



---

# 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*





ii

Copyright held by Mark Guzdial and Barbara Ericson, 2005.





Dedicated to our children Matthew, Katherine, and Jennifer.



# Preface

This book is intended to introduce computing, including programming, to students with no prior programming experience. One of the lessons from the research on computing education is that one doesn't just “learn to program.” One learns to program *something* [4, 17]. How motivating that *something* is can make the difference between learning to program or not [6]. Some people are interested in learning programming just for programming's sake—but that's not most people.

Unfortunately, most introductory programming books are written as if students have a burning desire to learn to program. They emphasize programming concepts and give little thought to making the problems that are being solved interesting and relevant. They introduce new concepts without showing why the students *should* want to know about them.

In this book students will learn about programming by writing programs to manipulate media. Students will create and modify images, such as correcting for “red-eye” and generating negative images. 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 seen on television and in movies.

Students in courses taught at Georgia Tech have found these programs interesting and motivating. Students have even reported turning in their programs and then continuing to work on them to see what else they can make.

This book is about teaching people to program in order to communicate. 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 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. So why learn to program these manipulations *yourself*? Why not just leave it to the developers of *Photoshop* and *iMovie*? The answer depends on your interests and career choices.

- If you have an interest in becoming a computing professional, then it's worthwhile for you to understand how to build programs used in communication. Much of the software in the future will be used for communications, so this is a great domain to start learning useful skills. Most computing classes today are taught in Java, so this book presents the right context for learning programming and in the right language for you.
- If you expect to be a *user* of applications in the future, knowing something of how your tools works can make you a so-called “*Power User*.” Most common applications today are much more powerful than most users realize. Many communications applications are actually themselves programmable

with *scripting languages* that enable users to automate tasks in the application. To use all the facilities of an application, it helps to have an understanding of what the application is doing—if you know what a *pixel* is, you can understand better why it’s useful to manipulate. To use the scripting facilities of an application, some knowledge of programming is a requirement.

- If you are a creative person who wants complete control of your communications, you want to know how to do without your applications if you need to, in order to implement your vision. Knowing *how* to do manipulate media with your own programs means that you *can* do what you want, if you ever 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.
- Finally, you may have no interest in programming your applications, or programming at all. Is it worthwhile for you to learn this stuff? Students who took our media computation classes at Georgia Tech told us *a year later* that the course was relevant *in their daily life* [15]. We live in a technological society, and much of that technology is used to manipulate what we see and hear in our media. If you know something of how that technology works, you have a way of thinking about how to use it, and how it may be used to change your perceptions. Students who are not computer science majors told us a year after finishing the course that they now had a new confidence around computers because they knew something about how they worked [15].

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. The last chapters at the end of the book are about *computing*, not just programming (chapters 15 and 16).

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

## TO TEACHERS

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

The majority of the book spends time giving students experiences with a variety of media in contexts that they find motivating. After that, 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 [22][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. Research in computing education suggests 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 ten chapters: three introductory, four on pictures, 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 our methods get too long to easily be debugged.

Our approach isn’t the more common approach of introducing one computing topic per chapter. We introduce computing concepts as needed to do a desired media manipulation (like using nested loops to mirror a picture). Some chapters introduce several computing concepts, while others repeat computing concepts in a different medium. We repeat concepts in different media to increase the odds that students will find an explanation and relevance that works for them, or better yet, find *two* or more explanations that work for them. The famous artificial intelligence researcher Marvin Minsky once said that if you understand something in only one way, you don’t understand it at all. Repeating a concept in different relevant settings can be a powerful way of developing flexible understandings.

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 [20].

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 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*. Some of the early methods are written the way that a beginning student would prefer, with values hard coded rather than passed in as parameters. It takes time and experience for students to realize that there is value in well-designed systems. Without experience to give the abstractions value, it's very difficult for beginning 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 [17]. 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 on-line, public posting of student work. Students learn from each other and try to outdo each other, in a spirit of creative competition.

### 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.
- Students with some programming experience could skip or review chapters 1 - 2 and begin at chapter 3. Students with object-oriented experience could start at chapter 4.
- Chapter 7 is about drawing using existing Java classes. It also introduces the concepts of inheritance and interfaces. The concepts introduced here are also used in chapter 14 (movies). If you are skipping movies you could skip this chapter as well.
- Chapters 8 through 10 replicate much of 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. You might want to at least cover class methods and private methods in chapter 10.

- Chapter 11 explains how to create classes. This is an essential chapter.
- Chapters 12 and 13 manipulate text. They also cover exceptions, reading and writing files, reading from the network, import statements, helper methods, some collection classes, iterators, generics, and working with databases. We recommend covering these chapters.
- Chapter 14 (on movies) introduces no new programming or computing concepts. While motivating, movie processing could be skipped for time.
- We do recommend getting to chapter 15 on speed. This is the first chapter that is more about computing than programming.
- Chapter 16 is about JavaScript. This gives students exposure to another language that is similar to Java. It also discusses interpreters and compilers. It could be skipped to save time.

## 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 was created at Rice University. It is free and easy to use. 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. Of course, you can also use more than one development environment. You could use DrJava for the interactions pane as well as another environment.

## TYPOGRAPHICAL NOTATIONS

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

```
public class Greeter
{
    public static void main(String [] args)
    {
        // show the string "Hello World" on the console
    }
}
```



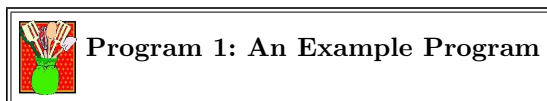
```
System.out.println(" Hello World" );  
}  
}
```

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.

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



Programs (recipes) appear like this:

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



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



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



**Debugging Tip: An Example Debugging Tip**  
If there's a good way to keep those bugs from creeping into your programs in the first place, they're highlighted here.



**Making it Work Tip: An Example How To Make It Work**  
Best practices or techniques that really help are highlighted like this.

## ACKNOWLEDGEMENTS

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.
- Matthew, Katherine, and Jennifer Guzdial all contributed pictures for use in this book.
- Thanks for permission to use their snapshots to Georgia Tech students: Jakita N. Owensby, and Tammy C.
- Thank you to the anonymous reviewers and to Brent Laminack for finding problems and for making suggestions to improve the book.
- Thank you to Thomas Bressoud and Matt Kretchmar at Denison University for trying an early version of the book and for their feedback on it.
- Thank you to the high school teachers in Georgia who took summer workshops using versions of this material and who taught it to their classes.



# Contents

<b>Preface</b>	<b>iv</b>
<b>Contents</b>	<b>xi</b>
<b>I Introduction</b>	<b>1</b>
<b>1 Introduction to Computer Science and Media Computation</b>	<b>2</b>
1.1 What is Computer Science About? . . . . .	2
1.2 What Computers Understand . . . . .	6
1.3 Media Computation: Why Digitize Media? . . . . .	8
1.4 Computer Science for Everyone . . . . .	9
1.4.1 It's About Communication . . . . .	10
1.4.2 It's About Process . . . . .	10
<b>2 Introduction to Java</b>	<b>15</b>
2.1 Java . . . . .	15
2.1.1 History of Java . . . . .	15
2.1.2 Introduction to Objects and Classes . . . . .	16
2.2 Introduction to DrJava . . . . .	17
2.2.1 Starting DrJava . . . . .	17
2.3 Java Basics . . . . .	20
2.3.1 Math Operators . . . . .	20
2.3.2 Printing the Result of a Statement . . . . .	23
2.3.3 Data Types in Math Expressions . . . . .	24
2.3.4 Casting . . . . .	24
2.3.5 Relational Operators . . . . .	25
2.3.6 Strings . . . . .	26
2.4 Variables . . . . .	28
2.4.1 Declaring Variables . . . . .	28
2.4.2 Using Variables in Calculations . . . . .	28
2.4.3 Memory Maps of Variables . . . . .	30
2.4.4 Object Variables . . . . .	31
2.4.5 Reusing Variables . . . . .	33
2.4.6 Multiple References to an Object . . . . .	35
2.5 Concepts Summary . . . . .	36
2.5.1 Statements . . . . .	36
2.5.2 Relational Operators . . . . .	37
2.5.3 Types . . . . .	37
2.5.4 Casting . . . . .	38
2.5.5 Variables . . . . .	38





<b>3</b>	<b>Introduction to Programming</b>	<b>42</b>
3.1	Programming is About Naming . . . . .	43
3.2	Files and their Names . . . . .	44
3.3	Class and Object Methods . . . . .	45
3.3.1	Invoking Class Methods . . . . .	45
3.3.2	Executing Object Methods . . . . .	47
3.4	Working with Turtles . . . . .	47
3.4.1	Defining Classes . . . . .	48
3.4.2	Creating Objects . . . . .	48
3.4.3	Sending Messages to Objects . . . . .	50
3.4.4	Objects Control Their State . . . . .	52
3.4.5	Additional Turtle Capabilities . . . . .	53
3.5	Creating Methods . . . . .	55
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 . . . . .	67
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 . . . . .	75
3.7.4	Creating new Methods . . . . .	75

**II Pictures 81**

<b>4</b>	<b>Modifying Pictures using Loops</b>	<b>82</b>
4.1	How Pictures are Encoded . . . . .	82
4.1.1	Color Representations . . . . .	87
4.2	Manipulating Pictures . . . . .	91
4.2.1	Exploring Pictures . . . . .	97
4.3	Changing color values . . . . .	98
4.3.1	Using a For-Each Loop . . . . .	99
4.3.2	Using While Loops . . . . .	101
4.3.3	Increasing/Decreasing Red (Green, Blue) . . . . .	105
4.3.4	Creating a Sunset . . . . .	120
4.3.5	Making Sense of Methods . . . . .	122
4.3.6	Variable Name Scope . . . . .	125
4.3.7	Using a For Loop . . . . .	129
4.3.8	Lightening and Darkening . . . . .	131
4.3.9	Creating a Negative . . . . .	132
4.3.10	Converting to Grayscale . . . . .	134





