volume even further.

>` sound1.decreaseVolume();
>` sound1.play();
```
8.3.4 Using a For Loop

Have you ever forgotten to declare the variable \texttt{index}? If you did the method wouldn’t compile. Did you ever forget to increment the the variable \texttt{index}? If you did the loop would never end until you hit \texttt{Reset}. Because of these problems programmers typically use a \texttt{for} loop instead of a \texttt{while} loop when they want to execute a block of commands a set number of times. A \texttt{for} loop is equivalent to a \texttt{while} loop (means the same thing to the computer). For loops are just less error prone for a programmer (though they are harder for a beginner to understand).

A \texttt{for} loop looks like this: \texttt{for (initialization; test; change)}. The initialization area lets you declare and initialize variables for use in the loop, the test is where you test if the loop should continue, and the change area is where you change the value of counters or indices used in the loop. For example, see the following new version of the method \texttt{decreaseVolume} which has been modified to use a \texttt{for} loop instead of a \texttt{while} loop.

Program 67: Decrease an input sound’s volume using a for loop

```java
/**
 * Method to halve the volume (amplitude) of the sound.
 */
public void decreaseVolume2()
{
    SoundSample[] sampleArray = this.getSamples();
    SoundSample sample = null;
    int value = 0;
    //int index = 0;
    // loop through all the samples in the array
    // while (index < sampleArray.length)
    for (int index = 0; index < sampleArray.length; index++)
    {
        sample = sampleArray[index];
        value = sample.getValue();
        sample.setValue((int) (value * 0.5));
        // index++;
    }
}
```

We have use the to-end-of-line comment ‘//’ to comment out some lines of code to show the difference between the \texttt{while} and \texttt{for} loops. Notice that what is different is that we don’t declare and initialize the index before the loop, it is done in the initialization part of the \texttt{for} statement. We also don’t increment the \texttt{index} as the last statement in the loop. This is moved to the change area in the \texttt{for} statement. So we have replaced three lines of code with one and made it more likely that we will remember to declare variables for use in the loop and change them. However, what really happens during execution is the same thing as what happened during the \texttt{while} loop. The declarations and initializations done in the
initialization part of the for loop will actually take place before the first test. The change of the loop variables will actually take place after each execution of the loop body and before the next test.

8.3.5 Making Sense of Methods

The lessons that we learned when writing picture methods (from Section 4.3.4) apply to sound methods as well. We want to write methods that do one and only one thing. We want to write methods that can be reused.

We can write methods that take an input value. For example, here’s a program to changeVolume. It accepts a factor that is multiplied by each sample value. This function can be used to increase or decrease the amplitude (and thus, the volume).

Program 68: Change a sound’s volume by a given factor

```java
/**
 * Method to change the volume (amplitude) of the sound 
 * by multiplying the current values in the sound by 
 * the passed factor.
 *
 * @param factor the factor to multiply by
 */
public void changeVolume(double factor)
{
    SoundSample[] sampleArray = this.getSamples();
    SoundSample sample = null;
    int value = 0;

    // loop through all the samples in the array
    for (int i = 0; i < sampleArray.length; i++)
    {
        sample = sampleArray[i];
        value = sample.getValue();
        sample.setValue((int) (value * factor));
    }
}
```

This program is clearly more flexible than increaseVolume() or decreaseVolume(). Does that make it better? Certainly it is for some purposes (e.g., if you were writing software to do general audio processing), but for other purposes, having separate and clearly named functions for increasing and decreasing volume may be better. Of course, you could modify increaseVolume() and decreaseVolume() to call changeVolume() with the appropriate factor. Remember that software is written for humans—write software that is understandable for the people who will be reading and using your software.

We are reusing the name sample a lot. We have used it in several methods in the Sound class. That’s okay. Names can have different meanings depending on their context. Variables declared in a method have meaning only inside that method. Methods can even use the same variable names as other methods. You can
even use the same variable names that you use in your methods in the interactions pane. This is a different context. If you create a variable in a method context (like `value` in Method 68 above), then that variable won’t exist when you get back out to the interactions pane. We can return values from a method context back out to the interactions pane (or a calling method) by using `return`, which we’ll talk more about later.

### 8.4 NORMALIZING SOUNDS

If you think about it some, it seems strange that the last two methods work! We can just multiply these numbers representing a sound—and the sound seems (essentially) the same to our ears just louder? The way we experience a sound depends less on the specific numbers than on the relationship between them. Remember that the overall shape of the sound waveform is dependent on many samples. In general, if we multiply all the samples by the same multiplier, we only effect our sense of volume (intensity), not the sound itself. (We’ll work to change the sound itself in future sections.)

A common operation that people want to do with sounds is to make them as **LOUD AS POSSIBLE**. That’s called normalizing. It’s not really hard to do, but it takes more lines of code than we’ve used previously and a few more variables, but we can do it. Here’s the algorithm, in English, that we need to tell the computer to do.

- We have to figure out what the largest sample in the sound is. If it’s already at the maximum value (the allowed range is -32768 to 32767 so the maximum allowed positive value is 32767), then we can’t really increase the volume and still get what seems like the same sound. Remember that we have to multiply all the samples by the same multiplier.

  It’s an easy (algorithm) to find the largest value—sort of a sub-program within the overall normalizing program. Define a name (say, `largest`) and assign it a small value (0 works). Now, check all the samples. If you find a sample with an absolute value larger than the `largest`, save that as the value for `largest`. Keep checking the samples, comparing to the new `largest`. Eventually, the very largest value in the array will be in the variable `largest`.

  To do this, we’ll need a way of figuring out the maximum value of two values. We can use an `if (value > largest)` to check if the current value is greater than the current largest and if so set that value to the largest. We can also save the index of that value so that we can check it with the sound explorer.

- Next, we need to figure out what value to multiply all the samples by. We want the largest value to become 32767. Thus, we want to figure out a multiplier such that

  \[(\text{multiplier})(\text{largest}) = 32767.\]

  Solve for the multiplier:

  \[\text{multiplier} = 32767/\text{largest}.\] The multiplier will need to be a floating point number (have a decimal component), so we need to convince Java that not
everything here is an integer. Turns out that’s easy—use 32767.0. Simply stick on ".0".

- Now, loop through all the samples, as we did for increaseVolume, and multiply the sample by the multiplier.

Here’s a program to normalize sounds.

```java
/**
 * Method to normalize (make as loud as possible) a sound.
 */
public void normalize()
{
    int largest = 0;
    int maxIndex = 0;
    SoundSample[] sampleArray = this.getSamples();
    SoundSample sample = null;
    int value = 0;

    // loop comparing the absolute value of the current value
    // to the current largest
    for (int i = 0; i < sampleArray.length; i++)
    {
        sample = sampleArray[i];
        value = Math.abs(sample.getValue());
        if (value > largest)
        {
            largest = value;
            maxIndex = i;
        }
    }

    // now calculate the multiplier to multiply by
    double multiplier = 32767.0 / largest;

    // print out the largest value and the multiplier
    System.out.println("The largest value was "+largest+
            " at index "+maxIndex);
    System.out.println("The multiplier is "+multiplier);

    // loop through all the samples and multiply by the
    // multiplier
    for (int i = 0; i < sampleArray.length; i++)
    {
        sample = sampleArray[i];
        sample.setValue((int)(sample.getValue() * multiplier));
    }
```

Program 69: Normalize the sound to a maximum amplitude
There are several notational items to note about this program.

- There is more than one loop in this method. That is okay. I can even use the same variable \( i \) in each loop. Since the variable is declared in the initialization area of the \texttt{for} loop it is only known in that loop. So, when we declare \( i \) again in the second loop it is seen as a new variable and not an attempt to declare the same variable more than once.

- There are \texttt{System.out.println()} statements in there! These statements can be really useful. First, they give you some feedback that the program is running—a useful thing in long-running programs. Second, they show you what it’s finding, which can be interesting. Third, it’s a terrific testing method and a way to debug your programs. Let’s imagine that the printout showed that the multiplier was less than 1.0. We know that that kind of multiplier decreases volume. You should probably suspect that something went wrong.

Here’s how to run this program:

```java
> Sound s = new Sound(FileChooser.getMediaPath("preamble.wav"));
> s.explore();
> s.normalize();
The largest value was 10216 at index 179377
The multiplier is 3.207419733750979
> s.play();
> s.explore();
```

Exciting, huh? Obviously, the interesting part is hearing the much louder volume, which is awfully hard to do in a book. So please try it now if you haven’t yet. But, you can see from Figure 8.19 that the values have increased.

![Figure 8.19: Comparing the original sound with the normalized one](image)

### 8.4.1 Generating Clipping

Earlier, we talked about clipping, what happens when the normal curves of the sound are broken by the limitations of the sample size. One way of generating clipping is to keep increasing the volume. Another way is to explicitly force clipping.
Section 8.4  Normalizing Sounds  297

What if you only had the largest and smallest possible sample values? What if all positive values (including zero), were the maximum value (32767) and all negative values were the minimum value (-32768)? Try this program, particularly on sounds with words in them.

Program 70: Set all samples to extreme values

```java
/**
 * Method to set all the sample values to the
 * maximum positive value if they were positive
 * (including 0) and the minimum negative
 * value if they were negative.
 */
public void forceToExtremes()
{
    SoundSample[] sampleArray = this.getSamples();
    SoundSample sample = null;

    // loop through the sample values
    for (int i = 0; i < sampleArray.length; i++)
    {
        // get the current sample
        sample = sampleArray[i];

        // if the value was positive (or zero) set it to the
        // maximum positive value
        if (sample.getValue() >= 0)
            sample.setValue(32767);

        // else force to max negative value
        else
            sample.setValue(-32768);
    }
}
```

Here’s how to run this program:

```java
> Sound s = new Sound(FileChooser.getMediaPath("preamble.wav"));
> s.play();
> s.explore();
> s.forceToExtremes();
> s.play();
> s.explore();
```

Look at Figure 8.20 and see that all the values have been set to extremes. When you play the sound back, you’ll hear a bunch of awful noises. That’s clipping. The really amazing thing is that you can still make out the words in sounds that
Modifying all Samples in a Sound

you manipulate with this method. Our ability to decipher words from noise is incredibly powerful.

**FIGURE 8.20:** Comparing the original sound with one with all values set to extremes.

### 8.5 CONCEPTS SUMMARY

In this chapter we worked with one-dimensional arrays, **while** loops, **for** loops, and conditionals.

#### 8.5.1 Arrays

Arrays are used to store many pieces of data of the same type. They allow you to quickly access a particular item in the array using an index. If you couldn’t use an array, you would have to create a separate variable name for each piece of data.

To declare a variable that refers to an array use the type followed by open ‘[’ and close ‘]’ square brackets and then the variable name.