	<b>xiii</b>
4.4 Concepts Summary . . . . .	136
4.4.1 Arrays . . . . .	136
4.4.2 Loops . . . . .	136
4.4.3 Comments . . . . .	138
<b>5 Modifying Pixels in a Matrix</b>	<b>143</b>
5.1 Copying Pixels . . . . .	143
5.1.1 Looping Across the Pixels with a Nested Loop . . . . .	144
5.1.2 Mirroring a Picture . . . . .	146
5.2 Copying and Transforming Pictures . . . . .	154
5.2.1 Copying . . . . .	155
5.2.2 Creating a Collage . . . . .	161
5.2.3 Blending Pictures . . . . .	167
5.2.4 Rotation . . . . .	169
5.2.5 Scaling . . . . .	173
5.3 Concepts Summary . . . . .	179
5.3.1 Two-dimensional Arrays . . . . .	179
5.3.2 Nested Loops . . . . .	179
5.3.3 Returning a Value from a Method . . . . .	181
5.3.4 Method Overloading . . . . .	181
<b>6 Conditionally Modifying Pixels</b>	<b>185</b>
6.1 Conditional Pixel Changes . . . . .	186
6.1.1 Comparing Colors . . . . .	187
6.1.2 Replacing Colors . . . . .	187
6.1.3 Reducing Red-Eye . . . . .	191
6.2 Simple Edge Detection: conditionals with two options . . . . .	193
6.2.1 Negation . . . . .	194
6.2.2 Testing for Both Conditions . . . . .	194
6.2.3 Conditionals with Two Options . . . . .	194
6.2.4 Simple Edge Detection . . . . .	195
6.3 Sepia-Toned and Posterized Pictures: Using multiple conditionals to choose the color . . . . .	198
6.4 Highlighting Extremes . . . . .	205
6.5 Combining Pixels: Blurring . . . . .	206
6.6 Background Subtraction . . . . .	209
6.7 Chromakey . . . . .	214
6.8 Concepts Summary . . . . .	218
6.8.1 Boolean Expressions . . . . .	219
6.8.2 Combining Boolean Expressions . . . . .	219
6.8.3 Conditional Execution . . . . .	219
<b>7 Drawing</b>	<b>225</b>
7.1 Drawing on Images Using the Graphics Class . . . . .	225
7.1.1 Drawing with <b>Graphics</b> methods . . . . .	228
7.1.2 Vector and bitmap representations . . . . .	234
7.1.3 Drawing Text (Strings) . . . . .	236





xiv

7.2	Programs as Specifying Drawing Process . . . . .	240
7.2.1	Why do we write programs? . . . . .	243
7.3	Using Graphics2D for Advanced Drawing . . . . .	243
7.3.1	Setting the Brush Width . . . . .	244
7.3.2	Copying Pictures by Drawing Images . . . . .	245
7.3.3	General Scaling . . . . .	249
7.3.4	Shearing . . . . .	250
7.3.5	Drawing with a GradientPaint . . . . .	252
7.3.6	Interfaces . . . . .	252
7.3.7	Blending Pictures Using AlphaComposite . . . . .	254
7.3.8	Clipping . . . . .	257
7.4	Concept Summary . . . . .	258
7.4.1	Packages . . . . .	259
7.4.2	Predefined Java Classes . . . . .	259
7.4.3	Inheritance . . . . .	260
7.4.4	Interfaces . . . . .	260

**III Sounds 265**

**8 Modifying all Samples in a Sound 266**

8.1	How Sound is Encoded . . . . .	266
8.1.1	The Physics of Sound . . . . .	267
8.1.2	Exploring Sounds . . . . .	270
8.1.3	Encoding Sounds . . . . .	272
8.2	Manipulating Sounds . . . . .	276
8.2.1	Opening Sounds and Manipulating Samples . . . . .	277
8.2.2	Using MediaTools for Looking at Sounds . . . . .	280
8.2.3	Introducing Loops . . . . .	282
8.3	Changing the Volume of Sounds . . . . .	285
8.3.1	Increasing Volume . . . . .	286
8.3.2	Did that Really Work? . . . . .	287
8.3.3	Decreasing Volume . . . . .	291
8.3.4	Using a For Loop . . . . .	293
8.3.5	Making Sense of Methods . . . . .	294
8.4	Normalizing Sounds . . . . .	295
8.4.1	Generating Clipping . . . . .	297
8.5	Concepts Summary . . . . .	299
8.5.1	Arrays . . . . .	299
8.5.2	Loops . . . . .	300
8.5.3	Conditional Execution . . . . .	301

**9 Modifying Samples using Ranges 308**

9.1	Manipulating Different Sections of the Sound Differently . . . . .	308
9.2	Create a Sound Clip . . . . .	310
9.3	Splicing Sounds . . . . .	312
9.4	Reversing a Sound . . . . .	320





	<b>xv</b>
9.5	Mirroring a Sound . . . . . 321
9.6	Concepts Summary . . . . . 322
9.6.1	Ranges in Loops . . . . . 322
9.6.2	Returning a Value from a Method . . . . . 322
<b>10</b>	<b>Combining and Creating Sounds 327</b>
10.1	Composing Sounds Through Addition . . . . . 327
10.2	Blending Sounds . . . . . 329
10.3	Creating an Echo . . . . . 330
10.3.1	Creating Multiple Echoes . . . . . 333
10.4	How Sampling Keyboards Work . . . . . 334
10.4.1	Sampling as an Algorithm . . . . . 340
10.5	Additive Synthesis . . . . . 340
10.5.1	Making Sine Waves . . . . . 340
10.5.2	Creating Sounds Using Static Methods . . . . . 341
10.5.3	Adding Sine Waves Together . . . . . 343
10.5.4	Checking our Result . . . . . 344
10.5.5	Square Waves . . . . . 345
10.5.6	Triangle Waves . . . . . 348
10.6	Modern Music Synthesis . . . . . 349
10.6.1	MP3 . . . . . 349
10.6.2	MIDI . . . . . 350
10.6.3	Private Methods . . . . . 352
10.7	Concepts Summary . . . . . 355
10.7.1	Class Methods . . . . . 355
10.7.2	Private Methods . . . . . 355
10.7.3	Build a Program from Multiple Methods . . . . . 356
<b>11</b>	<b>Creating Classes 359</b>
11.1	Identifying the Objects and Fields . . . . . 360
11.2	Defining a Class . . . . . 360
11.2.1	Defining Fields . . . . . 361
11.2.2	Inherited Methods . . . . . 363
11.2.3	Overriding Inherited Methods . . . . . 365
11.2.4	Default Field Initialization . . . . . 366
11.2.5	Declaring Constructors . . . . . 367
11.2.6	Using a Debugger . . . . . 369
11.3	Overloading Constructors . . . . . 370
11.4	Creating and Initializing an Array . . . . . 372
11.4.1	Calculating the Grade Average . . . . . 373
11.4.2	Using Step Into in the Debugger . . . . . 376
11.5	Creating Accessors (Getters) and Modifiers (Setters) . . . . . 378
11.5.1	Creating Accessors (Getters) . . . . . 379
11.5.2	Creating Modifiers (Setters) . . . . . 380
11.6	Creating a Main Method . . . . . 382
11.7	Javadoc Comments . . . . . 383
11.7.1	Class Comment . . . . . 384





- 11.7.2 Method Comments . . . . . 384
- 11.7.3 Constructor Comments . . . . . 385
- 11.7.4 Generating the Documentation . . . . . 385
- 11.8 Creating another Class . . . . . 385
  - 11.8.1 Adding Constructors . . . . . 387
  - 11.8.2 Adding Accessors and Modifiers . . . . . 388
- 11.9 Reusing a Class Via Inheritance . . . . . 389
  - 11.9.1 Dynamic (Run-time) Binding . . . . . 395
- 11.10 Concepts Summary . . . . . 396
  - 11.10.1 Declaring a Class . . . . . 396
  - 11.10.2 Fields . . . . . 396
  - 11.10.3 Constructors . . . . . 397
  - 11.10.4 Arrays . . . . . 397
  - 11.10.5 Using a Debugger . . . . . 397
  - 11.10.6 Javadoc Comments . . . . . 398
  
- IV Text, Files, Networks, Databases, and Unimedia 401**
  
- 12 Creating and Modifying Text 402**
  - 12.1 Text as Unimedia . . . . . 403
  - 12.2 Strings: Character Sequences . . . . . 403
    - 12.2.1 Unicode . . . . . 404
    - 12.2.2 String Methods . . . . . 405
    - 12.2.3 Processing Delimited Strings Using Split . . . . . 408
    - 12.2.4 Strings Don't Have a Font . . . . . 410
  - 12.3 Files: Places to put Your Strings and Other Stuff . . . . . 410
    - 12.3.1 Reading from Files . . . . . 411
    - 12.3.2 Handling Exceptions . . . . . 412
    - 12.3.3 Working with an ArrayList . . . . . 417
    - 12.3.4 Writing to a File . . . . . 422
    - 12.3.5 Generating a Form Letter . . . . . 423
    - 12.3.6 Modifying Programs . . . . . 425
  - 12.4 Other Useful Classes . . . . . 435
    - 12.4.1 Another Fun Class: Random . . . . . 436
  - 12.5 Networks: Getting our Text From the Web . . . . . 439
  - 12.6 Using Text to Shift Between Media . . . . . 445
  - 12.7 Concepts Summary . . . . . 451
    - 12.7.1 Exceptions . . . . . 451
    - 12.7.2 Reading and Writing Files . . . . . 451
    - 12.7.3 Reading from the Internet . . . . . 452
    - 12.7.4 Import Statements . . . . . 452
    - 12.7.5 While Loops . . . . . 452
  
- 13 Making Text for the Web 459**
  - 13.1 HTML: The Notation of the Web . . . . . 459
  - 13.2 Writing programs to generate HTML . . . . . 465