```java
SoundSample[] sampleArray;
```

This declares an array of **SoundSample** objects. The value stored at each position in the array is a reference to a **SoundSample** object.

Arrays are objects and you can find out how large an array is using: `arrayReference.length`.

Notice that this isn’t a method call but instead it accesses a public field.

You can get an element of the array using `arrayReference[index]`. Where the index values can range from 0 to `arrayReference.length-1`.

#### 8.5.2 Loops

Loops are used to execute a block of statements while a boolean expression is true. Most loops have variables that change during the loop which eventually cause the boolean expression to be false and the loop to stop. Loops that never stop are called infinite loops.

We used two types of loops in this chapter: **while** and **for**. The **while** loop is usually used when you don’t know how many times a loop needs to execute and the **for** loop is usually used when you do know how many times the loop will execute.

We introduced a special kind of **for** loop in this chapter which is often called a “for each” loop. This new loop was introduced in Java 1.5 and loops through all the elements of an array one at a time.
SoundSample[] sampleArray = this.getSamples();
int value = 0;

// loop through all the samples in the array
for (SoundSample sample : sampleArray) {
    value = sample.getValue();
    // do something to the value
    sample.setValue(value);
}

This declares a variable sample that is of the type SoundSample and each
time through the loop the sample variable will refer to a different element of the
array until all the elements have been processed.

The while loop has the keyword while followed by a boolean expression and
then a block of statements between an open and close curly brace. If the boolean
expression is true the body of the loop will be executed. If the boolean expression is
false execution will continue after the body of the loop (after the close curly brace).
If you just want to execute one statement in the body of the loop then you don’t
need the open and close curly braces, but you should indent the statement.

while (boolean expression) {
    statement1;
    statement2;
    ...
}

If you use a while loop to execute a block of statements a set number of times
you will need to declare a variable before the while and that variable will need to
be changed in the body of the loop. You may also need to declare other variables
that you use in the loop before the while. Don’t declare variables inside the loop
because you will use more memory that way.

SoundSample[] sampleArray = this.getSamples();
SoundSample sample = null;
int index = 0;
int value = 0;

// loop through all the samples in the array
while (index < sampleArray.length) {
    sample = sampleArray[index];
    value = sample.getValue();
    // do something to the value
    sample.setValue(value);
    index++;
}

The for loop does the same thing as a while loop but it lets you declare the
variables that you need for the loop, specify the boolean expression to test, and
specify how to change the loop variables all in one place. This means you are less
likely to forget to do each of these things.
SoundSample[] sampleArray = this.getSamples();
SoundSample sample = null;
int value = 0;

// loop through all the samples in the array
for (int index = 0; index < sampleArray.length; index++)
{
    sample = sampleArray[index];
    value = sample.getValue();
    // do something to the value
    sample.setValue(value);
}

8.5.3 Conditional Execution

To conditionally execute one statement use the if keyword followed by a boolean expression inside of an open and close parenthesis. Put the statement that you only want executed if the boolean expression is true on a new line and indent it. If the boolean expression is false then execution will continue with the next statement.

if (boolean expression)
    // statement to execute if the boolean expression is true
    statement
    // next statement

To conditionally execute a block of statements use the if keyword followed by a boolean expression inside of an open and close parenthesis. Put the statements to be executed when the boolean expression is true inside of an open and close curly brace. Indent the statements to make it easier to visually see that these statements will only be executed if the boolean expression is true. If the boolean expression is false execution will continue with the statement following the close curly brace.

if (boolean expression)
{
    statements
}

If you want to execute one block of statements if the boolean expression is true and another if it is false use the else keyword as well. Put the statements that you want to execute when the boolean expression is true inside of an open and close open and close curly brace after the if(booleanExpression). Next, add the keyword else and put the statements that you want executed when the boolean expression is false inside of an open and close curly brace.

if (boolean expression)
{
    statements
} else {
    statements
}
Section 8.5  Concepts Summary

OBJECTS AND METHODS SUMMARY

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

<table>
<thead>
<tr>
<th>Sound</th>
<th>Sound objects are encodings of sounds, typically coming from a WAV file.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SoundSample</td>
<td>An object that represents a sample in a sound. The value of a sample is between -32768 and 32767 representing the voltage that a microphone would generate at a given instant when recording a sound. The length of the instant is typically either (\frac{1}{44,100}) of a second (for CD-quality sound) or (\frac{1}{22,050}) of a second (for good enough sound on most computers). A SoundSample object remembers what sound it came from, so if you change its value, it knows to go back and change the right sample value in the sound.</td>
</tr>
</tbody>
</table>

Here are the functions used or introduced in this chapter:

Math methods

| Math.abs(number) | Takes a number and returns it as a positive number. |

FileChooser methods

| FileChooser.getMediaPath(String fileName) | Takes a filename as input, and returns the full path name of the file with the media directory before the filename. |
| FileChooser.pickAFile() | Lets the user pick a file and returns the complete path name as a string. |
| FileChooser.setMediaPath(String directory) | Takes a directory as input and sets the directory name to be what is added to the passed filename using getMediaPath. |

Sound methods
### Chapter 8 Modifying all Samples in a Sound

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>blockingPlay()</td>
<td>Plays the Sound object it is invoked on, and makes sure that no other sound plays at the exact same time. (Compare two blockingPlay's with two play's right after each other.)</td>
</tr>
<tr>
<td>getLength()</td>
<td>Returns the number of samples in the Sound object it is invoked on.</td>
</tr>
<tr>
<td>getSamples()</td>
<td>Returns an array of SoundSample objects for the Sound object it is invoked on.</td>
</tr>
<tr>
<td>getSampleValueAt(int index)</td>
<td>Takes an index (an integer value), and returns the value of the sample at that index for the Sound object it is invoked on.</td>
</tr>
<tr>
<td>getSamplingRate()</td>
<td>Returns the number representing the number of samples in each second for the Sound object it is invoked on.</td>
</tr>
<tr>
<td>play()</td>
<td>Plays the Sound object it is invoked on.</td>
</tr>
<tr>
<td>setSampleValueAt(int index, int value)</td>
<td>Takes an index, and a value, and sets the value of the sample at the given index in the Sound object it was invoked on to the given value.</td>
</tr>
<tr>
<td>write(String fileName)</td>
<td>Takes a filename (a string) and writes the sound in the Sound object it is invoked on to that file as a WAV file. (Make sure that the filename ends in “.wav” if you want the operating system to treat it right.)</td>
</tr>
</tbody>
</table>

### SoundSample methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getValue()</td>
<td>Returns the value for the SoundSample object it is invoked on.</td>
</tr>
<tr>
<td>setValue(int value)</td>
<td>Sets the value for the SoundSample object it is invoked on to be the passed value.</td>
</tr>
</tbody>
</table>

### PROBLEMS

**8.1.** Open up the Sonogram view and say some vowel sounds. Is there a distinctive pattern? Do “Oh’s” always sound the same? Do “Ah’s”? Does it matter if you switch speakers—are the patterns the same?

**8.2.** Get a couple of different instruments and play the same note on them into MediaTool application’s sound editor with the sonogram view open. Are all “C’s” made equal? Can you see some of why one sound is different than another?
8.3. Try out a variety of WAV files as instruments, using the piano keyboard in the MediaTools application sound editor. What kinds of recordings work best as instruments?

8.4. The increase volume Method 65 (page 284) uses a while loop. Convert it to a for loop.

8.5. The decrease volume Method 66 (page 290) uses a while loop. Convert it to a for loop.

8.6. In section 8.3.2, we walked through how Program 65 (page 284) worked. Draw the pictures to show how Program 66 (page 290) works, in the same way.

8.7. What happens if you increase a volume too far? Explore that by creating a Sound object, then increase the volume once, and again, and again. Does it always keep getting louder? Or does something else happen? Can you explain why?

8.8. Instead of multiplying samples by a multiplier (like 2 or 0.5), try adding a value to them. What happens to a sound if you add 100 to every sample? 1000?

8.9. Try sprinkling in some specific values into your sounds. What happens if you set the value of a few hundred samples in the middle of a sound to 32767? Or a few hundred -32768? Or a bunch of zeroes? What happens to the sound?

TO DIG DEEPER

There are many wonderful books on psychoacoustics and computer music. One of my favorites for understandability is *Computer Music: Synthesis, Composition, and Performance* by Dodge and Jerse [8]. The bible of computer music is Curtis Roads’ massive *The Computer Music Tutorial* [22].
C H A P T E R 9

Modifying Samples using Ranges

9.1 MANIPULATING DIFFERENT SECTIONS OF THE SOUND DIFFERENTLY
9.2 CREATE A SOUND CLIP
9.3 SPLICING SOUNDS
9.4 REVERSING A SOUND
9.5 MIRRORING A SOUND
9.6 CONCEPTS SUMMARY

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

• To manipulate parts of a sound differently.
• To create a sound clip.
• To splice sounds together to make sound compositions.
• To reverse sounds.
• To mirror sounds.

The computer science goals for this chapter are:

• To return a value from a method.
• To use ranges in iteration.
• To change more than one variable in a loop.
• To identify some algorithms that cross media boundaries.

9.1 MANIPULATING DIFFERENT SECTIONS OF THE SOUND DIFFERENTLY

Manipulating all of the samples in a sound in the same way can be useful, but really interesting effects come from chopping up sounds and manipulating them differentially: Some words this way, other sounds that way. How would you do that? We need to be able to loop through portions of the sample, without walking through the whole thing. This turns out to be an easy thing to do, but we need to manipulate samples somewhat differently (e.g., we have to use our for loop in a slightly different way).
Section 9.1 Manipulating Different Sections of the Sound Differently

Recall that each sample has an index number, and that we can get each individual sample value with `getSampleValue(int index)` (with an index number as input). We can set any sample with `setSampleValue(int index, int value)` (with inputs of an index number, and a new value). That’s how we can manipulate samples without using `getSamples()` and `SoundSample` objects. But we still don’t want to have to write code like:

```java
sound.setSampleValue(0, 12);
sound.setSampleValue(1, 28);
```

Not for tens of thousands of samples! So, we will continue to use a `for` loop. However, if we are not processing the entire sound in the same way the index value that we start at won’t necessarily be 0 and the last index value won’t necessarily be the length of the sound minus 1.

What if we want to increase the sound for the first half of the sound, then decrease it in the second half. How could we do that? First we will need to calculate the half-way point. We can determine that by dividing the length of the sound by 2. Since the length and 2 are both integers the result will also be an integer so no casting is needed (any values after the decimal point will be thrown away). We will need two loops. One loop will start at the beginning of sound (0) and loop till the half-way point. The second loop will start at the half-way point and loop to the end of the sound (length - 1).

Program 71: Increase the volume then decrease

```java
/**
 * Method to increase the first half of the sound (double it) and then decrease the second half (half it).
 */
public void increaseAndDecrease()
{
    int half = this.getLength() / 2;
    int value = 0;

    // loop through the first half of the sound
    for (int i = 0; i < half; i++)
    {
        // get the current value
        value = this.getSampleValueAt(i);

        // set the value to 2x the original
        this.setSampleValueAt(i, value * 2);
    }

    // loop through the second half of the sound
    for (int i = half; i < this.getLength(); i++)
    {
        // get the current value
        value = this.getSampleValueAt(i);

        // divide by 2 to halve
        this.setSampleValueAt(i, value / 2);
    }
}
```
value = this.getSampleValueAt(i);

    // set the value to half the original
    this.setSampleValueAt(i,(int) ( value * 0.5));
}
}

There are two loops in increaseAndDecrease(), each of which deals with one half of the sound.

- The first loop deals with the samples from 0 to halfway through the sound. Those samples all get multiplied by 2, to double their amplitude.

- The second loop goes from halfway through to the end of the sound. Here, we multiply each sample by 0.5 thus decreasing the sound by 50%.

Another Way of Writing Array References.
It’s worth pointing out that, in many languages, square brackets ([ ]) are standard notation for manipulating arrays. It works that way in Java. For any array, array[index] returns the index-th element in the array. The number inside the square brackets is always an index variable, but it’s sometimes referred to as a subscript, because of the way that mathematicians refer to the i-th element of a, e.g., a_i.

Let’s demonstrate how to get the elements at various index values.

```java
> Sound sound = new Sound(FileChooser.getMediaPath("croak.wav"));
> SoundSample[] sampleArray = sound.getSamples();
> System.out.println(sampleArray[0]);
Sample at index 0 has value 0
> System.out.println(sampleArray[1]);
Sample at index 1 has value 0
> System.out.println(sampleArray[8000]);
Sample at index 8000 has value 512
```

9.2 CREATE A SOUND CLIP

Sometimes a sound is too long and you want just part of it. This can happen when you record a sound. There may be silence before and after the actual sound. Or, you may want to pull one word out of a sound. You can “clip” a sound just as we clipped a picture by copying just part of the sound. What if we want to pull the “This” out of the sound thisisatest.wav? How can we tell where it ends? Open an explorer on the sound using sound.explore();. Click at the first flat area after the first non-flat area (silence should have values near 0). Click the PLAY BEFORE button to play the part of the sound before the current index. You can use the arrow buttons to change the current index as well. Using the explorer we see that the word “this” ends at about 8500 (Figure 9.1).

So to copy just part of a sound into another sound we will need to create a new Sound object. One of the ways to make a new Sound object is to tell it how
Section 9.2 Create a Sound Clip

FIGURE 9.1: Exploring the "This is a test" to find the end of the first word

many samples it will have. To calculate the number of samples we can subtract the ending value from the starting value and add 1.

We can then create a new sound and loop copying from the start to the end from the source Sound object into the target Sound object starting at the beginning of the target sound. We have to make sure to increment both the index in the source and the index in the target. If we forget to increment the source index we will copy the same source sample over and over and if we fail to increment the target index we will copy to the same place in the target over and over.

We need to return our new sound object in order to be able to refer to it again. To return something from a method we need to specify the type of the thing that will be returned in the method declaration (replacing the void keyword). We also need to use the keyword return followed by what we want to return.

Program 72: Create a sound clip

```java
/**
 * Method to create a new sound by copying just part of
 * the current sound to a new sound
 * @param start the index to start the copy at (inclusive)
 * @param end the index to stop the copy at (inclusive)
 * @return a new sound with just the samples from start to
 * end in it
 */
public Sound clip(int start, int end)
{
    // calculate the number of samples in the clip
```
int lengthInSamples = end - start + 1;
Sound target = new Sound(lengthInSamples); // hold clip
int value = 0; // holds the current sample value
int targetIndex = 0; // index in target sound

// copy from start to end from source into target
for (int i = start; i <= end; i++, targetIndex++) {
    value = this.getSampleValueAt(i);
    target.setSampleValueAt(targetIndex, value);
}

return target;

Notice that we said that we would return a Sound object from this method by saying that the type of thing returned is from the class Sound. At the end of the method we use the keyword return followed by the variable that refers to the new Sound object. So, in order to refer to this new Sound object again we will need to declare a variable and set the value of that variable to refer to the returned Sound object.

> Sound test = new Sound(FileChooser.getMediaPath("thisisatest.wav"));
> test.explore();
> Sound clip = test.clip(0,8500);
> clip.play();
> clip.explore();

Use the explorer on the original sound and the clip. Change the number of samples between pixels to be 100 in the clip. Then compare the sample values. Convince yourself that the clip does have the same values as the original. (Figure 9.2)

9.3 SPlicing sounds

Splicing sounds is a term that dates back to when sounds were recorded on tape, so juggling the order of things on the tape involved literally cutting the tape into segments and then gluing it back together in the right order. That’s “splicing”. When everything is digital, it’s much easier.

To splice sounds, we simply have to copy elements around in the array. It’s easiest to do this with two (or more) arrays, rather than copying within the same array. Splicing lets you create all kinds of sounds, speeches, non-sense, and art.

The easiest kind of splice to do is when the sounds are in separate files. All that you need to do is to copy each sound, in order, into a target sound. You need to keep track of the next index in the target sound. Here’s a recipe that creates the start of a sentence “Guzdial is...”. (Readers are welcome to complete the sentence.)
**Method to splic two sounds together with some silence between them**

* into the current sound

```java
public void splice() {
    Sound sound1 =
        new Sound(FileChooser.getMediaPath("guzdial.wav"));
    Sound sound2 =
        new Sound(FileChooser.getMediaPath("is.wav"));
    int targetIndex = 0; // the starting place on the target
    int value = 0;

    // copy all of sound 1 into the current sound (target)
    for (int i = 0; i < sound1.getLength(); i++, targetIndex++) {
        value = sound1.getSampleValueAt(i);
        this.setSampleValueAt(targetIndex, value);
    }

    // create silence between words by setting values to 0
    for (int i = 0; i < (int) (this.getSamplingRate() * 0.1); i++, targetIndex++) {
        this.setSampleValueAt(targetIndex, 0);
    }

    // copy all of sound 2 into the current sound (target)
    for (int i = 0;
```
To test this we need a “blank” sound that the two sounds will be copied to. You can use the file “sec3silence.wav” for this. It holds 3 seconds of silence, which should be more than enough.