	<b>xvii</b>
13.2.1	Creating a Web Page from a Directory . . . . . 470
13.2.2	Creating a Web Page from other Web Pages . . . . . 472
13.2.3	Adding Randomness to a Homepage . . . . . 474
13.3	Databases: A place to store our text . . . . . 477
13.3.1	Key and Value Maps . . . . . 477
13.3.2	Downcasting . . . . . 478
13.3.3	Generics . . . . . 481
13.4	Relational databases . . . . . 484
13.4.1	SQL . . . . . 486
13.4.2	Getting Started: Drivers and Connections . . . . . 486
13.4.3	Querying the Database . . . . . 490
13.4.4	Using a database to build Web pages . . . . . 494
13.5	Concepts Summary . . . . . 496
13.5.1	HTML . . . . . 496
13.5.2	Helper Methods . . . . . 496
13.5.3	Throwing an Exception . . . . . 497
13.5.4	The “Unnamed” Package . . . . . 497
13.5.5	HashMap . . . . . 497
13.5.6	Generics . . . . . 497
13.5.7	Iterators . . . . . 497
13.5.8	JDBC and SQL . . . . . 498
<b>V</b>	<b>Movies 501</b>
<b>14</b>	<b>Encoding, Manipulating, and Creating Movies 502</b>
14.1	Generating Frame-Based Animations . . . . . 503
14.2	Working with Video Frames . . . . . 513
14.2.1	Video manipulating examples . . . . . 513
14.3	Concepts Summary . . . . . 519
<b>VI</b>	<b>Topics in Computer Science 523</b>
<b>15</b>	<b>Speed 524</b>
15.1	Focusing on Computer Science . . . . . 524
15.2	What makes programs fast? . . . . . 525
15.2.1	What computers really understand . . . . . 525
15.2.2	Compilers and Interpreters . . . . . 526
15.2.3	The Special Case of Java . . . . . 535
15.2.4	How fast can we really go? . . . . . 536
15.2.5	Making searching faster . . . . . 538
15.2.6	Algorithms that never finish or can’t be written . . . . . 541
15.2.7	Why is Photoshop faster than our programs in Java? . . . . . 543
15.3	What makes a computer fast? . . . . . 543
15.3.1	Clock rates and actual computation . . . . . 543
15.3.2	Storage: What makes a computer slow? . . . . . 544





xviii

15.3.3 Display . . . . .	545
15.4 Concepts Summary . . . . .	546
<b>16 JavaScript</b>	<b>549</b>
16.1 JavaScript syntax . . . . .	549
16.2 JavaScript inside of Web pages . . . . .	551
16.3 User interfaces in JavaScript . . . . .	553
16.4 Multimedia in JavaScript . . . . .	559
16.5 Concepts Summary . . . . .	561

## APPENDICES

<b>A Quick Reference to Java</b>	<b>563</b>
A.1 Variables . . . . .	563
A.2 Method Declarations . . . . .	564
A.3 Loops . . . . .	565
A.4 Conditionals . . . . .	566
A.5 Operators . . . . .	567
A.6 String escapes . . . . .	567
A.7 Classes . . . . .	567
A.8 Fields . . . . .	568
A.9 Constructors . . . . .	568
A.10 Packages . . . . .	569





# List of Figures

1.1	A cooking recipe—the order of the steps is important. . . . .	3
1.2	Comparing programming languages: A common simple programming task is to print the words “Hello, World!” to the screen. . . . .	13
1.3	Eight wires with a pattern of voltages is a byte, which gets interpreted as a pattern of eight 0’s and 1’s, which gets interpreted as a decimal number. . . . .	14
2.1	DrJava Preferences Window . . . . .	18
2.2	DrJava Splash Screen . . . . .	18
2.3	DrJava (with annotations) . . . . .	20
2.4	A calculator with a number in memory . . . . .	28
2.5	Declaring primitive variables and memory assignment . . . . .	30
2.6	Showing the parent and child relationship between mammal and dog (left) and <code>Object</code> and <code>String</code> (right) . . . . .	32
2.7	Declaring object variables and memory assignment . . . . .	33
2.8	Shows creation and reuse of an object variable. . . . .	34
2.9	An object with multiple references to it . . . . .	35
2.10	An object with no references to it . . . . .	36
3.1	A window that shows a <code>World</code> object. . . . .	49
3.2	A window that shows a <code>Turtle</code> object in a <code>World</code> object. . . . .	50
3.3	A window that shows two <code>Turtle</code> objects in a <code>World</code> object. . . . .	51
3.4	The result of messages to the first <code>Turtle</code> object. . . . .	52
3.5	The result of messages to the second <code>Turtle</code> object. . . . .	52
3.6	The turtle won’t leave the world . . . . .	53
3.7	Drawing two squares with a turtle. . . . .	55
3.8	Defining and executing <code>drawSquare()</code> . . . . .	59
3.9	An object stores data for that object and has a reference to the class that created it . . . . .	61
3.10	Showing the result of sending the width as a parameter to <code>drawSquare</code> . . . . .	63
3.11	Creating a <code>Picture</code> object using <code>new Picture()</code> . . . . .	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 beach-smaller.jpg. . . . .	68
3.15	Picking, making, and showing a picture, when naming the pieces. The picture shown is tammy.jpg. Tammy is one of the computer science graduate students at Georgia Tech. . . . .	71
4.1	A depiction of the first five elements in an array . . . . .	83
4.2	The top left corner of a battleship guess board with a miss at B-3. . . . .	85
4.3	Picturing a 2-d array as row-major or column-major . . . . .	85
4.4	An example matrix (two-dimensional array) of numbers . . . . .	86





xx LIST OF FIGURES

4.5	Upper left corner of DrJava window with part magnified 600%	86
4.6	Image shown in the picture explorer: 100% image on left and 500% on right (close-up of the branch over the mountain)	87
4.7	Merging red, green, and blue to make new colors	88
4.8	Picking colors using the HSB color model	88
4.9	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	89
4.10	The Macintosh OS X RGB color picker	90
4.11	Picking a color using RGB sliders from Java	90
4.12	RGB triplets in a matrix representation	91
4.13	Directly modifying the pixel colors via commands: Note the small black line on the left under the line across the leaf	97
4.14	Exploring the caterpillar with the line	97
4.15	Using the MediaTools image exploration tools on barbara.jpg	98
4.16	Flowchart of a while loop	102
4.17	The original picture (left) and red-decreased version (right)	107
4.18	Using the picture explorer to convince ourselves that the red was decreased	117
4.19	Overly blue (left) and red increased by 30% (right)	119
4.20	Original (left) and blue erased (right)	120
4.21	Original beach scene (left) and at (fake) sunset (right)	122
4.22	Flowchart of a for loop	130
4.23	Original picture, lightened picture, and darkened picture	133
4.24	Negative of the image	134
4.25	Color picture converted to grayscale	135
5.1	Once we pick a mirror point, we can just walk $x$ halfway and copy from $(x,y)$ to $(width - 1 - x,y)$	147
5.2	Original picture (left) and mirrored along the vertical axis (right)	148
5.3	A motorcycle mirrored horizontally, top to bottom (left) and bottom to top (right)	150
5.4	Temple of Hephaistos from the ancient agora in Athens, Greece	151
5.5	Coordinates where we need to do the mirroring	151
5.6	The manipulated temple	152
5.7	Copying a picture to a canvas	156
5.8	Copying a picture midway into a canvas	157
5.9	Copying part of a picture onto a canvas	159
5.10	Flowers in the <code>mediasources</code> folder	161
5.11	Collage of flowers	164
5.12	Blending the picture of Katie and Jenny	169
5.13	Rotating some numbers in a table to the left 90 degrees	170
5.14	Copying a picture to a blank page rotated to the left 90 degrees	171
5.15	Scaling the picture of Jakita (a CS graduate student at Georgia Tech) down	174
5.16	Scaling up a picture	176
6.1	Flowchart of an if statement	186
6.2	Increasing reds in the browns	189
6.3	On left the couch color changes, on right the couch color doesn't change	190





LIST OF FIGURES xxi

6.4	Finding the range of where Jenny’s eyes are red . . . . .	192
6.5	After fixing red-eye. . . . .	193
6.6	Flowchart of an if with an else . . . . .	195
6.7	Original picture and after edge detection . . . . .	197
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 kids in front of a wall, and a picture of the wall . . . . .	214
6.18	Swapping a country bridge for the wall, using background subtraction, with a threshold of 50 . . . . .	215
6.19	Mark in front of a blue sheet . . . . .	216
6.20	Mark on the moon . . . . .	217
6.21	Mark on the beach . . . . .	217
7.1	Adding a grid of lines to a picture (barbara.jpg) . . . . .	227
7.2	Viewing the Java API for java.awt.Graphics . . . . .	230
7.3	A box washed up on the shore of the beach . . . . .	230
7.4	An example drawn picture . . . . .	233
7.5	A drawn face (left) and the face with enclosing rectangles (right) . . . . .	234
7.6	Shows font information including the baseline . . . . .	236
7.7	Drawing a string on a picture . . . . .	238
7.8	Drawing a string centered on a picture . . . . .	239
7.9	A programmed gray scale effect . . . . .	240
7.10	Nested colored rectangles . . . . .	242
7.11	Nested outlined rectangles . . . . .	243
7.12	Drawing a red X on a picture . . . . .	246
7.13	Drawing a turtle on a beach . . . . .	247
7.14	Documentation for Graphics2D . . . . .	248
7.15	Original picture and picture scaled up 2 times in x and down by half in y . . . . .	250
7.16	Picture sheared by 1.0 in x . . . . .	252
7.17	A beach with a sun that is filled with a gradient from yellow to red . . . . .	254
7.18	Two pictures with a horizontal overlap . . . . .	257
7.19	Clip picture using an ellipse . . . . .	258
8.1	Raindrops causing ripples in the surface of the water, just as sound causes ripples in the air . . . . .	267
8.2	One cycle of the simplest sound, a sine wave . . . . .	268
8.3	The note A above middle C is 440 Hz . . . . .	269
8.4	Some synthesizers using triangular (or <i>sawtooth</i> ) or square waves. . . . .	270
8.5	Sound editor main tool . . . . .	270
8.6	Viewing the sound signal as it comes in . . . . .	271
8.7	Viewing the sound in a spectrum view . . . . .	272





xxii LIST OF FIGURES

8.8	Viewing a sound in spectrum view with multiple “spikes” . . . . .	273
8.9	Viewing the sound signal in a sonogram view . . . . .	274
8.10	Area under a curve estimated with rectangles . . . . .	274
8.11	A depiction of the first five elements in a real sound array . . . . .	276
8.12	A sound recording graphed in the MediaTools . . . . .	276
8.13	The sound editor open menu in MediaTools application . . . . .	281
8.14	MediaTools application open file dialog . . . . .	281
8.15	A sound opened in the editor in MediaTools application . . . . .	282
8.16	Exploring the sound in the editor in MediaTools application . . . . .	282
8.17	Comparing the graphs of the original sound (left) and the louder one (right) . . . . .	287
8.18	Comparing specific samples in the original sound (left) and the louder one (right) . . . . .	288
8.19	Comparing the original sound with the normalized one . . . . .	298
8.20	Comparing the original sound with one with all values set to extremes.	299
9.1	Exploring the ”This is a test” to find the end of the first word . . .	311
9.2	Exploring the sound clip . . . . .	313
9.3	Comparing the original sound (left) to the spliced sound (right) . . .	318
9.4	Comparing the original sound (left) to the reversed sound (right) . .	321
9.5	Comparing the mirror point in the original sound (left) to the mir- rored sound (right) . . . . .	322
10.1	The top and middle waves are added together to create the bottom wave . . . . .	328
10.2	The original “ahh” sound, the original bassoon note, and the blended sound . . . . .	331
10.3	The original “This is a test” sound (left), and the sound with an echo (right) . . . . .	332
10.4	The original sound (left), and the sound with the frequency doubled (right) . . . . .	335
10.5	The sine wave with a frequency of 880 and a maximum amplitude of 4000 . . . . .	343
10.6	The raw 440 Hz signal on top, then the 440+880+1320 Hz signal on the bottom . . . . .	345
10.7	FFT of the 440 Hz sound . . . . .	345
10.8	FFT of the combined sound . . . . .	345
10.9	The 440 Hz square wave (top) and additive combination of square waves (bottom) . . . . .	347
10.10	FFT’s of the 440 Hz square wave (top) and additive combination of square waves (bottom) . . . . .	347
11.1	DrJava with the class definition for Student . . . . .	361
11.2	Showing a student object with a reference to its class. . . . .	363
11.3	Showing a student object with a reference to its class and a reference from the class to its parent class. . . . .	364
11.4	Showing a student object with the default initialization of the fields. .	366
11.5	DrJava with debug mode turned on . . . . .	370
11.6	A breakpoint highlighted in red . . . . .	371
11.7	Execution stops at the breakpoint . . . . .	372