```java
> String silence = FileChooser.getMediaPath("sec3silence.wav");
> Sound target = new Sound(silence);
> target.play();
> target.splice();
> target.play();
```

There are three loops in this method `splice`, each of which copies one segment into the current (target) sound—a segment being either a word or a silence between words.

- The function starts by creating sound objects for the word “Guzdial” (`s1`) and the word “is” (`s2`).

- Notice that we set `targetIndex` (the index for the target sound) equal to 0 before the first loop. We then increment it in every loop, but we never again set it to a specific value. That’s because `targetIndex` is always the index for the `next empty sample` in the target sound. Because each loop follows the previous, we just keep tacking samples onto the end of the current sound (the target).

- In the first loop, we copy each and every sample from `s1` into the current sound `this`. We have the index `i` go from 0 to one before the length of `s1`. We get the sample value at index `i` from `s1`, then set the sample value at `targetIndex` in the current sound to that value. We then increment both `i` and `targetIndex`.

- In the second loop, we create 0.1 seconds of silence. Since `this.getSamplingRate()` gives us the number of samples in one second of the current sound, 0.1 times that tells us the number of samples in 0.1 seconds. We don’t get any source value here—we simply set the `targetIndex`-th sample to 0 (for silence), then increment the `targetIndex`.

- Finally, we copy in all the samples from `s2`, just like the first loop where we copied in `s1`.

The more common kind of splicing is when the words are in the middle of an existing sound, and you need to pull them out from there. The first thing to do in splicing like that is to figure out the index numbers that delimit the pieces you’re interested in. Using the explorer, that’s pretty easy to do.
Section 9.3 Splicing Sounds

- Open an explorer using `sound.explore()`.
- Click the mouse button to choose a current position and then play the sound before or after that current position.
- Or, select part of the sound by clicking the mouse when the cursor points to where a place to start and dragging the mouse to the end of an area of interest. This selection will highlight and you can play the selection.

Using exactly this process, Mark found the ending points of the first few words in `preamble10.wav`. (He assumed that the first word starts at the index 0, though that might not always be true for every sound.)

<table>
<thead>
<tr>
<th>Word</th>
<th>Ending index</th>
</tr>
</thead>
<tbody>
<tr>
<td>We</td>
<td>15730</td>
</tr>
<tr>
<td>the</td>
<td>17407</td>
</tr>
<tr>
<td>People</td>
<td>26726</td>
</tr>
<tr>
<td>of</td>
<td>32131</td>
</tr>
<tr>
<td>the</td>
<td>33413</td>
</tr>
<tr>
<td>United</td>
<td>40052</td>
</tr>
<tr>
<td>States</td>
<td>55510</td>
</tr>
</tbody>
</table>

Writing a loop that copies things from one array to another requires a little bit of juggling. You need to think about keeping track of two indices: Where you are in the array that you’re copying from, and where you are in the array that you’re copying to. These are two different variables, tracking two different indexes. But they both increment in the same way.

Below is the recipe that changes the preamble from "We the people of the United States" to "We the UNITED people of the United States."

```java
public void splicePreamble()
{
    String file = FileChooser.getMediaPath("preamble10.wav");
    Sound source = new Sound(file);
    int targetIndex = 0; // start copying to first sample value
    int value = 0;

    // loop copying the "We the " into the current sound
    for (int sourceIndex = 0; sourceIndex < 17407;
         sourceIndex++, targetIndex++) {
```

Be sure to set the media path before trying this on your computer.
value = source.getSampleValueAt(sourceIndex);
this.setSampleValueAt(targetIndex, value);

// loop copying the "united" into the current sound
for (int sourceIndex = 33414;
     sourceIndex < 40052;
     sourceIndex++, targetIndex++) {
value = source.getSampleValueAt(sourceIndex);
this.setSampleValueAt(targetIndex, value);
}

// copy the "people of the United States"
for (int sourceIndex = 17408;
     sourceIndex < 55510;
     sourceIndex++, targetIndex++) {
value = source.getSampleValueAt(sourceIndex);
this.setSampleValueAt(targetIndex, value);
}

> String silence = FileChooser.getMediaPath("sec3silence.wav");
> Sound target = new Sound(silence);
> target.play();
> target.splicePreamble();
> target.play();

The first loop copies the words “We the” into the current sound. The second loop is copies the word “united” into the current sound. The last loop copies the words “people of the United States” into the current sound. Notice that the value of targetIndex is set to 0 at the beginning so we start copying at the beginning of the current sound. In each loop we increment targetIndex but we never reset it’s value so it always points to the next place in the current sound to copy to.

Figure 9.3 shows the original preamble10.wav file in the left sound explorer, and the new spliced one (saved with write(String fileName)) on the right.

Let’s see if we can figure out what’s going on mathematically. Recall the table back on page 311. First we copy the range from 0 to 17406 to the target sound. This means we copied 17407 pixels (17406 - 0 + 1 = 17407). After the first loop the value of targetIndex will be 17407. Next we copy the range from 33414 to 40051 which means we copy (40051 - 33414 + 1 = 6,638) 6638 pixels. After the second loop the value of targetIndex will be 24045 (17407 + 6638 = 24045). Next we copy the range from 17408 to 55509 which is (55509 - 17408 + 1 = 38102) pixels. The total number of copied pixels is (17407 + 6638 + 38102 = 62,147). The value of targetIndex will be 62147 after the last loop. You can add System.out.println("Target index is " + targetIndex); after each loop to
check that this is correct.

Program 75: Splice preamble and show target index

```java
/**
 * Method to splice "We the " then "United" then
 * "people of the United States" into the current
 * sound
 */
public void splicePreamble()
{
    String file = FileChooser.getMediaPath("preamble10.wav");
    Sound source = new Sound(file);
    int targetIndex = 0; // start copying to first sample value
    int value = 0;

    // loop copying the "We the " into the current sound
    for (int sourceIndex = 0;
        sourceIndex < 17407;
        sourceIndex++, targetIndex++) {
        value = source.getSampleValueAt(sourceIndex);
        this.setSampleValueAt(targetIndex, value);
    }

    // print the value of the target index
    System.out.println("Target index is " + targetIndex);

    // loop copying the "united" into the current sound
    for (int sourceIndex = 33414;
        sourceIndex < 40052;
        sourceIndex++, targetIndex++) {
        value = source.getSampleValueAt(sourceIndex);
        this.setSampleValueAt(targetIndex, value);
    }

    // print the value of the target index
    System.out.println("Target index is " + targetIndex);

    // copy the "people of the United States"
    for (int sourceIndex = 17408;
        sourceIndex < 55510;
        sourceIndex++, targetIndex++) {
        value = source.getSampleValueAt(sourceIndex);
        this.setSampleValueAt(targetIndex, value);
    }

    // print the value of the target index
    System.out.println("Target index is " + targetIndex);
}
```
We can also use the explorer to check that the last copied pixel is at 62146 by checking the value at 62147. It should still be 0, as should all the values from that index to the end of the sound.

Each of the loops that copies part of the preamble sound into the current sound is very similar. To make a general splice method we will pass in the Sound object to copy from, the starting index to use in that passed sound, the index to stop before in the passed sound, and the place to start the copy to in current sound.

![Program 76: General splice method](image)

```java
/**
 * Method to copy part of the passed sound into this sound at
 * the given start index
 * @param source the source sound to copy from
 * @param sourceStart the starting index to copy from in the
 * source (the copy will include this)
 * @param sourceStop the ending index (the copy won’t include
 * this)
 * @param targetStart the index to start copying into
 */
public void splice(Sound source, int sourceStart, int sourceStop, int targetStart) {
    // loop copying from source to target
    for (int sourceIndex = sourceStart, targetIndex = targetStart; sourceIndex < sourceStop && targetIndex < this.getLength(); sourceIndex++, targetIndex++)
        this.setSampleValueAt(targetIndex, source.getSampleValueAt(sourceIndex));
}
```

FIGURE 9.3: Comparing the original sound (left) to the spliced sound (right)
This new object method can be used to splice “united” in the phrase “We the people of the United States” as shown below:

### Program 77: Using the General Splice Method

```java
/**
 * Method to splice the preamble into the current sound so that it says We the United people of the United States
 */
public void splicePreamble2()
{
    Sound preamble = new Sound(chooser.getMediaPath("preamble10.wav"));

    // first splice the "we the" into the current sound
    this.splice(preamble, 0, 17407, 0);

    // now splice the "united" into the current sound
    this.splice(preamble, 33414, 40052, 17407);

    /* now splice the "people of the United States" into the current sound */
    this.splice(preamble, 17408, 55510, 24045);
}
```

You can execute this new method using the following:

```java
> Sound target = new Sound(chooser.getMediaPath("sec3silence.wav"));
> target.splicePreamble2();
> target.explore();
```

Compare the sound created using the `splicePreamble` method versus the sound created using the `splicePreamble2` method. They should result in exactly the same sound. Why should we try to write general methods? Take a look at `splicePreamble2` it is much easier to read than `splicePreamble`. We want general methods because they are easier to reuse and make our programs smaller and easier to understand.

### 9.4 Reversing a Sound

In the splicing example, we copied the samples from the words just as they were in the original sound. We don’t have to always go in the same order. We can reverse the words—or make them faster, slower, louder, or softer. For an example, here’s
a recipe that reverses a sound so that you can play it backwards.

Program 78: Reverse a sound

```java
/**
 * Method to reverse the current sound.
 */
public void reverse()
{
    Sound orig = new Sound(this.getFileName());
    int length = this.getLength();

    // loop through the samples
    for (int targetIndex = 0, sourceIndex = length - 1;
            targetIndex < length && sourceIndex > 0;
            targetIndex++, sourceIndex--)
    {
        this.setSampleValueAt(targetIndex, orig.getSampleValueAt(sourceIndex));
    }
}
```

This method firsts creates another Sound object from the same file as the current Sound object. This will make a copy of the original sound. Next the method saves the length of the current Sound object. Then it loops.

The loop initializes the value of `targetIndex` to 0, and the value of `sourceIndex` to the length of the sound minus 1. It loops while `targetIndex` is less than the length of the sound and the `sourceIndex` is greater than 0. It increments `targetIndex` by 1 after the body of the loop and it decrements the value of `sourceIndex` by one each time through the loop.

Why does is start `sourceIndex` at the length of the sound minus 1 and decrement it each time through the loop? Remember that the last valid index is at the length minus 1, which is why the `sourceIndex` starts with this value. So we copy from the end of the source sound (length - 1) to the beginning of the target sound (0) during the first execution of the loop. The second time through the loop we copy from the next to last sound sample in the source (length - 2) to the second position in the target (1). We will keep looping until the `targetIndex` equals the length of the sound.

To use this method to reverse a sound try:

```bash
> Sound s = new Sound(FileChooser.getMediaPath("croak.wav"));
> s.play();
> s.explore();
> s.reverse();
> s.play();
> s.explore();
```
9.5 MIRRORING A SOUND

Once we know how to play sounds forwards and backwards, mirroring the sound is the exact same process as mirroring pictures! Compare this to Program 19 (page 140). Do you see that this is the same algorithm, though we’re dealing with a different medium?

Program 79: Mirror a sound, front to back

```java
/**
 * Method to mirror a sound front to back
 */
public void mirrorFrontToBack()
{
    int length = getLength(); // save the length
    int lastIndex = length - 1; // last valid index
    int mirrorPoint = length / 2; // mirror around this
    int value = 0; // hold the current value

    // loop from 0 to mirrorPoint
    for (int i = 0; i < mirrorPoint; i++) {
        value = this.getSampleValueAt(mirrorPoint - i);
        this.setSampleValueAt(mirrorPoint + i);
    }
}
```

To use this recipe try:

```java
> Sound s = new Sound(FileChooser.getMediaPath("croak.wav");
> s.explore();
> s.mirrorFrontToBack();
> s.explore();
```

The length of the sound in the file "croak.wav" is 8808, so the mirror point is at 4404. Use the explorer to check the values on either side of the mirror point.
9.6 CONCEPTS SUMMARY

This chapter covered working with ranges in loops and how to return a value from a method.

9.6.1 Ranges in Loops

To limit the range of a loop change the starting value and/or ending value. For example to mirror the front of the sound to the back start with an index value of 1 and end before the mirror point (midpoint).

```java
public void mirrorFrontToBack() {
    int length = getLength(); // save the length
    int mirrorPoint = length / 2; // mirror around this
    int value = 0; // hold the current value

    // loop from 1 to mirrorPoint
    for (int i = 1; i < mirrorPoint; i++) {
        value = this.getSampleValueAt(mirrorPoint − i);
        this.setSampleValueAt(mirrorPoint+i, value);
    }
}
```

9.6.2 Returning a Value from a Method

To declare a method you specify the visibility for the method, the type of thing it returns, the name of the method, and the parameter list inside parentheses. This is followed by the body of the method which is inside of an open and close curly brace.