LIST OF FIGURES xxiii

11.8 After clicking on Step Over . . . . . 373

11.9 Showing a student object after a constructor that takes a name and  
array of grades has executed. . . . . 374

11.10 Breakpoint in the constructor that takes a name and grade array . . 376

11.11 Execution stopped at the breakpoint . . . . . 377

11.12 Execution stopped after step over . . . . . 378

11.13 Execution stopped at the second breakpoint . . . . . 379

11.14 Execution stopped at the beginning of `getAverage` . . . . . 380

11.15 Showing the HTML documentation generated from Javadoc comments 386

11.16 UML class diagram . . . . . 387

11.17 Result of commands to a confused turtle. . . . . 392

11.18 Result of overriding the `turn` method. . . . . 393

11.19 Diagram of the methods executed by `fred.turnLeft()`. . . . . 394

11.20 Result of fixing the methods `turnLeft()` and `turnRight()`. . . . . 395

12.1 Diagram of a directory tree . . . . . 411

12.2 Showing a class stack before a method returns and after . . . . . 412

12.3 A depiction of the inheritance tree for some exception classes . . . . 414

12.4 Original cartoon (left) and after the class has been modified (right)) 430

12.5 A depiction of the inheritance tree for some of the Reader Classes . 442

12.6 Sound-as-text file read into Excel . . . . . 447

12.7 Sound-as-text file graphed in Excel . . . . . 448

12.8 A visualization of the sound “her.wav” . . . . . 450

13.1 Simple HTML page source . . . . . 461

13.2 Simple HTML page open in Internet Explorer . . . . . 461

13.3 HTML styles . . . . . 462

13.4 Inserting an image into an HTML page . . . . . 463

13.5 An HTML page with a link in it . . . . . 464

13.6 Inserting a table into an HTML page . . . . . 464

13.7 Creating an image thumbnail page . . . . . 473

13.8 An example relational table . . . . . 485

13.9 Representing more complex relationships across multiple tables . . . 485

13.10 ODBC Data Source Administrator Window . . . . . 487

13.11 Create New Data Source Window . . . . . 488

13.12 An ODBC data source for the Microsoft Access person database . . 488

14.1 A few frames from the first movie: Moving a rectangle . . . . . 505

14.2 Frames from the tickertape method . . . . . 507

14.3 Moving two rectangles at once . . . . . 508

14.4 Frames from moving Mark’s head around . . . . . 509

14.5 Frames from the make sunset movie . . . . . 511

14.6 Frames from the slow fade-out movie . . . . . 512

14.7 Movie tools in MediaTools . . . . . 513

14.8 Frames from the Mommy watching Katie movie . . . . . 513

14.9 Frames from the original movie of the kids crawling in front of a blue  
screen . . . . . 516

14.10 Frames from the kids on the moon movie . . . . . 517

14.11 Some frames from the original too blue movie . . . . . 517

14.12 Some frames from the color corrected movie . . . . . 519





xxiv LIST OF FIGURES

15.1	Results of running the <code>GraphicsInterpreter</code>	530
15.2	A couple sample computer advertisements	543
16.1	Simple JavaScript function	552
16.2	Showing the parts of the simple JavaScript function	552
16.3	Using JavaScript to insert HTML	553
16.4	Using JavaScript to compute a loop	554
16.5	Computing a list that counts to ten	554
16.6	Inserting the date and time into a web page	555
16.7	Example JavaScript dialog windows	555
16.8	Example catching the <code>onClick</code> event	556
16.9	Opening a JavaScript window	557
16.10	Changing the new JavaScript window	558
16.11	Changing color of list items	558
16.12	A simple HTML form	559
16.13	Inch/centimeter converter in JavaScript	559







# PART ONE

---

# INTRODUCTION

**Chapter 1 Introduction to Computer Science and Media Computation**

**Chapter 2 Introduction to Java**

**Chapter 3 Introduction to Programming**





## C H A P T E R 1

# Introduction to Computer Science and Media Computation

- 
- 1.1 WHAT IS COMPUTER SCIENCE ABOUT?
  - 1.2 WHAT COMPUTERS UNDERSTAND
  - 1.3 MEDIA COMPUTATION: WHY DIGITIZE MEDIA?
  - 1.4 COMPUTER SCIENCE FOR EVERYONE
- 

### Chapter Learning Objectives

- To explain what computer science is about and what computer scientists are concerned with.
- To explain why we digitize media.
- To explain why it's valuable to study computing.
- To use the concept of an *encoding*.
- To explain the basic components of a computer.

### 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 as shown in the recipe in (Figure 1.1).

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



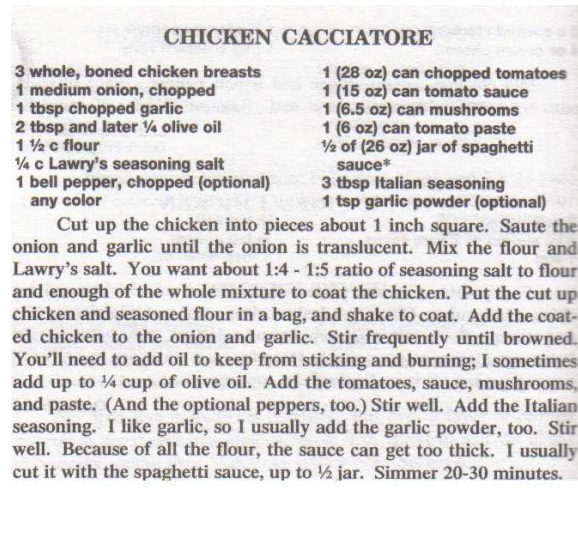


FIGURE 1.1: A cooking recipe—the order of the steps is important.

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 to 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,

4 Chapter 1 Introduction to Computer Science and Media Computation

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 con-

sistently), 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 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!

The programming language that we’re using in this book is *Java* (<http://java.sun.com> for more information on Java). Java is a very popular programming language. Delta uses it to handle its web site (<http://www.delta.com>). NASA used it on the Mars Rover “Spirit” (<http://www.sun.com/aboutsun/media/features/mars.html>). It has been used in touchscreen kiosks for Super Bowl fans (<http://java.sun.com/features/1998/01/superbowl.html>).

Java is known for being object-oriented, platform neutral (runs on many 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 substantially

reduced 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.



**Computer Science Idea: Programs versus Algorithms**

A program is written in a programming language and can be executed by a computer. An algorithm can be written in English and is a description of a process. Many programs can implement an algorithm in many different programming languages.

**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. It might be more appropriate to say that computers are used to *encode* (represent) numbers. 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<sup>1</sup>.



**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

<sup>1</sup>We’ll talk more about this level of the computer in chapter 15

that a computer is working with is *encoded* in its bytes: JPEG pictures, Excel spreadsheets, Word documents, annoying Web pop-up ads, and the latest spam email.

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

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



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

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

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

We can similarly create layers of encodings in the computer for our specific



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

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



**Computer Science Idea: Moore's Law**

Gordon Moore, one of the founders of Intel (maker of computer processing chips for computers running Windows operating systems), made the claim that the number of transistors (a key component of computers) would double at the same price every 18 months, effectively meaning that the same amount of money would buy twice as much computing power every 18 months. This Law has continued to hold true for decades.

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

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

**1.3 MEDIA COMPUTATION: WHY DIGITIZE MEDIA?**

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

What if we encoded each dot in a picture as a collection of three bytes, one each for the amount of red, green, and blue at that dot on the screen? We could collect a bunch of these three-byte-sets to specify all the dots of a given picture. That's a pretty reasonable way of representing pictures, and it's essentially how we're going to do it in chapter 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* learn about computation? Of course, people who want to be computer scientists will need to learn about computation. Why should anyone who doesn't want to be a computer scientist learn about computer science?

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

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

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

### 1.4.1 It's About Communication

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

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

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

### 1.4.2 It's About Process

In 1961, Alan Perlis gave a talk at MIT where he made the argument that computer science, and programming explicitly, should be part of a general, liberal education [13]. 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 2), 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. What is a program?
- 1.2. What is an algorithm?
- 1.3. What is memory used for in a computer?

Section 1.4 Computer Science for Everyone 11

- 1.4. What type of computer scientist studies how recipes are written? What type of computer scientist studies how to make a computer think? What type of computer scientist studies the units used in recipes?
- 1.5. What is Moore’s Law? What does it have to do with computers getting faster and cheaper?
- 1.6. *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.”
- 1.7. Look in the classified section of your newspaper. What kinds of jobs can people get with a degree in computer science? How much money do they make? How many jobs are available?
- 1.8. Go to <http://www.howstuffworks.com> and find out how digital cameras work.
- 1.9. Go to <http://www.howstuffworks.com> and find out how digital recording and CDs work.
- 1.10. Go to <http://www.howstuffworks.com> and find out remote entry devices work.
- 1.11. Find an ASCII table on the Web: A table listing every character and its corresponding numeric representation. Write down the sequence of numbers whose ASCII values make up your name.
- 1.12. Find a *Unicode* table on the Web. What’s the difference between ASCII and Unicode? How many bytes does each use to represent a character?
- 1.13. 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?)
- 1.14. How many digits are used in the binary number system? How many digits are used in the decimal number system? How would you represent 3, 5, 8, and 13 in the binary number system?
- 1.15. What is the hexadecimal number system? How many digits are used in the hexadecimal number system? How would you represent 4, 18, 33, and 64 in this number system?
- 1.16. What is the octal number system? How many digits are used in the octal number system? How would you represent 4, 18, 33, and 64 in this number system?
- 1.17. How many digits are in one byte? How many different numbers can be represented by one byte? What if you have two bytes? Four bytes?
- \*1.18. How would you represent negative numbers in bytes? Do a search on the web for “negative numbers” and see what you find.
- \*1.19. 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.
- 1.20. Look up Alan Kay and the *Dynabook* on the Web. Who is he, and what does he have to do with media computation?
- 1.21. 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?
- 1.22. 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* [23] 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.



### Python/Jython

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

### Java

```
public class HelloWorld  
{  
    public static 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.

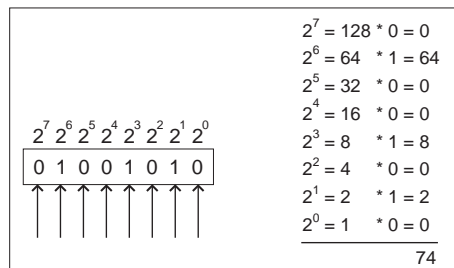


FIGURE 1.3: Eight wires with a pattern of voltages is a byte, which gets interpreted as a pattern of eight 0's and 1's, which gets interpreted as a decimal number.

## C H A P T E R 2

# Introduction to Java

- 
- 2.1 JAVA
  - 2.2 INTRODUCTION TO DRJAVA
  - 2.3 JAVA BASICS
  - 2.4 VARIABLES
  - 2.5 CONCEPTS SUMMARY
- 

### 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 programs (*applets*) embedded in HTML pages to do 3D rotation of a molecule and animation of a sorting algorithm. They showed this at a conference. At that time web pages didn't respond to user action. They simply displayed text and unchanging graphics. The audience was amazed to see the user rotate a 3d molecule on a web page.

A patent search found that there was an existing programming language with the copyrighted name Oak, so the team brainstormed new names at a local coffee house and Java was selected. Java was released 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 we 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 that 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 the same 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

We recommend that you program 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 development environment. Simply set the classpath (place to look for classes that you are using in your program) to include the classes used in this book. Check your documentation for your development environment to see how to do this. We recommend using DrJava because it is free, 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. Since it is free, you can use it just for the interactions pane, and do your coding in another development environment if you prefer.

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

1. Make sure that you have Java 1.4 or above installed on your computer. If you don't have it, load it from the CD, or you can get it from the Sun site at <http://www.java.sun.com>.
2. You'll need to install DrJava. You can either load it from the CD or get it from <http://drjava.org/>. Be sure to get a version of DrJava that works with the version of Java you are using! The current stable release at the time this book was being written only worked with Java 1.4. To use Java 1.5 you would need to download the beta version of DrJava. If you have more than one version of Java installed on your machine you will probably need to use the jar file version of DrJava instead of the DrJava executable. See the CD for more information on using the jar file version of DrJava.
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-

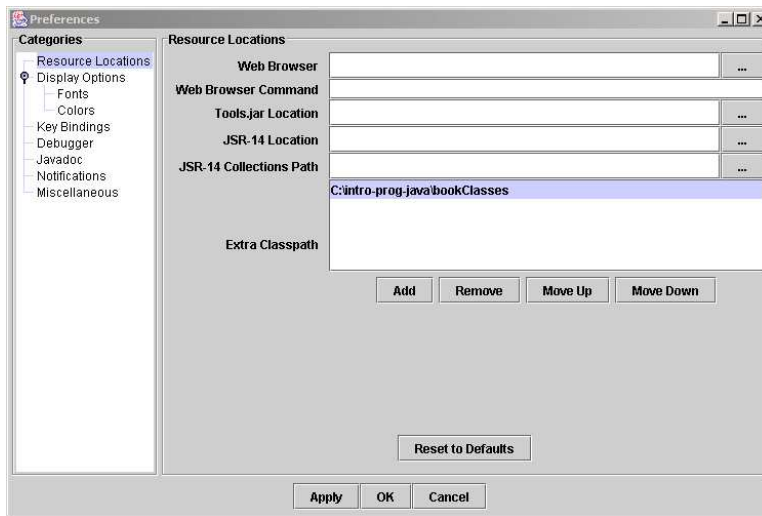


FIGURE 2.1: DrJava Preferences Window

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 Macintosh, you'll probably have to type commands in your *Terminal* application where you `cd` to the correct directory then type `./DrJava`. See the instructions on the CD for what will work for your kind of computer.


 **Common Bug: DrJava is slow to start**  
DrJava will take a while to load on all platforms. Don't worry—you'll see (Figure 2.2) for a long time. This is called a *splash screen*, which is a small picture that displays while a program is loading. If you see the splash screen (Figure 2.2), DrJava will load.



FIGURE 2.2: DrJava Splash Screen



### 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).
- 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 COMPILER 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.

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

20 Chapter 2 Introduction to Java

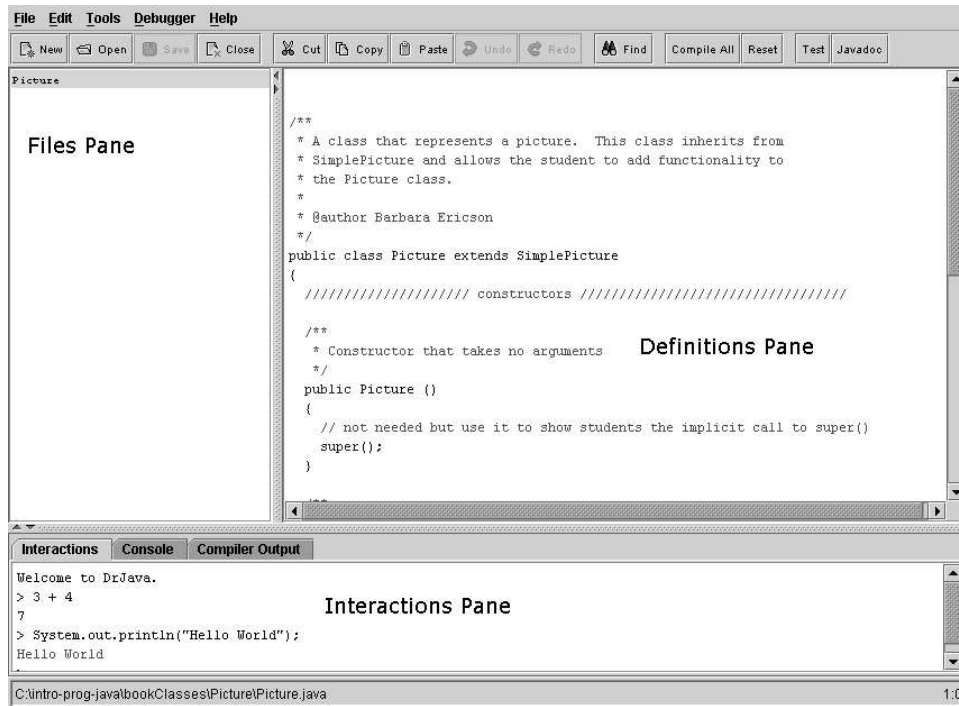


FIGURE 2.3: DrJava (with annotations)

'/\*\*' 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.



### 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 a main method. To get the above example to work in another development environment we could have written the following class definition in a file named “Test.java”.

```
public class Test
{
    public static void main(String [] args)
    {
        System.out.println(34 + 56);
        System.out.println(26 - 3);
        System.out.println(3 * 4);
        System.out.println(4/2);
    }
}
```

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:

```
> 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:

```
> java Test
```

We have included this `Test` class in your `bookClasses` directory. You can continue to use the `Test` class and just change the code in the `main` method to try the examples we show in DrJava’s interactions pane. We will explain all about classes and main methods in chapter 11.

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. Remember that it is **free**, so even if you use another development environment you can download it and use it too, at least for the interactions pane!

### 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, how do you show the result of a statement in the interactions pane? The phrase `System.out.println()` is an important one to know. The meaning for `System.out.println()` is “Use the `PrintStream` object known as `out` in 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. This is called the *modulus* operator.

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 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.

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

### 2.3.3 Data Types in Math Expressions

Java takes how you specify numbers seriously. If it sees you using integers, it thinks you want an integer result. If it sees you using floating point numbers, it thinks you want a floating point result. Sounds reasonable, no? But how about:

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

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

We call integer and floating point numbers two different types of *data*. By data we mean the values that we use in computation. The type of the data, which is also called the *data type*, determines how many bits are used to represent the value and how the bits are interpreted by the computer.

### 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 precise. We will use this type for most of the floating point numbers in this book. Notice that we can cast either the 1 or 2 to `double` and the answer will then be given as a `double`. We could cast both the 1 and 2 to `double` and the result would be fine.



However, if we cast the result of the integer division to a double it is too late since the result of integer division of 1 by 2 is 0 since the result is an integer.

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

```
> System.out.println(3 > 2);  
true  
> System.out.println(2 > 3);  
false  
> System.out.println('a' < 'b');  
true  
> System.out.println('j' > 'c');  
true  
> System.out.println(2 == 2);  
true  
> System.out.println(2 != 2);  
false  
> System.out.println(2 >= 2);  
true  
> System.out.println(2 <= 2);  
true  
> System.out.println(true == false);  
false
```

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, 502893, and -2350. 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 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.

```
> 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* or *concatenating* 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.

```
> 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.

```
> 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

### 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 'is equal to' or 'is equivalent to' 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 the type `double` to store the bill amount, tip, and total amount since these can have decimal points. 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
```



#### Common Bug: Mistyping

You just saw a whole bunch of Java statements, 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:

```
> double tip = bil * 0.2;
Error: Undefined class 'bil'
```

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.

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 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.