```java
visibility returnType name(parameterList) {
    // statements in method
    // return a value
    return valueToReturn;
}
```
Methods that do not return any value use the keyword `void` as the `returnType`. Methods that do return a value use the type of that value for the `returnType` and then have a `return` keyword in them that is followed by the thing to return. Remember that a type is any of the primitive types or the name of a class.

Here is an example public method declaration that doesn’t return anything and the name of the method is `mirrorFrontToBack` and it doesn’t take any parameters.

```java
public void mirrorFrontToBack()
```

Here is an example public method declaration that returns an object of the class `Sound`.

```java
public Sound echo(int delay, int numEchoes)
```

Notice that it gives a return type of `Sound`. The body of the method must have the keyword `return` in it and it must return an object that is an instance of the class `Sound`.

**PROBLEMS**

9.1. Rewrite Program 71 (page 305) so that two input values are provided to the function: The sound, and a `percentage` of how far into the sound to go before dropping the volume.

9.2. Rewrite Program 71 (page 305) so that you normalize the first second of a sound, then slowly decrease the sound in steps of 1/5 for each following second. (How many samples are in a second? `getSamplingRate()` is the number of samples per second for the given sound.)

9.3. Try rewriting Program 71 (page 305) so that you have a linear increase in volume to halfway through the sound, then linearly decrease the volume down to zero in the second half.

9.4. I think that if we’re going to say “We the UNITED people” in the splice (Program 74 (page 311)), the “UNITED” should be really emphasized—really loud. Change the recipe so that the word “united” is maximally loud (normalized) in the phrase “united people.”

9.5. Try using a stopwatch to time the execution of the recipes in this chapter. Time from hitting return on the command, until the next prompt appears. What is the relationship between execution time and the length of the sound? Is it a linear relationship, i.e., longer sounds take longer to process and shorter sounds take less time to process? Or is it something else? Compare the individual recipes. Does normalizing a sound take longer than raising (or lowering) the amplitude a constant amount? How much longer? Does it matter if the sound is longer or shorter?

9.6. Make an audio collage. Make it at least five seconds long, and include at least two different sounds (e.g., come from different files). Make a copy of one of those different sounds and modify it using any of the techniques described in this chapter (e.g., mirroring, splicing, and volume manipulations). Splice together the original two sounds and the modified sound to make the complete collage.

9.7. Compose a sentence that no one ever said, by combining words from other sounds into a grammatically correct new sound. Write a method named `audioSentence` to generate a sentence out of individual words. Use at least three words in your sentence! You can use the words in the `mediasources` folder on your CD or record
Chapter 9 Modifying Samples using Ranges

your own words. Be sure to include a tenth (1/10) of a second pause between the words. (Hint 1: Remember that zeroes for the sample values generate silence or pause.) (Hint 2: Remember that the sampling rate is the number of samples per second. From there, you should be able to figure out how many samples need to be made zero to generate a 1/10 of a second pause.) Be sure to access your sounds in your Media Folder using `getMediaPath` so that it will work for users of your program will work as long as they first execute `setMediaPath`.

9.8. Write a method called `erasePart` to set all the samples in the 2nd second of "thisisatest.wav" to 0's—essentially, making the 2nd second go silent. (Hint: Remember that `getSamplingRate()` tells you the number of samples in a single second in a sound.) Play and return the partially-erased sound.

9.9. We’ve seen a function that reverses a sound and a function that can process samples by index number.

Write a function called `reverseLastHalf` that reverses just the second half of the current sound. For example, if the sound said “MarkBark” the returned sound should say “MarkkraB.”

9.10. Write a method similar to Program 79 (page 317) that mirrors from back to front.

TO DIG DEEPER

When you are using the MediaTools application, you are actually using a programming language called Squeak, developed initially and primarily by Alan Kay, Dan Ingalls, Ted Kaehler, John Maloney, and Scott Wallace [17]. Squeak is now open-source¹, and is an excellent cross-platform multimedia tool. There is a book that introduces Squeak including its sound capabilities [13], and another book on Squeak [14] that includes a chapter on Siren, a variation of Squeak by Stephen Pope especially designed for computer music exploration and composition.

¹http://www.squeak.org
CHAPTER 10

Making Sounds by Combining Pieces

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

• To blend sounds so that one fades into another.
• To create echoes.
• To change the frequency (pitch) of a sound.
• To create sounds that don’t exist in nature by composing more basic sounds (sine waves).
• To choose between sound formats such as MIDI and MP3 for different purposes.

The computer science goals for this chapter are:

• To explain blending as an algorithm that crosses media boundaries.
• To build programs from multiple methods.
• To introduce class (static) methods.
• To introduce private methods.

10.1 COMPOSING SOUNDS THROUGH ADDITION
Creating sounds digitally that didn’t exist previously is lots of fun. Rather than simply moving around samples or multiplying them, we actually change their values—add waves together. The result are sounds that never existed until you made them.
In physics, adding sounds involves issues of canceling waves out and enforcing other factors. In math, it’s about matrices. In computer science, it’s the easiest process in the world! Let’s say that you’ve got a sound, `source`, that you want to add in to the current `Sound` object. Simply add the values at the same index numbers! That’s it!

```java
// loop through all of the source
for (int i = 0; i < source.getLength(); i++) {

    // add source sound value and this sound value
    value = this.getSampleValueAt(i) + 
            source.getSampleValueAt(i);

    // set the value in this sound to the new value
    this.setSampleValueAt(i, value);
}
```

FIGURE 10.1: The top and middle waves are added together to create the bottom wave
10.2 BLENDING SOUNDS

In this example, we take two sounds—someone saying "Aah!" and an bassoon instrument sound of C in the fourth octave—and blend the two sounds. The way we do this is to first copy part of the first sound, "Aah!", then copy 50% of each sound, and then copying the second sound. This is very much like mixing 50% of each sound at a mixing board. It’s also very much like the way that we blended pictures in Program 27 (page 160)!

Program 80: Blending two sounds

```java
/**
 * Method to overlap or blend two sounds. Start
 * by copying the first 20,000 samples from sound1 into
 * the current sound then copy the sum of half of sound1
 * and half of sound2 for the next 20,000 samples and
 * end with the next 20,000 samples from sound2.
 */

public void blendSounds() {
    Sound sound1 =
        new Sound(FileChooser.getMediaPath("aah.wav"));
    Sound sound2 =
        new Sound(FileChooser.getMediaPath("bassoon-c4.wav"));
    int value = 0;

    // copy the first 20,000 samples from sound1 into target
    for (int index=0; index < 20000; index++)
        this.sampleValueAt(index,
            sound1.getSampleValueAt(index));

    // copy the next 20,000 samples from sound1 and blend that
    // with the first 20,000 samples from sound2
    for (int index = 0; index < 20000; index++) {
        value = (int) ((sound1.getSampleValueAt(index + 20000) * 0.5) +
            (sound2.getSampleValueAt(index) * 0.5));
        this.sampleValueAt(index + 20000, value);
    }

    // copy the next 20,000 samples from sound2 into the target
    for (int index=20000; index < 40000; index++)
        this.sampleValueAt(index + 20000,
            sound2.getSampleValueAt(index));
}
```
Chapter 10  Combining and Creating Sounds

Common Bug: Set the media folder first!
If you’re going to write code that usesFileChooser.getMediaPath(String baseName), you’ll need to execute FileChooser.setMediaPath(String directory) first.

Like blending the picture (Program 27 (page 160)), there are loops in this function for each segment of the blended sound.

- We start by creating the sound1 and sound2 sounds for blending. The length of these sounds is over 40,000 samples, but we’re just going to use the first 40,000 as an example.

- In the first loop, we simply get 20,000 samples from sound1 and copy them into the current sound this. Notice that we’re not using a separate index variable for the target–instead, we’re using the same index variable, index, for both sounds since we are copying from 0 to 19,999 from sound1 into 0 to 19,999 in the current sound.

- In the next loop, we copy 20,000 samples from both sound1 and sound2 blended into the current sound. We get a sample from each of sound1 and sound2, then multiply each by 0.5 and add the results together. The result is a sample that represents 50% of each. Notice that we are using one index variable here as well but adding 20,000 to the value of that for determining the index of sound1 and the current sound. So we blend values from sound1 starting at index 20,000 and from sound2 starting at index 0 and the blended values go into the current sound starting at index 20,000.

- Finally, we copy another 20,000 samples from sound2. The result sounds like “Aah,” first, then half of each, then just a bassoon note. Notice that we start the index at 20,000 for the next place to copy from sound2. This means we need to add 20,000 to that value for the index in the current sound (since there are already 40,000 values in the current sound).

To create the blended sound first create a sound using the file that has 3 seconds of silence. Then explore it to see what it looks like before the blending. Next, blend the two sounds into the silent sound. Finally, explore the new sound.

> Sound target = new Sound(FileChooser.getMediaPath("sec3silence.wav"));
> target.explore();
> target.blendSounds();
> target.explore();

10.3 CREATING AN ECHO
Creating an echo effect is similar to the splicing recipe (Program 74 (page 311)) that we saw in the last chapter, but involves actually creating sounds that didn’t exist before. We do that by actually adding wave forms. What we’re doing here is adding
Section 10.3 Creating an Echo

FIGURE 10.2: The original “ahh” sound, the original bassoon note, and the blended sound

samples from a delay number of samples away into the sound, but multiplied by 0.6 so that they’re fainter.

```java
/**
 * Method to add an echo to a sound
 * @param delay the number of samples before the echo starts
 */
public void echo(int delay)
{
    // make a copy of the original sound
    Sound s = new Sound(this.getFileName());
    int value = 0;

    // loop from delay to end of sound
    for (int i = delay; i < this.getLength(); i++) {
        /* get the value back by delay samples from the
         * copy of the sound and make it fainter
         */
        value = (int) (s.getSampleValueAt(i - delay) * 0.6);

        /* set the value at the current index to the sum
         * of the current value and the echo
         */
    }
}
```

Program 81: Make a sound and a single echo of it
Chapter 10  Combining and Creating Sounds

```javascript
this.setSampleValueAt(i, 
  this.getSampleValueAt(i) + 
  value);
```

**How it works:** The `echo` function takes a delay: the number of samples before the echo starts. Try this with different amounts of delay. With low values of delay, the echo will sound more like **vibrato**. Higher values (try 10,000 or 20,000) will give you a real echo.

- This method creates a copy of the current sound `s`. This is where we’ll get the original, unadulterated samples for creating the echo. (You could try this without creating a copy to get some interesting layered echoes.)

- Next we declare a variable `value` to hold a value of a sample.

- Our loop starts with the index `i` being set to the passed delay and continues through the rest of the sound.

- The echoed sound is `delay` samples back, so `i-delay` is the sample we need. We multiply it by 0.6 to make it softer in volume.

- We then add the echoed sample to the current sample at `i` and set it in the current Sound object.

Try this method on sounds with words in them.

```javascript
> Sound sound = new Sound(FileChooser.getMediaPath("thisisatest.wav"));
> sound.explore();
> sound.echo(20000);
> sound.explore();
```

**FIGURE 10.3:** The original “This is a test” sound (left), and the sound with an echo (right)
10.3.1 Creating Multiple Echoes

This recipe actually lets you set the number of echoes that you get. You can generate some amazing effects like this.

```java
/**
 * Method to create multiple echoes of the current sound
 * @param delay the number of samples before the echo starts
 * @param numEchoes the number of echoes desired
 * @return a new sound with the echoes in it
 */
public Sound echo(int delay, int numEchoes)
{
    int soundLength = this.getLength();
    Sound echoSound = new Sound(numEchoes * delay + soundLength);
    int value = 0;
    int echoIndex = 0;
    int echoValue = 0;
    double echoAmplitude = 1; // to start

    // copy the original sound
    echoSound.splice(this, 0, soundLength, 0);

    /* loop starting with 1 to create the first echo at the right place and end then when == the number of echoes */
    for (int echoCount = 1; echoCount <= numEchoes; echoCount++)
    {
        // decrease the volume (amplitude) of the echo
        echoAmplitude = echoAmplitude * 0.6;

        // echo the whole sound
        for (int i = 0; i < soundLength; i++)
        {
            echoIndex = i + (delay * echoCount);
            echoValue = (int) (this.getSampleValueAt(i) * echoAmplitude);
            echoSound.setSampleValueAt(echoIndex, echoValue + echoSound.getSampleValueAt(echoIndex));
        }
    }
    return echoSound;
}
```

To try out this recipe create a `Sound` object and then invoke this method on the `Sound` object. Be sure to save the resulting `Sound`.