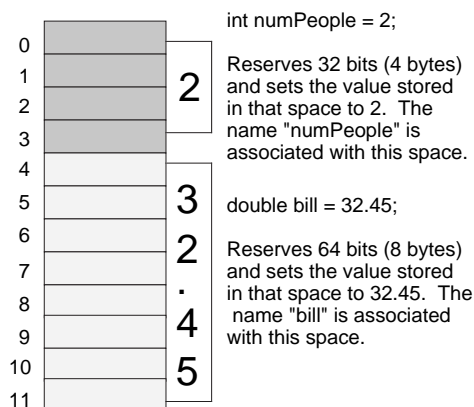


FIGURE 2.5: Declaring primitive variables and memory assignment

When we print out a variable name using `System.out.println(bill)`; the 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, but is now 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 multiplying 40% (0.40) times the original price. Next, calculate the first discount total by subtracting the first discount amount from the original price. Then 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);
23.0352
```

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 128,000,000 bytes of memory and we used 40 bytes then we have only used 0.000003125% 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`.

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 2.6). If you need a mammal you can use a dog, but if you need a dog another mammal (like a cat) won't do. Because a dog is a kind of mammal we know that it has the same characteristics that a mammal does such as breathing oxygen, bearing live young, having hair, etc. We say that it inherits characteristics from mammal. The `String` class is a child of the `Object` class so it is a kind of object (Figure 2.6). 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

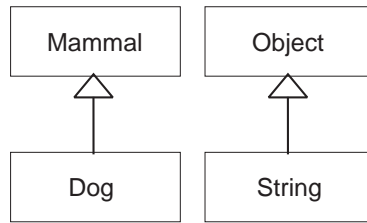


FIGURE 2.6: Showing the parent and child relationship between mammal and dog (left) and `Object` and `String` (right)

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 of characters use: `String name;`

Object variables reserve space for something which is a *reference* to an object. A reference isn't the address of the object in memory. It is more like a Dewey Decimal System number. Once you know the Dewey Decimal System number for a book you can find the book on the library shelves. An object reference gives the computer a way to find an object in memory.

Object variables do not reserve space for the object. If the object variable doesn't reference an object yet it has the value `null`.

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

When the variable `test` was declared as type `String` space was reserved for an object reference and the value of the `test` variable 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*.



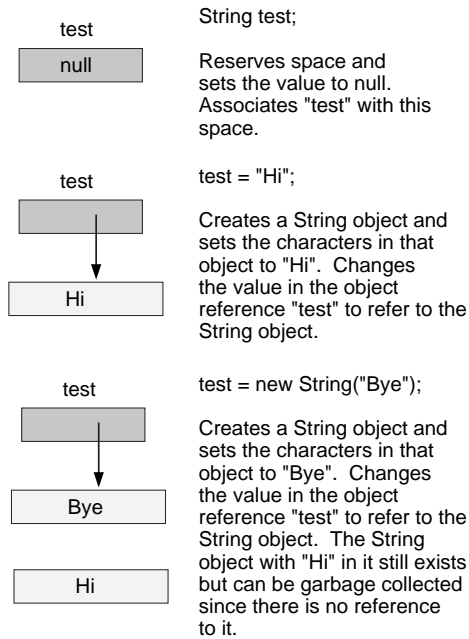


FIGURE 2.7: Declaring object variables and memory assignment

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.

```
> 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).

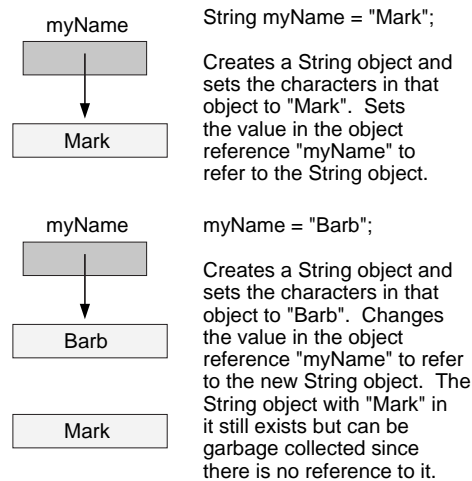


FIGURE 2.8: Shows creation and reuse of an object variable.



**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.

```
> String myName = "Mark";
> System.out.println(myName);
Mark
> String myName = "Sue";
Error: Redefinition of 'myName'
```

The *binding* between the name and the data only exists (a) until the name gets assigned to something else or (b) you quit DrJava or (c) you reset the interactions

pane (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;  
> System.out.println(name2);  
Suzanne Clark
```

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.

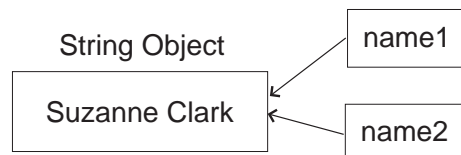


FIGURE 2.9: An object with multiple references to it

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`.

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

36 Chapter 2 Introduction to Java

```
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.

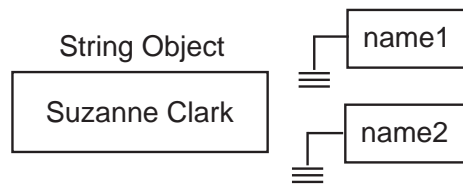


FIGURE 2.10: An object with no references to it

## 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. 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.

```
> 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

+	Addition	Used to add numbers together ( $3 + 4 = 7$ )
-	Subtraction	Used to subtract one number from another ( $5 - 2 = 3$ )
*	Multiplication	Used to multiply two numbers together ( $2 * 3 = 6$ )
/	Division	Used to divide one number by another ( $18 / 2 = 9$ )
%	Modulus (Remainder)	Used to find the remainder of one number divided by another ( $19 \% 2 = 1$ )

### 2.5.2 Relational Operators

<	Less Than	Used to check if one value is less than another ( $2 < 3$ ) is true
>	Greater Than	Used to check if one value is greater than another ( $3 > 2$ ) is true
==	Equals	Used to check if two values are the same ( $2 == 2$ ) is true
!=	Not Equals	Used to check if two values aren't equal ( $2 != 3$ ) is true
<=	Less Than or Equal	Used to check if one value is less than or equal to another ( $2 <= 3$ ) is true
>=	Greater Than or Equal	Used to check if one value is greater than or equal to another ( $3 >= 2$ ) is true

### 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 talked about several kinds of types (encodings) of data.

Floating point numbers	Java primitive types <code>double</code> or <code>float</code> e.g., 5.2, -3.01, 928.3092	Numbers with a decimal point in them.
Integers	Java primitive types <code>int</code> , <code>byte</code> , <code>short</code> , <code>long</code> e.g., -3, 5239, 0	Numbers without a decimal point—they can't represent fractions.
Characters	Java primitive type <code>char</code> e.g. 'a', 'b', '?'	A character is delimited by a pair of single quotes.
Strings	Java <code>String</code> object e.g., "Hello!"	A sequence of characters (including spaces, punctuation, etc.) delimited on either end with a double quote character.
Booleans	Java primitive type <code>boolean</code> with only two possible values	The value of a boolean can be the reserved word <code>true</code> or the reserved word <code>false</code> .

### 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
0
```

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`).

```
> 1.0 / 2
0.5
> (double) 1 / 2
0.5
```

### 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 a reference to an object which allows the address of the object to be determined.

If the object variable doesn't refer to any object yet it has the value `null`. You can change what object a variable 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? What are the differences between object variables and primitive variables?
- What is garbage collection?
- What are relational operators? What are math operators?
- What is a string?

2.2. What objects would you encounter in a bank?

2.3. What objects would you encounter in going to a movie?

2.4. What objects would you encounter in a clothing store?

2.5. What objects are involved in a airplane flight?

2.6. What objects are in your classroom?

2.7. What objects would you encounter when you go to the dentist?

2.8. 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?

2.9. Use the interactions pane to calculate how much money you will make if you work 40 hours at \$13.00 and 10 hours at time and a half?

2.10. 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?

2.11. Test your understanding of Java with the following:

- 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?
- What does the code `System.out.println(2 < 3);` do?
- What does the code `System.out.println(2 == 3);` do?
- What does the code `System.out.println(3 >= 2);` do?

40 Chapter 2 Introduction to Java

- 2.12. What does `int x = 1 / 3; System.out.println(x);` do and why?
- 2.13. What does `double d = 1 / 2.0; System.out.println(d);` do and why?
- 2.14. What does `double d1 = 1 / 3; System.out.println(d1);` do and why?
- 2.15. What does the `double d2 = (double) 1 / 3; System.out.println(d2);` do and why?
- 2.16. 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
- 2.17. Declare variables for each of the following:
- 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
- 2.18. Which of the following is the correct way to declare a variable that represents a price?
- `declare double price = 0;`
  - `int price = 0;`
  - `Integer price = 0.0;`
  - `double PRICE = 0.0;`
  - `double price;`
- 2.19. Which of the following is the correct way to declare a variable that represents the desired quantity of an item in an order?
- `double numItems;`
  - `INT numItems;`
  - `int numItems;`
  - `DOUBLE numItems;`
- 2.20. Which of the following is the correct way to declare a variable that represents if an order has been canceled?
- `BOOLEAN canceled = false;`
  - `boolean canceled = false;`
  - `boolean CANCELED = false;`
  - `boolean canceled = FALSE;`



## 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. If you are someone who wants lots and lots of examples see Deitel and Deitel’s *Java, How to Program*.



## C H A P T E R 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 encodings for, such that those names make the computer do expected actions and interpret our data in expected ways. Some of the programming language's names allow us to define *new* names—which allows us to create our own layers of encoding. We can associate a name with a location in memory, this is called declaring a variable. We can associate a name with a group of Java statements, we call this defining a method (function). In Java you can also assign a name to a group of related variables and methods (functions) when you define a class (type).



**Computer Science Idea: Programs are for people, not computers.**

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

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

There are good names and bad names. Bad names aren't curse words, or TLA's (Three Letter Acronyms), but names that aren't understandable or easy to use. A good set of encodings and names allow one to describe methods in a way that's natural, without having to say too much. The variety of different programming languages can be thought of as a collection of sets of namings-and-encodings. Some are better for some tasks than others. Some languages require you

to write more to describe the same program (function) than others—but sometimes that “more” leads to a much more (human) readable program that helps others to understand what you’re saying.

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

How the units and values (*data*) of a program can be interpreted is often also named. Remember how we said in Section 1.2 (page 6) that everything is stored in groups of eight bits called bytes, and we can interpret those bytes as numbers? In some programming languages, you can say explicitly that some value is a *byte*, and later tell the language to treat it as a number, an *integer* (or sometimes *int*). Similarly, you can tell the computer that these series of bytes is a collection of numbers (an *array of integers*), or a collection of characters (a *String*), or even as a more complex encoding of a single *floating point number* (any number with a decimal point in it).

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

### 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 8)?

Disk capacity per dollar is increasing *faster* than computer speed per dollar! Computer disks today can store whole movies, hours (days?) of sounds, and the equivalent of hundreds of film rolls of pictures.

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

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

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

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

### 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 executed using the class name or on an object of the class. Object methods can only be executed on an object of the class. Class methods are used for general methods that don’t pertain to a particular object. They are defined using the keyword `static`. Object methods work with a particular object’s data (the object the method was called on).