```java
> Sound sound = new Sound(FileChooser.getMediaPath("croak.wav"));
```
Chapter 10 Combining and Creating Sounds

> Sound echo = sound.echo(8000,5);
> echo.play();

10.4 HOW SAMPLING KEYBOARDS WORK

Sampling keyboards are keyboards that use recordings of sounds (e.g., pianos, harps, trumpets) to create music by playing those sound recordings in the desired pitch. Modern music and sound keyboards (and synthesizers) allow musicians to record sounds in their daily lives, and turn them into “instruments” by shifting the frequency of the original sounds. How do the synthesizers do it? It’s not really complicated. The interesting part is that it allows you to use any sound you want as an instrument.

Sampling keyboards use huge amounts of memory to record lots of different instruments at different pitches. When you press a key on the keyboard, the recording closest (in pitch) to the note you pressed is selected, then the recording is shifted to exactly the pitch you requested.

This first recipe works by creating a sound that skips every other sample. You read that right—after being so careful treating all the samples the same, we’re now going to skip half of them! In the mediasources directory, you’ll find a sound named c4.wav. This is the note C, in the fourth octave of a piano, played for one second. It makes a good sound to experiment with, though really, any sound will work.

Program 83: Double the frequency of a sound

```java
/**
 * Method to double the frequency of a sound by taking every second sample. The result will be a higher sound.
 */
public void doubleFreq()
{
    // make a copy of the original sound
    Sound s = new Sound(this.getFileName());

    /* loop through the sound and increment target index by one but source index by 2 and set target value to the copy of the original sound */
    for (int sourceIndex=0, targetIndex = 0;
         sourceIndex < this.getLength();
         sourceIndex=sourceIndex+2, targetIndex++)
    {
         this.setSampleValueAt(targetIndex, s.getSampleValueAt(sourceIndex));
    }

    // clear out the rest of this sound
    for (int i = this.getLength() / 2;
         i < this.getLength();
         i++)
    {
         this.setSampleValueAt(i, 0);
    }
```
Section 10.4  How Sampling Keyboards Work  329

Here’s how to use the double frequency method.

```java
i++)
    this.setSampleValueAt(i, 0);
}
```

> Sound s = new Sound(FileChooser.getMediaPath("c4.wav"));
> s.explore();
> s.doubleFreq();
> s.explore();

![FIGURE 10.4: The original sound (left), and the sound with the frequency doubled (right)](image)

This method starts like the other ones in this chapter by making a copy of the sound. Then it loops through the sound but it increments the index that keeps the position in the source sound `sourceIndex` by 2 and the index that keeps the position in the target sound `targetIndex` by 1. This will copy the sample value at `sourceIndex` 0 to `targetIndex` 0, then `sourceIndex` 2 to `targetIndex` 1, then `sourceIndex` 4 to `targetIndex` 2, and so on. Since the resulting sound will be half as long as it was the second loop just fills the rest of the sound with zeroes.

Try it! You’ll see that the sound really does double in frequency with the result that it sounds higher!

How did that happen? It’s not really all that complicated. Think of it this way–The frequency of the original sound is really the number of cycles that pass by in a certain amount of time. If you skip every other sample, the new sound has just as many cycles, but has them in half the amount of time!

Now let’s try the other way: Let’s take every sample twice! What happens then?

To do this, we need to use a cast to “throw away” the fractional part of a floating point number using a cast to integer. To cast a floating point number to an integer number use `(int)`.

> System.out.println((int)0.5)
0

\(^1\text{You are now trying this out as you read, aren’t you?}\)
Chapter 10  Combining and Creating Sounds

> System.out.println((int)1.5)

1

Here’s the recipe that halves the frequency. The for loop moves the targetIndex along the length of the sound. The sourceIndex is now being incremented—but only by 0.5! The effect is that we’ll take every sample in the source twice. The sourceIndex will be 1, 1.5, 2, 2.5, and so on, but because we’re using the (int) of that value, we’ll take samples 1, 1, 2, 2, and so on.

Program 84: Half the frequency

```java
/**
 * Method to halve the frequency of a sound by taking each sample twice. The result will be a lower sound.
 */
public void halveFreq()
{
    // make a copy of the original sound
    Sound s = new Sound(this.fileName);

    /* loop through the sound and increment target index
     * by one but source index by 0.5 and set target value
     * to the copy of the original sound
     */
    for (double sourceIndex = 0, targetIndex = 0;
         targetIndex < this.getLength();
         sourceIndex = sourceIndex + 0.5, targetIndex++)
        this.setSampleValueAt((int)targetIndex, s.getSampleValueAt((int)sourceIndex));
}
```

This method first creates a copy of the sound. Then it loops through the sound incrementing the sourceIndex by 0.5 and the targetIndex by 1. We get a sample value from source at the integer value using ((int)) of the sourceIndex. We set the target at the integer value using ((int)) of the targetIndex to the sample value that we got from the copy of the sound. We then add 0.5 to the sourceIndex. This means that the sourceIndex, each time through the loop, will take on the values 0.0, 0.5, 1.0, 1.5, 2.0, 2.5, and so on. But the integer part of this sequence is 0, 0, 1, 1, 2, 2, and so on. The result is that we take each sample from the source sound twice.

Think about what we’re doing here. Imagine that the 0.5 above were actually 0.75 or 3.0. Would this work? The for loop would have to change, but essentially the idea is the same in all these cases. We are sampling the source data to create the target data. Using a sample index of 0.5 slows down the sound and halves the frequency. A sample index larger than one speeds up the sound and increases the frequency.
Let’s try to generalize this sampling with the below recipe. (Note that this one won’t work right!)

Program 85: Changing the frequency of a sound: BROKEN!

```java
/**
 * Method to change the frequency of a sound by the
 * passed factor
 * @param factor the amount to increment the source
 * index by. A number greater than 1 will increase the
 * frequency and make the sound higher
 * while a number less than one will decrease the
 * frequency and make the sound lower.
 */
public void changeFreq1(double factor) {
    // make a copy of the original sound
    Sound s = new Sound(this.getFileName());

    /* loop through the sound and increment the target index
     * by one but increment the source index by the factor
     */
    for (double sourceIndex=0, targetIndex = 0;
        targetIndex < this.getLength();
        sourceIndex=sourceIndex+factor, targetIndex++)
    {
        this.setSampleValueAt((int) targetIndex,
            s.getSampleValueAt((int) sourceIndex));
    }
}
```

Here’s how we could use this:

```java
> s = new Sound(FileChooser.getMediaPath("c4.wav"));
> s.explore();
> s.changeFreq(0.75);
> s.explore();
```

That will work really well! But what if the factor for sampling is MORE than 1.0?

```java
>Sound hello = new Sound(FileChooser.getMediaPath("Elliot-hello.wav"));
> hello.changeFreq1();
You are trying to access the sample at index: 54759, but the last valid index is at 54757.
```

Why? What’s happening? Here’s how you could see it: Print out the sourceIndex just before the setSampleValueAt. You’d see that the sourceIndex
becomes *larger* than the source sound! Of course, that makes sense. If each time through the loop, we increment the `targetIndex` by 1, but we’re incrementing the `sourceIndex` by *more than one*, we’ll get past the end of the source sound before we reach the end of the target sound. But how do we avoid it?

Here’s what we want to happen: If the `sourceIndex` ever gets equal or larger than length of the source, we want to reset the `sourceIndex`—probably back to 0. The key word there is *if*, or even *if*.

We can *can* tell Java to make decisions based on a *test*. We use *if* as a statement to do something if a test is true. In our case, the test is `sourceIndex >= s.getLength()`. We can test on `<`, `>`, `==` (for equality), `!=` (for inequality, not-equals) and even `<=` and `>=`. An *if* statement takes a *block*, just as *while* and *for* do. The block defines the things to do if the *test* in the *if* statement is true. In this case, our block is simply `sourceIndex = 0;`. The block of statements is defined inside of an open curly brace `"` and a close curly brace `"`. If you just have one statement that you want to execute it doesn’t have to be in a block but it is better to keep it in a block.

The below recipe generalizes this and allows you to specify how much to shift the samples by.

---

### Program 86: Changing the frequency of a sound

```java
/**
 * Method to change the frequency of a sound
 * by the passed factor
 * @param factor the amount to increment the source
 * index by. A number greater than 1 will increase the
 * frequency and make the sound higher
 * while a number less than one will decrease the frequency
 * and make the sound lower.
 */
public void changeFreq(double factor) {
    // make a copy of the original sound
    Sound s = new Sound(this.getFileName());

    /* loop through the sound and increment the target index
     * by one but increment the source index by the factor
     */
    for (double sourceIndex=0, targetIndex = 0;
        targetIndex < this.getLength();
        sourceIndex=sourceIndex+factor, targetIndex++)
    {
        if (sourceIndex >= s.getLength()) {
            sourceIndex = 0;
        }
        this.setSampleValueAt((int) targetIndex,
            s.getSampleValueAt((int) sourceIndex));
    }
}
```

---
We can actually set the factor so that we get whatever frequency we want. We call this factor the \textit{sampling interval}. For a desired frequency \( f_0 \), the sampling interval should be:

\[
\text{samplingInterval} = \frac{\text{sizeOfSourceSound}}{\text{samplingRate}} f_0
\]

This is how a keyboard synthesizer works. It has recordings of pianos, voices, bells, drums, whatever. By \textit{sampling} those sounds at different sampling intervals, it can shift the sound to the desired frequency.

The last recipe of this section plays a single sound at its original frequency, then at two times, three times, four times, and five times the frequency. We need to use \texttt{blockingPlay} to let one sound finish playing before the next one starts. Try it with just \texttt{play} and you’ll hear the sounds collide as they’re generated faster than the computer can play them.

**Program 87: Playing a sound in a range of frequencies**

```java
/**
 * Method to play a sound 5 times and each time increase the frequency. It doesn’t change the original sound.
 */
public void play5Freq()
{
    Sound s = null;

    // loop 5 times but start with 1 and end at 5
    for (int i = 1; i < 6; i++)
    {
        // reset the sound
        s = new Sound(this.getFileName());

        // change the frequency
        s.changeFreq(i);

        // play the sound
        s.blockingPlay();
    }
}
```

To use this method try:

```java
> Sound s = new Sound(FileChooser.getMediaPath("c4.wav"));
> s.play5Freq();
```

This method loops with the value of \( i \) staring at 1 and ending before it is 6. This will loop 5 times. Why start at 1 instead of 0? What would happen if we used
a factor of 0 to change the frequency? We would end up with silence for the first sound.

10.4.1 Sampling as an Algorithm

You should recognize a similarity between the halving recipe Program 84 (page 330) and the recipe for scaling a picture up (larger) Program 30 (page 167). To halve the frequency, we take each sample twice by incrementing the source index by 0.5 and using the casting (\texttt{int}) to get the integer part of that. To make the picture larger, we take each pixel twice, by adding 0.5 to the source index variable and using the casting on that. These two methods are using the same \textit{algorithm}. The details of pictures vs. sounds aren’t critical. The point is that the same basic process is being used in each.

We have seen other algorithms that cross media boundaries. Obviously, our increasing red and increasing volume functions (and the decreasing versions) are essentially doing the same things. The way that we blend pictures or sounds is the same. We take the component color channels (pixels) or samples (sounds) and add them using percentages to determine the amount from each that we want in the final product. As long as the percentages total 100%, we’ll get a reasonable output that reflects the input sounds or pictures at the correct percentages.

Identifying algorithms like these are useful for several reasons. If we understand the algorithm in general (e.g., when it’s slow and when it’s fast, what it works for and what it doesn’t, what the limitations are), then the lessons learned apply in the specific picture or sound instances. The algorithms are also useful to know for designers. When you are designing a new program, you can keep in mind the algorithms so that you can use them when they apply.

When we double or half the sound frequency, we are also shrinking and doubling the length of the sound (respectively). You might want a target sound whose length is \textit{exactly} the length of the sound, rather than have to clear out extra stuff from a longer sound. You can do that with \texttt{new Sound(int lengthInSamples)}. \texttt{new Sound(44000)} returns a new empty sound of 44000 samples.

10.5 ADDITIVE SYNTHESIS

Additive synthesis creates sounds by adding sine waves together. We saw earlier that it’s really pretty easy to add sounds together. With additive synthesis, you can shape the waves yourselves, set their frequencies, and create “instruments” that never existed.

10.5.1 Making Sine Waves

Let’s figure out how to produce a set of samples to generate a sound at a given frequency and amplitude.

From trigonometry, we know that if we take the sine of the radians from 0 to \(2\pi\), we’ll get a circle. Spread that over time, and you get a sine wave. In other words, if you took values from 0 to \(2\pi\), computed the sine of each value, and graphed the computed values. You’d get a sine wave. From your really early math courses, you know that there’s an infinity of numbers between 0 and 1. Computers don’t
handle infinity very well, so we’ll actually only take some values between 0 to $2\pi$.

To create the below graph, Mark filled 20 rows (a totally arbitrary number) of a spreadsheet with values from 0 and $2\pi$ (about 6.28). Mark added about 0.314 (6.28/20) to each preceding row. In the next column, he took the sine of each value in the first column, then graphed it.

Now, if we want to create a sound at a given frequency, say 440 Hz. This means that we have to fit an entire cycle like the above into $1/440$ of a second. (440 cycles per second, means that each cycle fits into $1/440$ second, or 0.00227 seconds.) Mark made the above picture using 20 values. Call it 20 samples. How many samples do we have to chop up the 440 Hz cycle into? That’s the same question as: How many samples must go by in 0.00227 seconds? We know the sampling rate—that’s the number of samples in one second. Let’s say that it’s 22050 samples per second (our default sampling rate). Each sample is then $(1/22050)$ 0.0000453 seconds. How many samples fit into 0.00227? That’s $0.00227/0.0000453$, or about 50. What we just did here mathematically is:

$$\text{interval} = 1/\text{frequency}$$

$$\text{samplesPerCycle} = \frac{\text{interval}}{\text{samplingRate}} = (\text{samplingRate})(\text{interval})$$

Now, let’s spell this out in Java code. To get a waveform at a given frequency, say 440 Hz, we need 440 of these waves in a single second. Each one must fit into the interval of $1/\text{frequency}$. The number of samples that needs to be produced during the interval is the sampling rate divided by the frequency, or interval $(1/f) \ast (\text{samplingrate})$. Call that the samplesPerCycle.

At each entry of the sound sampleIndex, we want to:

- Get the fraction of sampleIndex/samplesPerCycle.
- Multiply that fraction by $2\pi$. That’s the number of radians we need. Take the \text{sin} of (sampleIndex/samplesPerCycle) $\ast 2\pi$.
- Multiply the result by the desired amplitude, and put that in the sample value at sampleIndex.

### 10.5.2 Creating Sounds Using Static Methods

All of the methods that we have written work with an existing object. But, if the method we want to write this time creates the object there won’t be a current object to modify. How can we invoke this method if we don’t have an object of the class to invoke it on? We can use a class method. A class method is also called a static method.
Class methods can be called using `ClassName.methodName(parameterList)`. Recall that when a class definition is compiled it creates a file (`ClassName.class`) that contains all the information in the class definition in a form that the computer can understand. When you use a class name for the first time in your code or in the interactions pane Java looks for that file and loads the definition of that class. It also creates an object that represents that class.

We have been using `new Class(parameterList)` to create new objects of the class by asking that object that represents the class to create an object for us. Now we will use `ClassName.methodName(parameterList)` to ask the object that represents the class to create an object of that class and return it.

How do we make a method a class method? We add the keyword `static` to it, usually after the visibility. Now when you hear “static” you may think of noise on your radio or that something won’t move. But, in Java “static” means something that exists in the object that represents the class. So, why not use “class” instead of “static”? The keyword `class` is used in Java to define new classes. But, when you see the keyword `static` think `class`. So, the keyword `static` in a method declaration tells you that the method is a “class” method.

To build sounds, there are some silent sounds in the media sources. Our sine wave generator will use the one second of silence sound to build a sine wave of one second. We’ll provide an amplitude as input—that will be the maximum amplitude of the sound. (Since sine generates between −1 and 1, the range of amplitudes will be between `−amplitude` and `amplitude`.)

Program 88: Generate a sine wave at a given frequency and amplitude