#### 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 5.0). 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 5.0 (also called jdk 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 in Java are specified between single quotes: 'A'. The name of that method is `getNumericValue` and you can use `System.out.println` to display the value that the method `getNumericValue` returns:

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

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

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



### Debugging Tip: Common typos

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

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

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

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

### 3.3.2 Executing Object Methods

Object methods are methods that *must* be executed 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.

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

Notice that the value of `name` didn't change even though we invoked the method `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

Dr. 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 a class as an object factory. The factory can create many objects. A class is also like a cookie cutter. You can make many cookies from one cookie cutter and they will all have the same shape. Or you can think of the class as a blueprint and the objects as the houses that you can create from the blueprint.

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` is to 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.
```



**Common Bug: Finding Classes**  
You should have set your classpath to include the classes from the book in chapter 2. If you didn’t do this you will get an error message (Undefined Class) when you try to create a `World` object. Make sure that the full path to the directory that has the classes from the book is in your classpath. The classpath tells Java where to look for the compiled class definitions. Java needs to load the class definition before it can create an object of a class.

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



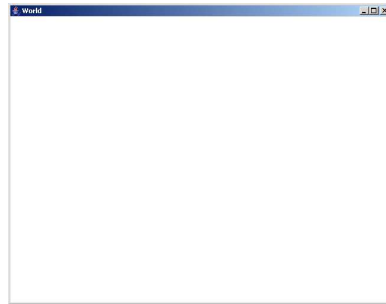


FIGURE 3.1: A window that shows a `World` object.

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 its 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`.

```
> 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 itself.

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

## 50 Chapter 3 Introduction to Programming

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).

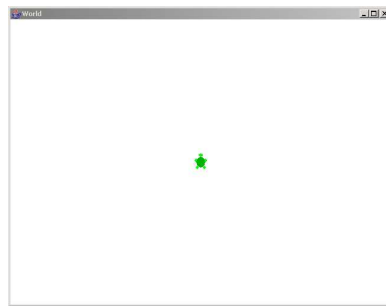


FIGURE 3.2: A window that shows a `Turtle` object in a `World` object.

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.

### 3.4.3 Sending Messages to Objects

We have been talking about executing or invoking methods on classes and objects. A more object-oriented way of saying that is that 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



FIGURE 3.3: A window that shows two `Turtle` objects in a `World` object.

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.

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 we can just ask the method to be executed 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.

```
> 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);
```

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();
```

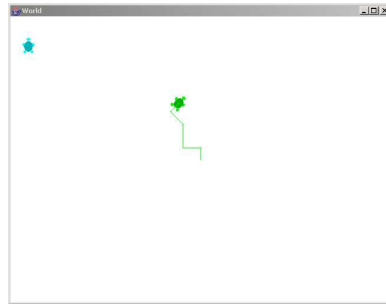


FIGURE 3.4: The result of messages to the first `Turtle` object.

```
> 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?

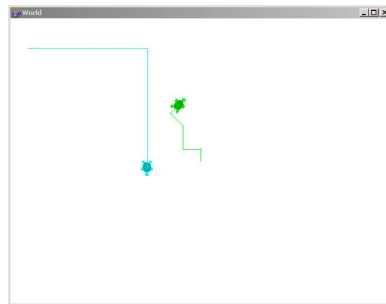


FIGURE 3.5: The result of messages to the second `Turtle` object.

#### 3.4.4 Objects Control Their State

In object-oriented programming we ask an object to do 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 its 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`.

```
> 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.

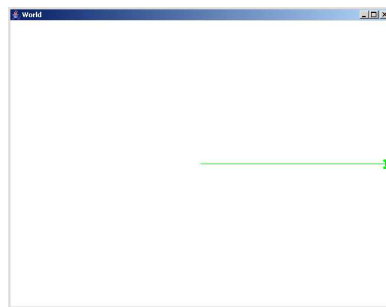


FIGURE 3.6: The turtle won't leave the world

It may seem strange that `turtle1` stopped when it reached 639 but the first pixel is at 0 and the last is 639. If we 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 its 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 `penUp()`. To start showing the trail again send the turtle the message `penDown()`.

You can ask a turtle to move to a particular location by sending it the message `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.

54 Chapter 3 Introduction to Programming

You can ask a turtle to use a particular name by sending it the message `setName(name)` where `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 `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 `new Turtle(x,y,world)` to create a turtle object that is located at (x,y). Let’s turn off seeing the turtle when we draw the second square by sending it the message `hide()`.

```
> 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.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);
> System.out.println(turtle1);
Jane turtle at 200, 200 heading 0.
```

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.

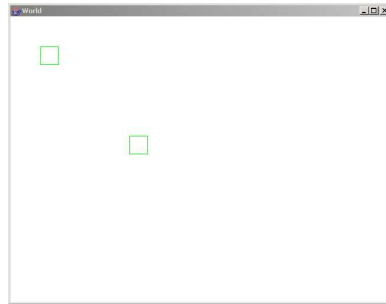


FIGURE 3.7: Drawing two squares with a turtle.

### 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 a new method before you try and use it.

You have seen how you declare variables in Java:

```
type name; or type name = expression;
```


To declare a method in Java use:

```
visibility type methodName(parameterList)
```

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* of statements which is an open curly brace followed by a series of statements followed by a close curly brace. The statements in the block will be executed when the method is invoked.



**Common Bug: Curly Braces Come in Pairs**  
Each open curly brace in your Java code must have a matching close curly brace. You should indent code inside of a pair of curly braces. Indentation doesn't matter to the compiler but makes your code easier to read and understand. Be careful not to mix up curly braces and parentheses.

To declare a method that will draw a square we can use:

```
public void drawSquare ()
{
    // statements to execute when the method is executed
}
```

The visibility in this method declaration is `public`. **Visibility** means who can invoke the method (ask for the method to be executed). The keyword `public` means that 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 in this method declaration is `void`. 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.

The method name in this declaration is `drawSquare`. By convention method names start with a lowercase letter and the first letter of each additional word is uppercase: `drawSquare`. Another example method name is `turnRight`.

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 statements 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 method 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).



**Debugging Tip: Compile All**

The COMPILE ALL button will compile all open files. If you have an empty file open named (UNTITLED) go ahead and close it. DrJava creates this for you so that you can start writing a new class definition. But, we won’t do that right away. Select the name of the file in the Files Pane and press the CLOSE button to close a file.



**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). This is also known as the *scope* of a variable. 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

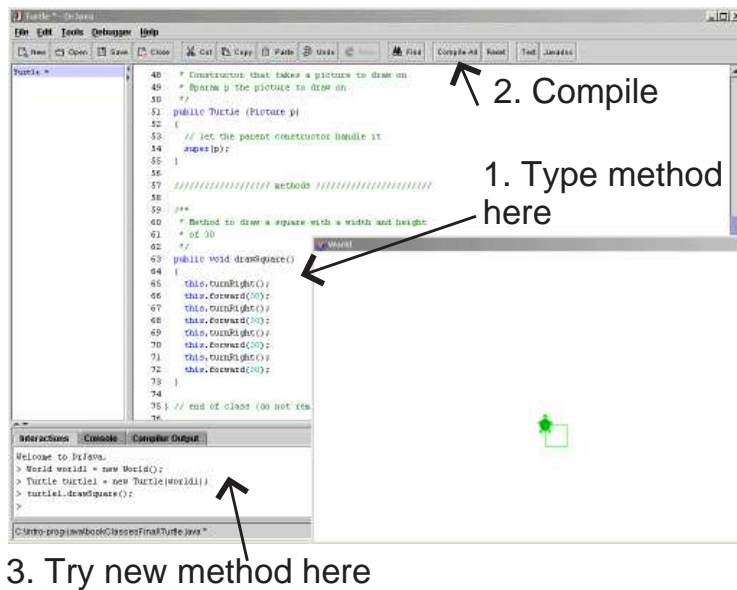



FIGURE 3.8: Defining and executing drawSquare()

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 parentheses 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();
```

When you invoke the method `drawSquare()` on the `Turtle` object referenced by the variable `turtle1` the Java virtual machine has to find the method to execute. Methods are defined inside of a class definition so we defined `drawSquare` inside of the class `Turtle` by editing the file `Turtle.java`. Next we compiled the source file `Turtle.java` which created a `Turtle.class` file which contained the code for the `Turtle` class in byte codes for the Java virtual machine.

The first time you use a class the Java virtual machine loads the compiled class definition (the `Turtle.class` file) and creates an object that contains all the information about the class including the code for the methods. Every object has a reference to the object that defines its class. The object that defines a class is an object of a class named `Class`. You can get the `Class` object using the method `getClass()`.

```
> System.out.println(world1.getClass());  
class World  
> System.out.println(turtle1.getClass());  
class Turtle
```

The Java virtual machine will check for a method with the same name and parameter list in the object that defines the class (an object of the class `Class` and if it is found it will execute that method. This means that the statements in the body of the method will be executed starting with the first statement. The object that the method was invoked on is implicitly passed to the method as well and can be referred to using the keyword `this`.

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.

	<b>Program 3: Draw Square Using a Variable for Width</b>
---	--



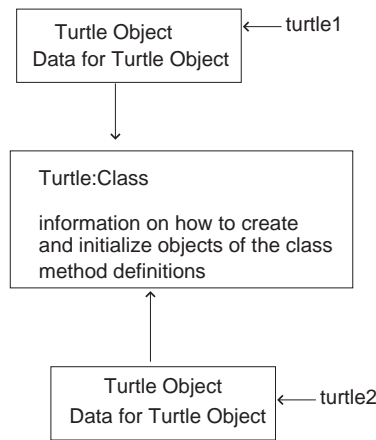


FIGURE 3.9: An object stores data for that object and has a reference to the class that created it

```
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 think 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**

```
/**
 * Method to draw a square with a width and height
 * of some passed amount.
 * @param width the width and height to use
 */
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);
}
```

Type in the new method declaration and compile. Let's try this new method out.