```java
/** 
 * Method to create a one second sine wave sound with the 
 * given frequency and maximum amplitude 
 * @param freq the desired frequency 
 * @param maxAmplitude the maximum amplitude 
 * @return the new sound 
 */
public static Sound createSineWave(int freq, int maxAmplitude) {
    Sound s = new Sound(FileChooser.getMediaPath("sec1silence.wav"));
    double samplingRate = s.getSamplingRate();
    double rawValue = 0;
    int value = 0;
    double interval = 1.0 / freq; // length of cycle in seconds
    double samplesPerCycle = interval * samplingRate;
    double maxValue = 2 * Math.PI;

    // loop through the length of the sound
    for (int i = 0; i < s.getLength(); i++) {
        // calculate the value between −1 and 1
```
Section 10.5 Additive Synthesis

```
rawValue = Math.sin((i / samplesPerCycle) * maxValue);

// multiply by the desired max amplitude
value = (int)(maxAmplitude * rawValue);

// set the value at this index
s.setSampleValueAt(i, value);
```}

Notice that this method creates an object of the Sound class and returns it. In order to be able to refer to this Sound object again be sure to set a variable to refer to it. We can invoke any class method using ClassName.methodName(parameterList). Let’s build a sine wave of 880 Hz at an amplitude of 4000.

```java
> Sound s = Sound.createSineWave(880,4000);
> s.explore();
```

**FIGURE 10.5:** The sine wave with a frequency of 880 and a maximum amplitude of 4000

### 10.5.3 Adding Sine Waves Together

Now, let’s add sine waves together. Like we said at the beginning of the chapter, that’s pretty easy: Just add the samples at the same indices together. Here’s a function that adds one sound into a second sound.

```
/**
 * Method to add the passed sound to this sound
 */
```

Program 89: Add two sounds together
Chapter 10  Combining and Creating Sounds

```java
* @param source the sound to combine with this one
*/
public void add(Sound source)
{
    int value = 0;  // holder for new value
    // loop through all of the source
    for (int i = 0; i < source.getLength(); i++) {
        // add source sound value and this sound value
        value = this.getSampleValueAt(i) + source.getSampleValueAt(i);
        // set the value in this sound to the new value
        this.setSampleValueAt(i, value);
    }
}
```

How are we going to use this function to add together sine waves? We need both of them at once? Turns out that it’s easy:

Let’s add together 440 Hz, 880 Hz (twice 440), and 1320 Hz (880+440), and we’ll increase the amplitudes. We’ll double the amplitude each time: 2000, then 4000, then 8000. We’ll add them all up into a sound called `sound440`. At the end, we generate a 440 Hz sound so that we can listen to them both and compare.

```java
> Sound s440 = Sound.createSineWave(440,2000);
> Sound s880 = Sound.createSineWave(880,4000);
> Sound s1320 = Sound.createSineWave(1320,8000);
> s440.add(s880);
> s440.add(s1320);
> s440.explore();
> Sound orig440 = Sound.createSineWave(440,2000);
> orig440.explore();
```

10.5.4 Checking our Result

How do we know if we really got what we wanted? We can test our code by using the sound tools in the MediaTools. First, we save out a sample wave (just 440 Hz) and the combined wave.

```java
> orig440.write(FileChooser.getMediaPath("just440.wav"));
> s440.write(FileChooser.getMediaPath("combined440.wav"));
```

Common Bug: Beware of adding amplitudes past 32767

When you add sounds, you add their amplitudes, too. A maximum of 2000+4000+8000 will never be greater than 32767, but do worry about that. Remember what happened when the amplitude got too high last chapter...
Open up each of these in turn in the sound editor. Right away, you’ll notice that the wave forms look very different (Figure 10.6). That tells you that we did something to the sound, but what?

The way to really check your additive synthesis is with a fast fourier transform (FFT). Generate the FFT for each signal. You’ll see that the 440 Hz signal has a single spike (Figure 10.7). That’s what you’d expect—it’s supposed to be a single sine wave. Now, look at the combined wave form’s FFT (Figure 10.8). It’s what it’s supposed to be! You see three spikes there, and each succeeding one is double the height of the last one.

10.5.5 Square Waves

We don’t have to just add sine waves. We can also add square waves. These are literally square-shaped waves, moving between +1 and −1. The FFT will look very different, and the sound will be very different. It can actually be a much richer sound.
Try this method instead of the sine wave generator and see what you think. Note the use of an if statement to swap between the positive and negative sides of the wave half-way through a cycle.

Program 90: Square wave generator for given frequency and amplitude

```java
/**
 * Method to generate a 1 second sound with square waves
 * with the passed frequency and maximum amplitude.
 * @param freq the desired frequency
 * @param maxAmplitude the maximum amplitude
 * @return the created sound
 */
public static Sound createSquareWave(int freq, int maxAmplitude) {
    Sound s = new Sound(chooser.getMediaPath("sec1silence.wav"));
    double samplingRate = s.getSamplingRate();
    double rawValue = 0;
    int value = 0;
    double interval = 1.0 / freq; // length of cycle in seconds
    double samplesPerCycle = interval * samplingRate;
    double samplesPerHalfCycle = (int) (samplesPerCycle / 2);
    double maxValue = 2 * Math.PI;

    // loop through the length of the sound
    for (int soundIndex = 0, sampleCounter = 0;
         soundIndex < s.getLength();
         soundIndex++, sampleCounter++) {
        // check if first half of cycle
        if (sampleCounter < samplesPerHalfCycle) {
            value = maxAmplitude;
        } else {
            // make the value negative
            value = maxAmplitude * -1;
            /* if the sample counter is greater than the
             * samples per cycle reset it to 0
             */
            if (sampleCounter > samplesPerCycle) {
                sampleCounter = 0;
            }
        }
        // set the value
        s.setSampleValueAt(soundIndex, value);
    }
```

Section 10.5 Additive Synthesis

```java
return s;
}
```

Use it like this:

```java
> Sound sq440 = Sound.createSquareWave(440,4000);
> sq440.play();
> Sound sq880 = Sound.createSquareWave(880,8000);
> Sound sq1320 = Sound.createSquareWave(1320,10000);
> sq440.write(FileChooser.getMediaPath("square440.wav"));
> sq440.add(sq880);
> sq440.add(sq1320);
> sq440.play();
> sq440.write(FileChooser.getMediaPath("squareCombined.wav"));
```

You'll find that the waves (in the wave editor of MediaTools) really do look square (Figure 10.9), but the most amazing thing is all the additional spikes in FFT (Figure 10.10). Square waves really do result in a much more complex sound.

![Figure 10.9: The 440 Hz square wave (top) and additive combination of square waves (bottom)](image1)

![Figure 10.10: FFT’s of the 440 Hz square wave (top) and additive combination of square waves (bottom)](image2)
10.5.6 Triangle Waves

Try triangle waves instead of square waves with this recipe.

Program 91: Generate triangle waves

```java
/**
 * Method to create a one second triangle wave sound
 * with the given frequency and maximum amplitude
 * @param freq the desired frequency
 * @param maxAmplitude the maximum amplitude
 * @return the new sound
 */
public static Sound createTriangleWave(int freq, int maxAmplitude)
{
    Sound s =
        new Sound(FileChooser.getMediaPath("sec1silence.wav"));
    double samplingRate = s.getSamplingRate();
    int value = 0;
    double interval = 1.0 / freq; // length of cycle in seconds
    double samplesPerCycle = interval * samplingRate;
    int samplesPerQuarterCycle =
        (int)(samplesPerCycle / 4);
    int increment =
        (int)(maxAmplitude / samplesPerQuarterCycle);

    // loop through the length of the sound
    for (int soundIndex = 0;
        soundIndex < s.getLength();
        soundIndex++, value = value + increment)
    {
        // check if the value is equal to the desired max
        if (value >= maxAmplitude ||
            value <= maxAmplitude * -1)
        {
            increment = increment * -1;
            value = value + increment;
        }

        // set the sample value
        s.setSampleValueAt(soundIndex, value);
    }
    return s;
}
```

Since this is a class method it can be invoked using

```
ClassName.methodName(parameterList).
```

> Sound triangle = Sound.createTriangleWave(440,4000);
> triangle.play();
> triangle.explore();

You have seen other class (static) methods. What about Math.abs(int number), FileChooser.pickAFile(), and ColorChooser.pickAColor()? All of these are class methods which is why you can invoke them using ClassName.methodName(parameterList).

Class methods are useful for general methods that don’t require an object of the class to be created (Math.abs or for methods that create objects of the class createTriangleWave).

10.6 MODERN MUSIC SYNTHESIS

Additive synthesis is how early music synthesizers work. Nowadays, additive synthesis isn’t too common because the sounds that it generates aren’t natural sounding. Synthesizing from recorded sounds is quite common, but it isn’t pure synthesis in the sense of creating sounds out of nothing.

The most common synthesis technique today is probably FM synthesis or frequency modulation synthesis. In FM synthesis, an oscillator (a programmed object that generates a regular series of outputs) controls (modulates) frequencies with other frequencies. The result is a richer sound, less tinny or computer sounding.

Another common technique is subtractive synthesis. In subtractive synthesis, out-and-out noise is used as the input, and then filters are applied to remove unwanted frequencies. The result is, again, a richer sound, though typically not as rich as FM synthesis.

Why would we want to create sounds or music with computers anyway? What’s the point when there are lots of great sounds, music, and musicians in the world? The point is that if you want to tell someone else how you got that sound, so that they could replicate the process, or even modify the sound in some way (perhaps making it better), a program is the way to do it. A program succinctly captures and communicates a process—how a sound or piece of music is generated.

10.6.1 MP3

Nowadays, the most common kind of audio file that you have on your computer is probably MP3 files (or perhaps MP4 or one of its related or descendant file types). MP3 files are sound (and video, in some cases) encodings based on the MPEG-3 standard. They are audio files, but compressed in special ways.

One way in which MP3 files are compressed is called lossless compression. As we know, there are techniques for storing data that use fewer bits. For example, we know that every sample is typically two bytes wide. What if we didn’t store every sample, but instead stored the difference from the last sample to the current sample? The difference between samples is usually much smaller than 32,767 to -32,768—it might be +/- 1000. That takes fewer bits to store.

But MP3 also uses lossy compression. It actually throws away some of the sound information. For example, if there’s a really soft sound immediately after or
simultaneous with a really loud sound, you won’t be able to hear the soft sound. A
digital recording keeps all those frequencies. MP3 throws away the ones you can’t
actually hear.

WAV files are kind of compressed, but not as much as MP3, and they only
use lossless techniques. Some WAV files use MP3 compression which makes them
really MP3 files. MP3 files tend to be much smaller than the same sound in a WAV
format. AIFF files are similar to WAV files.

10.6.2 MIDI

MIDI is the Musical Instrument Digital Interface. It’s really a set of agreements
between manufacturers of computer music devices (sequencers, synthesizers, drum
machines, keyboards, etc.) for how their devices will work together. Using MIDI,
you can control various synthesizers and drum machines from different keyboards.

MIDI doesn’t really record what something sounds like, instead it encodes
how it is played. Literally, MIDI encodes information like “Press the key down on
synthesized instrument X at pitch Y” then later “Release the key Y on instrument
X.” The quality of MIDI sound depends entirely on the synthesizer, the device
generating the synthesized instrument.

MIDI files tend to be very small. Instructions like “Play key #42 on track
7” are only some five bytes long. This makes MIDI attractive in comparison with
large sound files. MIDI has been particularly popular for karaoke machines.

MIDI has an advantage over MP3 or WAV files in that they can specify a
lot of music in very few bytes. But MIDI can’t record any particular sound. For
example, if you want to record a particular person’s style of playing an instrument,
or record anyone singing, you don’t want to use MIDI. To capture actual sounds,
you need to record the actual samples, so you’ll need MP3 or WAV.

Most modern operating systems have pretty good synthesizers built into them.
We can actually use them from Java. We have created a class MidiPlayer that has
a method playNote that takes as input a note as a number and a duration (how
long to play the sound) in milliseconds. The note numbers correspond to keys, not
to frequencies. C in the first octave is 1, C# is 2. C in the fourth octave is 60, D is
62, and E is 64. See http://www.harmony-central.com/MIDI/Doc/table2.html for
more information on the note numbers. If you don’t specify what instrument you
want to play the note on it will simulate a piano.

Here’s a simple example of playing some MIDI notes from DrJava.

> MidiPlayer player = new MidiPlayer();
> player.playNote(62,250); // d quarter note
> player.playNote(60,500); // c half note

The 250 and 500 specify the number of milliseconds to play the note. If you
want a measure to take one second (1000 milliseconds) then a quarter note would
be 250 milliseconds and a half note would be 500 milliseconds.

We can write a method to play a song. How about writing a method to play
part of Jingle Bells? Here is a method that plays the first 4 measures from it. Put
this in the MidiPlayer class (before the ending curly brace) and compile it.

```
/**
 * Method to play the first 4 measures of Jingle Bells
 * with each measure taking 1000 milliseconds (1 second)
 * this is 2/4 time
 */
public void playJingleBells4()
{
  // measure 1
  playNote(52, 250);  // e eighth note
  playNote(60, 250);  // c eighth note
  playNote(58, 250);  // b flat eighth note
  playNote(56, 250);  // a flat eighth note

  // measure 2
  playNote(52, 500);  // e quarter note
  rest(250);          // rest
  playNote(52, 125);  // e sixteenth note
  playNote(52, 125);  // e sixteenth note

  // measure 3
  playNote(52, 500);  // e eighth note
  playNote(60, 250);  // c eighth note
  playNote(58, 250);  // b flat eighth note
  playNote(56, 250);  // a flat eighth note

  // measure 4
  playNote(53, 1000); // f half note
}
```

This method only plays the first 4 measures of Jingle Bells. This may not sound like “Jingle Bells” to you since I am using the original version first published in 1859 by James Pierpont.

To play this using the default piano sounding instrument do the following:

```
> MidiPlayer player = new MidiPlayer();
> player.playJingleBells4();
```

You can change the instrument that you want to use to play the notes using the method setInstrument(int num) where the num is a number from 0 to 127 that maps to an instrument. We have create constants for some of the instrument numbers as you can see at the top of the MidiPlayer class definition. To play the first 4 measures of Jingle Bells on a flute do the following:

```
> MidiPlayer player = new MidiPlayer();
```
Music often repeats. In *Jingle Bells* the first verse is played and then the refrain. Next, the second verse is played and then the refrain is played again. The first verse and second verse are a bit different, but many of the measures in the two verses are the same. If we want to write a method that plays the first two verses of *Jingle Bells* with each verse followed by the refrain, we could put all the measures in it for both verses and the refrains, but then it would be very long (67 measures). Another option is to pull out the measures in the refrain and make a method that just plays the refrain and then call that method from the one that plays *Jingle Bells*.

Does this new method that plays just the refrain need to be public? If we don’t think any other class will need access to the new method we can make it private. Private methods can only be invoked from code in the class that they are declared in. You will get an error if you try to invoke a private method in code that is in another class.

To declare a method to be private use the keyword `private` for the visibility. Remember that to declare a method you must specify:

```
visibility returnType methodName( parameterList )
```

The following method is a private method that will play the refrain of *Jingle Bells*.

```
/**
 * Method to play refrain of Jingle Bells
 */
private void playJingleBellsRefrain()
{
    // measure 1
    playNote(60,250);  // c eighth note
    playNote(60,250);  // c eighth note
    playNote(60,500);  // c quarter note

    // measure 2
    playNote(63,250);  // e flat eighth note
    playNote(63,250);  // e flat eighth note
    playNote(63,500);  // e flat quarter note

    // measure 3
    playNote(60,250);  // c eighth note
    playNote(60,250);  // c eighth note
    playNote(65,375);  // f dotted eighth note
    playNote(65,125);  // f sixteenth note
```

Program 93: Playing the refrain
// measure 4
playNote(64,1000); // e half note

// measure 5
playNote(65,250); // f eighth note
playNote(61,250); // d flat eighth note
playNote(56,250); // a flat eighth note
playNote(64,250); // f eighth note

// measure 6
playNote(63,250); // e flat eighth note
playNote(60,250); // c eighth note
playNote(56,250); // a flat eighth note
playNote(56,125); // a flat sixteenth note
playNote(58,125); // b flat sixteenth note

// measure 7
playNote(60,250); // c eighth note
playNote(58,250); // b flat eighth note
playNote(56,250); // a flat eighth note
playNote(58,250); // b flat eighth note

// measure 8
playNote(60,1000); // c half note

// measure 9
playNote(60,250); // c eighth note
playNote(60,250); // c eighth note
playNote(60,500); // c quarter note

// measure 10
playNote(63,250); // e flat eighth note
playNote(63,250); // e flat eighth note
playNote(63,500); // e flat quarter note

// measure 11
playNote(60,250); // c eighth note
playNote(60,250); // c eighth note
playNote(65,250); // f eighth note
playNote(65,250); // f eighth note

// measure 12
playNote(64,1000); // e half note

// measure 13
playNote(53,250); // f eighth note
playNote(61,250); // d flat eighth note
playNote(60,250); // c eighth note
playNote(58,250); // b flat eighth note

// measure 14
Chapter 10 Combining and Creating Sounds

playNote(56,250); // a flat eighth note
playNote(63,250); // e flat eighth note
playNote(62,250); // d eighth note
playNote(63,125); // e flat sixteenth note
playNote(63,125); // e flat sixteenth note

// measure 16
playNote(55,250); // f eighth note
playNote(63,250); // e flat eighth note
playNote(61,250); // d flat eighth note
playNote(58,250); // b flat eighth note

// measure 17
playNote(56,500); // a flat quarter note
rest(500); // rest

Here is a method that will play the first verse of Jingle Bells.

Program 94: Playing the first verse

/**
 * Method to play the first verse of jingle bells
 * with each measure taking 1000 milliseconds (1 second)
 * It is in 2/4 time
 */
private void playJingleBellsV1()
{
    // measure 1
    playNote(52,250); // e eighth note
    playNote(60,250); // c eighth note
    playNote(58,250); // b flat eighth note
    playNote(56,250); // a flat eighth note

    // measure 2
    playNote(52,500); // e quarter note
    rest(250); // rest
    playNote(52,125); // e sixteenth note
    playNote(52,125); // e sixteenth note

    // measure 3
    playNote(52,500); // e eighth note
    playNote(60,250); // c eighth note
    playNote(58,250); // b flat eighth note
    playNote(56,250); // a flat eighth note

    // measure 4
    playNote(53,1000); // f half note
Section 10.6 Modern Music Synthesis 349

// measure 5
playNote(53,250); // f eighth note
playNote(61,250); // d flat eighth note
playNote(60,250); // c eighth note
playNote(58,250); // b flat eighth note

// measure 6
playNote(55,1000); // g half note

// measure 7
playNote(55,250); // g eighth note
playNote(65,250); // f eighth note
playNote(63,250); // e flat eighth note
playNote(61,250); // d flat eighth note

// measure 8
playNote(60,1000); // c half note

// measure 9
playNote(52,250); // e eighth note
playNote(60,250); // c eighth note
playNote(58,250); // b flat eighth note
playNote(56,250); // a flat eighth note

// measure 10
playNote(52,1000); // e half note

// measure 11
playNote(52,250); // e eighth note
playNote(60,250); // c eighth note
playNote(58,250); // b flat eighth note
playNote(56,250); // a flat eighth note

// measure 12
playNote(53,1000); // f half note

// measure 13
playNote(53,250); // f eighth note
playNote(61,250); // d flat eighth note
playNote(60,250); // c eighth note
playNote(58,250); // b flat eighth note

// measure 14
playNote(55,250); // g eighth note
playNote(63,250); // e flat eighth note
playNote(62,250); // d eighth note
playNote(63,250); // e flat eighth note

// measure 15
playNote(65,250); // f eighth note
playNote(63,250); // e flat eighth note
Here is a method that will play the second verse of *Jingle Bells*.

**Program 95: Playing the second verse**

```java
/** *
 * Method to play the second verse of *jingle bells*
 * with each measure taking 1000 milliseconds (1 second)
 * It is in 2/4 time
 */
private void playJingleBellsV2() {

    // measure 1
    rest(750);
    playNote(52, 250); // e eighth note

    // measure 2
    playNote(52, 250); // e eighth note
    playNote(60, 250); // c eighth note
    playNote(58, 250); // b flat eighth note
    playNote(56, 250); // a flat eighth note

    // measure 3
    playNote(52, 500); // e quarter note
    rest(250); // rest
    playNote(52, 125); // e sixteenth note
    playNote(52, 125); // e sixteenth note

    // measure 4
    playNote(52, 500); // e eighth note
    playNote(60, 250); // c eighth note
    playNote(58, 250); // b flat eighth note
    playNote(56, 250); // a flat eighth note

    // measure 5
    playNote(53, 750); // f dotted quarter note
    playNote(53, 250); // f eighth note

    // measure 6
    playNote(53, 250); // f eighth note
    playNote(61, 250); // d flat eighth note
}
```
Section 10.6 Modern Music Synthesis

playNote(60, 250); // c eighth note
playNote(58, 250); // b flat eighth note

// measure 7
playNote(55, 750); // g dotted quarter note
playNote(55, 250); // g eighth note

// measure 8
playNote(55, 250); // g eighth note
playNote(65, 250); // f eighth note
playNote(63, 250); // e flat eighth note
playNote(61, 250); // d flat eighth note

// measure 9
playNote(60, 750); // c dotted quarter note
playNote(52, 250); // e eighth note

// measure 10
playNote(52, 250); // e eighth note
playNote(60, 250); // c eighth note
playNote(58, 250); // b flat eighth note
playNote(56, 250); // a flat eighth note

// measure 11
playNote(52, 750); // e dotted quarter note
playNote(52, 250); // e eighth note

// measure 12
playNote(52, 250); // e eighth note
playNote(60, 250); // c eighth note
playNote(58, 250); // b flat eighth note
playNote(56, 250); // a flat eighth note

// measure 13
playNote(53, 750); // f dotted quarter note
playNote(53, 250); // f eighth note

// measure 14
playNote(53, 250); // f eighth note
playNote(61, 250); // d flat eighth note
playNote(60, 250); // c eighth note
playNote(58, 250); // b flat eighth note

// measure 15
playNote(55, 250); // g eighth note
playNote(63, 250); // e flat eighth note
playNote(62, 250); // d eighth note
playNote(63, 250); // e flat eighth note

// measure 16
playNote(65, 250); // f eighth note
Here is a method that will play the first two verses of Jingle Bells with each verse followed by the refrain write a method that calls the other methods.

```java
public void playJingleBells()
{
    // play verse 1
    playJingleBellsV1();

    // play refrain
    playJingleBellsRefrain();

    // play verse 2
    playJingleBellsV2();

    // play refrain
    playJingleBellsRefrain();
}
```

The advantage to breaking methods into smaller reusable methods is that if there is an mistake in one of the smaller methods it is easier to find and fix. If I had created one very long method with all the measures in it I probably would have copied and pasted the measures in the refrain. If I had to fix notes in the refrain I would have to fix them in both copies of it. By pulling out repeated code I have only one copy of it to fix and check for errors.

### 10.7 CONCEPTS SUMMARY

In this chapter we introduced class methods, private methods, and building methods from other methods.

#### 10.7.1 Class Methods

Class methods are general methods like `Math.abs(int num)` or methods that create objects like `Sound.createTriangleWave(int freq, int maxAmplitude)`. Class
methods do not work on object data (no implicit this current object is passed to the method).

Class methods are defined with the keyword static on the method declaration: \texttt{visibility static returnType methodName(parameterList)}.

\begin{verbatim}
public static Sound createTriangleWave(int freq, int maxAmplitude)
\end{verbatim}

You can invoke a class method using \texttt{ClassName.methodName(parameterList)}. To invoke the class method \texttt{createTriangleWave} on the class \texttt{Sound} and then save a reference to the new \texttt{Sound} object in a variable called \texttt{triangle} do the following:

\begin{verbatim}
> Sound triangle = Sound.createTriangleWave(440,4000);
\end{verbatim}

If the method doesn't use object data then it can be a class method. If a method does use object data then it should be an object method.

10.7.2 Private Methods

Private methods can only be invoked from code in the same class that they are declared in. Use private methods to break down long public methods into smaller reusable parts.

To declare a method to be private use the \texttt{private} keyword for the visibility. Remember that to declare a method you can use \texttt{visibility [static] returnType methodName(parameterList)}. The \texttt{static} keyword is optional and is used for declaring class methods.

\begin{verbatim}
private void playJingleBellsRefrain()
\end{verbatim}

If you try to access a private method from code outside the class the method is defined in you will get an error.

\begin{verbatim}
> MidiPlayer player = new MidiPlayer();
> player.playJingleBellsRefrain();
java.lang.IllegalArgumentException: Class koala.dynamicjava.interpreter.EvaluationVisitor can not access a member of class MidiPlayer with modifiers "private" at sun.reflect.Reflection.ensureMemberAccess(Reflection.java:57) at java.lang.reflect.Method.invoke(Method.java:317)
\end{verbatim}

10.7.3 Build a Program from Multiple Methods

You can build a program (method) from several methods. If you want to play a whole song like \texttt{Jingle Bells} you can break this into smaller reusable methods.

\begin{verbatim}
/**
 * Method to play Jingle Bells
 */
public void playJingleBells()
{
    // play verse 1
\end{verbatim}
Chapter 10 Combining and Creating Sounds

```java
playJingleBellsV1();
    // play refrain
playJingleBellsRefrain();
    // play verse 2
playJingleBellsV2();
    // play refrain
playJingleBellsRefrain();
}
```

Long methods are hard to read and understand. Try to break down methods into smaller parts.

**PROBLEMS**

10.1. Rewrite the echo function (Program 81 (page 325)) to generate two echoes back, each delay samples previous. Hint: Start your index loop at 2*delay + 1, then access one echo sample at index-delay and another at index - 2*delay.

10.2. How long is a sound compared to the original when it’s been frequency doubled (Program 83 (page 328))?  

10.3. Hip-hop DJ’s move turntables so that sections of sound are moved forwards and backwards quickly. Try combining backwards play (Program 78 (page 316)) and frequency shifting (Program 83 (page 328)) to get the same effect. Play a second of a sound quickly forward, then quickly backward, two or three times. (You might have to move faster than just double the speed.)

10.4. Consider changing the if block in the frequency shift recipe (Program 86 (page 332)) to `sourceIndex = sourceIndex - getLength(source)`. What’s the difference from just setting the `sourceIndex` to 1? Is this better or worse? Why?

10.5. If you use the shifting recipe (Program 86 (page 332)) with a factor of 2.0 or 3.0, you’ll get the sound repeated or even triplicated. Why? Can you fix it? Write `shiftDur` that takes a number of samples (or even seconds) to play the sound.

10.6. Change the `shift` function in Program 86 (page 332) to `shiftFreq` which takes a frequency instead of a factor, then plays the given sound at the desired frequency.

10.7. Using the sound tools, figure out the characteristic pattern of different instruments. For example, pianos tend to have a pattern the opposite of what we created—the amplitudes decrease as we get to higher sine waves. Try creating a variety of patterns and see how they sound and how they look.

10.8. When musicians work with additive synthesis, they will often wrap envelopes around the sounds, and even around each added sine wave. An envelope changes the amplitude over time: It might start out small, then grow (rapidly or slowly), then hold at a certain value during the sound, and then drop before the sound ends. That kind of pattern is sometimes called the attack-sustain-decay (ASD) envelope. Pianos tend to attack quickly then decay quickly. Flutes tend to attack slowly and sustain as long as you want. Try implementing that for the sine and square wave generators.

10.9. Write a method to play a simple song like *Jingle Bells*. The song should have at least one repeating part. Make sure that you create a private method for the repeating part.
TO DIG DEEPER

Good books on computer music will talk a lot about creating sounds from scratch like in this chapter. One of my favorites for understandability is *Computer Music: Synthesis, Composition, and Performance* by Dodge and Jerse [8]. The bible of computer music is Curtis Roads’ massive *The Computer Music Tutorial* [22].

One of the most powerful tools for playing with this level of computer music is **CSound**. It’s a software music synthesis system, free, and totally cross-platform. The book by Richard Boulanger [5] has everything you need for playing with CSound.

**jMusic** is a free programming library written for musicians in Java (see [http://jmusic.ci.qut.edu.au/jmtutorial/t1.html](http://jmusic.ci.qut.edu.au/jmtutorial/t1.html)). It allows you to compose music. It can also import and export MIDI and audio files.
Bibliography


Index

==, 332

greater-than-or-equals, 332

abstraction, 62, 63
ACM Turing Award, 17
acuity, 83
algebra, 46
algorithm, 9, 12, 294, 317, 334
   definition, 12
   mirroring, 317
alpha channel, 86
American Standard Code for Information Interchange (ASCII), 14
amplitude, 265
analog, 16
analog-to-digital conversion, 270
analog-to-digital conversion (ADC), 270
appending, 30
applets, 20
arguments, 64
array, 44, 274
   access elements, 81
   copying, 311
   definition, 81
   element, 274
   index, 274
   notation, 306
artificial intelligence, 9
ASCII, 14
attack-sustain-decay (ASD) envelope, 354
AutoCAD, 234

background subtraction, 210
base file name, 66
binary, 13
binary number system, 13
binding, 37
bit, 13
bitmap, 80
bitmap graphical representations, 234
blend, 159, 323
blending
   pictures, 159
Index

sounds, 323
block, 57, 332
blue, 15
blurring, 207
BMP, 234
body of method, 57
boxing, 46
Brando, Marlon, 284
byte, 13, 44
definition, 44

C, 11
calculus, 17
casting, 27, 107
changeVolume, 293
channel, 85
chroma key, 214
class, 20, 43
fully qualified name, 91
class fields
accessing, 91
class method, 335
class methods, 46
definition, 46
class variables
accessing, 91
clipped, 273
clipping, 255, 273, 296
CMYK color model, 85
code, 44
collage, 154
color
sepia, 200
color replacement, 187
ColorChooser.pickAColor(), 92
colorObj.brighter(), 92
colorObj.darker(), 92
commands, 74
compile, 23, 101
compiling
definition, 23
compressed, 45, 81, 87
compression, 81, 234
compressions, 265
computational recipe, 8
computer, 13
computer hardware, 270
computer music, 10
constant, 252
constants, 10, 133
context, 58
coordinates, 82
copying, 147
cropping, 151
CSound, 355
cycle, 265
cycles per second, 266
data, 44
data representation, 15
data structures, 9, 15
databases, 9
debugging
  using print statements, 145
debugging, 290, 296
  using print statements, 145
decibels, 266
decimal number, 13
definitions pane, 23
Digital, 16
Digital media, 16
digital-to-analog conversion, 271
digital-to-analog conversion (DAC), 270
digitization, 16
digitizing media
  sounds, 270
digitizing media
  pictures, 83
  why?, 16
directory, 45, 66
disk, 45
double quote, 29
drawing tool, 235
DrJava, 21
  definitions pane, 23
  interactions pane, 23
  files pane, 22
  installing, 21
  running slowly, 22
  starting, 21
Dynabook, 18
edge detection, 194
editor, 21
emergent properties, 10
encoding, 8, 13, 66
envelopes, 354
equal temperament, 267
equality, 332
equals-to, 332
error
    illegal start of expression, 57
    semicolon expected, 57
evaluation, 63, 73
explore(), 89
expression, 25

Fast Fourier Transform, 269
FFT, 269
file, 45
file extension, 66
FileChooser.getMediaPath(fileName), 145, 147
files pane, 22
fillPolygon(int[] xArray, int[] yArray, int numPoints), 228
filters, 343
Flash, 234
floating point
typing, 27
floating point number, 18, 44
FM synthesis, 343
font, 235
    how defined, 234
for loop, 122
Fourier transform, 269
frequency, 265, 266, 271
frequency domain, 269
frequency modulation synthesis, 343
full file name, 65, 66
fully qualified name, 91
function
    arguments, 63
    input values, 63
    input variables, 63
    parameters, 63
function calls, 73
functions, 46
fundamental, 267

garbage collection, 36
general, 116
getColor(), 90
getHeight(), 89
getPixel(x,y), 89
getPixels(), 89
getRed(), 90
getWidth(), 89
getX(), 89
getY(), 89
GIF, 234
going digital, 16
graphics, 10
grayscale, 127
greater-than, 332
green, 15

hard disk, 45
hardware, 270
Hertz, 266
hierarchical decomposition, 118
hierarchy, 118
HSB, 84
HSV color model, 84
HTML, 14
human-computer interface, 9

if, 332, 340
immutable, 48
implicitly, 58
import, 91
    how to, 91
    why, 91
infinite loop, 99, 100, 162
    stopping one, 99
Ingalls, Dan, 320
inherit, 34, 247
inheritance, 34
inherits, 247
input, 46
input values, 63
input variables, 63
integer, 44
    typing, 27
Intel, 15
intelligence, 9
intelligent systems, 9
intensity, 127, 265
interactions pane, 23
interface, 9, 251, 259
iterate, 280

Java, 11, 19
    installing, 21
Javadoc comments, 24
JPEG, 66, 81, 234

Kaehler, Ted, 320
Kay, Alan, 320
kilobyte, 233
kilobytes, 87
knots, 9

less-than, 332
less-than-or-equals, 332
liberal education, 17
Lisp, 11
literal, 47
loop, 280
    nested, 137
lossless compression, 234, 343
lossy compression, 81, 234, 343
luminance, 83, 86, 128

makeSound
    what it does, 275
Maloney, John, 320
matrix, 82
    definition, 82
max, 294
maximum, finding, 294
media computation, 15
MediaTools, 94, 268
    application, 279
    picture tools, 94
    sound tools, 268
megabyte, 87
memory, 13, 22
memory leak, 36
method, 43
    definition, 46, 56
    method body, 57
    method overloading, 61
    methods, 46
methods vs. recipes, 117
MIDI, 344
mirroring, 317
mixing sounds, 323
Moore’s Law, 15
Moore, Gordon, 15
MP3, 343
MP4, 343
MPEG-3, 343
Musical Instrument Digital Interface, 344

negative image, 125
nested, 137
nested loop, 172
nested loops, 137
networking, 10
new Color(redValue, greenValue, blueValue), 90
new Picture(fileName), 88
new Picture(width, height), 170
new Sound(fileName), 71
new Sound(int lengthInSamples), 334
noise, 269, 343
normalizing, 294
not-equals, 332
Nyquist theorem, 271
    applications, 271

object, 275
    definition, 20
object methods, 46
    definition, 46
object reference, 70
object variable, 70
object-oriented, 19
objects, 275
operating system, 45
operators, 25
overloading, 159, 206
overtones, 267

package
    definition, 91
packages, 227
painting tool, 235
parameters, 63
pass by value, 64
path, 66
path separator, 66
PCM, 273
percentage, 96
Perlis, Alan, 17
Index

physics
    color, 15
picture
    show(), 67
picture element, 15
picture objects, 80
picture tool, 94
pictureObject.repaint(), 92
pitch, 266
pitch interval, 267
pixel, 15, 80, 82
    definition, 82
pixelation, 207
pixelization, 83
pixels, 80
placeholder, 119
play(), 71
playNote, 344
posterizing, 202
Postscript, 234
precedence, 25
primitive, 69, 70
primitive type, 69
private visibility
    definition, 56
process, 17
program, 9, 12, 43, 44
    defined, 44
    definition, 12
programming language, 44
programming languages, 10
psychoacoustics, 266
public visibility
    definition, 56
pulse coded modulation (PCM), 273
ranges, 10
rarefactions, 265
recipe, 8
recipes vs. methods, 117
red, 15
reference, 70
reserved words, 29
return, 119, 294
reusable, 116
RGB color model, 84
Roads, Curtis, 303, 355
rounding errors, 92
run length encoding, 234

sample, 15, 271
sample index, 330
sample objects, 275
sample size, 296
sample sizes, 272
sampling, 166, 330, 333
sampling rate, 277
sampling interval, 333
sampling rate, 272
    in additive synthesis, 335
sawtooth, 268
scope, 106, 118
    definition, 118
sepia-toned, 200
setColor(color), 90
setRed(redValue), 90
short-circuit evaluation, 219
show(), 67, 88
sign bit, 273
signal view, 268
sine wave, 265
Siren, 320
slice, 269
software engineering, 9, 17
sonogram view, 269
sound
amplitude, 265
blockingPlay, 279
clocking, 273
decibels, 266
decreasing volume, 290
frequency, 265
frequency domain, 269
fundamental, 267
getLength(), 276
getSamples, 275
getSampleValueAt, 276
getSamplingRate, 277
ggetValue, 276
how to manipulate, 274
increasing volume, 284
intensity, 265
new Sound(fileName), 71
overtones, 267
Index

pitch, 266
play(), 71
pulse coded modulation, 273
sampling rate, 272
setSampleValueAt, 276
setValue, 276
sonogram view, 269
splicing, 308
volume, 265, 284, 290
sound cursor, 279
sound editor, 268
sound object, 274
sound pressure level, 266
source, 147
specification, 235
spectrum view, 269
spike, 269
splicing, 308
square brackets, 306
square waves, 339
Squeak, 320
static method, 335
stepping, 103
String, 44, 73
strings, 29
  appending, 30
strongly typed, 45
sub-program, 294
sub-recipe
  sampling, 330
subscript, 306
subsitution, 73
substitution, 63
subtractive synthesis, 343
Sun, 21
symbols, 43
syntax, 56
synthesizers, 328
systems, 9

target, 147
test, 332
testing, 288
  using print statements, 145
testing methods, 288, 296
The Godfather, 284
time, 9
this, 58, 105

time domain, 269

trace, 103

tracing, 290

using print statements, 145

transforming, 147

transistor, 15

transparency, 86, 159

TrueType, 234

two’s complement notation, 273

type, 43

types

byte, 44
double, 44
float, 44
int, 44
integer array, 44
String, 44

unboxing, 46

Unicode, 18

variable, 68

declaration, 31, 68

object, 70

primitive, 70

variables, 46

reusing, 36

vector representation, 234

vibrato, 326

visibility, 56

definition, 56

private, 56

public, 56

volume, 265

walking through, 103

Wallace, Scott, 320

WAV, 66

while, 96

while loop, 96, 281

window (of time), 269

wrapper, 46

wrapper classes

definition, 46

write(fileName), 92