```
> World world1 = new World();
> Turtle turtle1 = new Turtle(world1);
> turtle1.drawSquare(200);
```

When you execute `turtle1.drawSquare(200)`; you are asking the object referred to by the variable named `turtle1` to execute a method named `drawSquare` that takes an integer parameter. The method `drawSquare` will use 200 for the parameter `width` everywhere it appears in the method `drawSquare`. The parameter name `width` is known throughout the body of the method. This is very similar to `drawSquare2()` but has the advantage that we don't need to change the method and recompile to use a different width.

An important reason for using parameters is to make a method more *general*. Consider method `drawSquare(int width)`. That method handles the *general* case of drawing a square. We call that kind of generalization *abstraction*. Abstraction

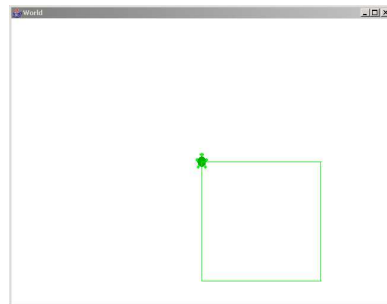


FIGURE 3.10: Showing the result of sending the width as a parameter to `drawSquare`

leads to general solutions that work in lots of situations.



**Making it Work Tip: Use names that make sense**  
We called the first method `drawSquare()` and the second `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, `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 method, by specifying its name with *input values* (also called the *arguments*) inside parentheses, such as `turtle1.drawSquare(20)`; or `new Turtle(20,30,world1)`, each parameter variable is set to a **copy** of the argument value. This is called *pass by value*. All arguments in Java are passed by making a copy of their value. Does this mean that we make a copy of the `World` object when we pass it as a parameter? No, we just make a copy of the object reference which means we make another reference to that `World` object.

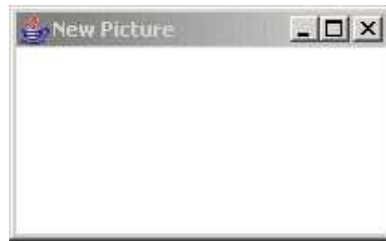


FIGURE 3.11: Creating a Picture object using `new Picture()`

## 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 created 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;` or `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 fully qualified file name of a file to read the picture information from.



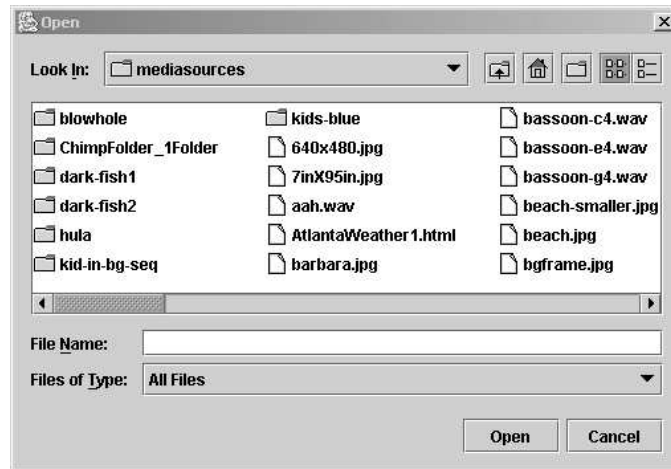



FIGURE 3.12: The File Chooser

What is the *fully qualified 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.

```
> System.out.println(FileChooser.pickAFile());
```



**Common Bug: File Chooser doesn't appear**  
If you don't see the window with the file chooser in it after typing in the code above, try minimizing your DrJava window. Sometimes the file chooser comes up behind the DrJava window.

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

- Double-click on folders/directories to open them.
- Click on the top right iconic button to see the details about the files such as the types of files they are (if you put the cursor over the button and leave it there it will show "Details"). To create a picture we want to pick a file with a type of "JPEG Image". To create a sound we would pick a file with a type of "Wave Sound".
- Click on the file name to select it and then click OPEN, or double-click, to select a file.

Once you select a file, what gets returned is the *full file name* as a string (a sequence of characters). (If you click CANCEL, `pickAFile()` returns `null` which

is a predefined value in Java that means that it doesn't refer to a valid object). 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. “flower1.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

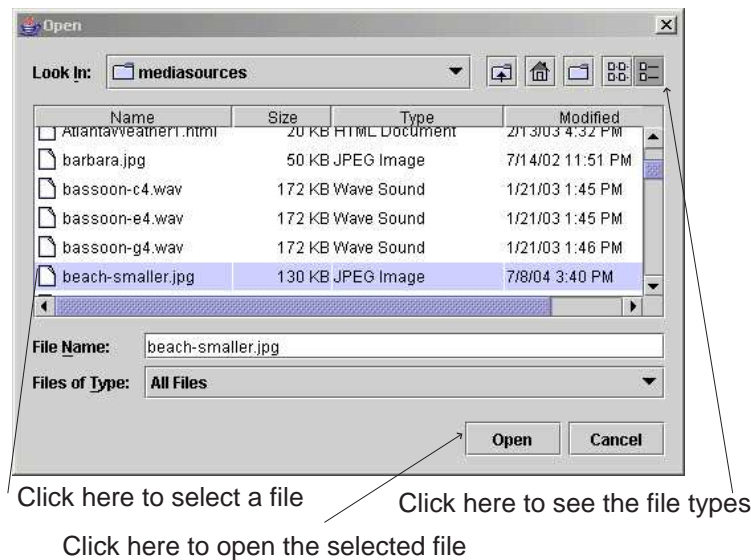


FIGURE 3.13: File chooser with media types identified

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()`.

```
> System.out.println(new Picture(FileChooser.pickAFile()));  
Picture, filename  
c:\intro-prog-java\mediasources\beach-smaller.jpg height 360 width  
480
```

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! 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.

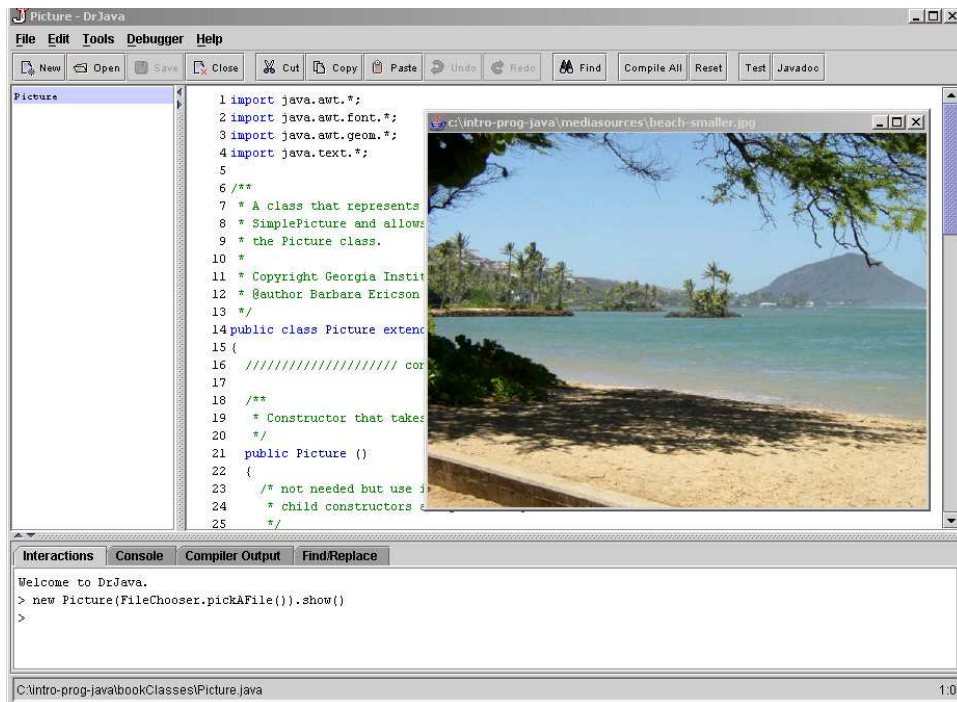


FIGURE 3.14: Picking, making, and showing a picture, using the result of each method in the next method. The picture used is `beach-smaller.jpg`.

- 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”.

```
> 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 ( `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 a uppercase letter and `pictureObj` is a variable name since it starts with a lowercase letter. If a name has several words in it the convention is to uppercase the first letter of each additional word like `pickAFile()`. A convention is the usual way of doing something which means that the compiler won’t care if you don’t do it this way but other programmers will tar and feather you because it will make your programs harder to understand.



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

In Java all methods 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 `pictureObj.show()` not `pictureObj.show`.

If you try `pictureObj.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. If the type is `int` then 32 bits of space (4 bytes) is reserved. If the type is `double` then 64 bits of space (8 bytes) is reserved.

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

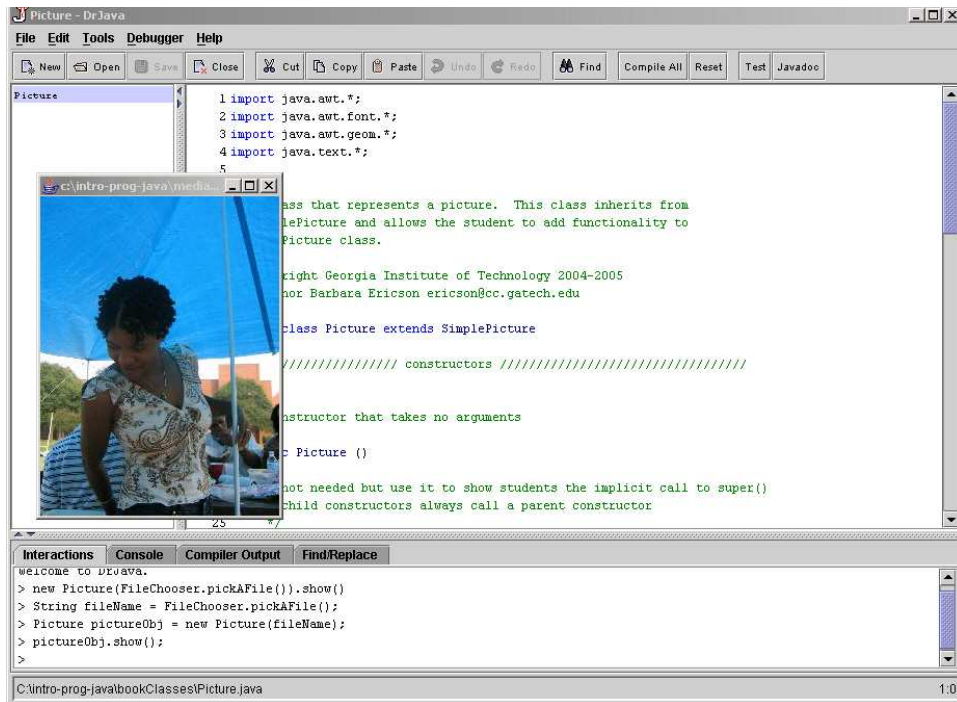


FIGURE 3.15: Picking, making, and showing a picture, when naming the pieces. The picture shown is tammy.jpg. Tammy is one of the computer science graduate students at Georgia Tech.

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();
```

The `FileChooser.pickAFile()`; allows you to pick a file with a file chooser and the `System.out.println` that is around this displays the full file name that was picked. The code `System.out.println(new Sound(FileChooser.pickAFile()))`; also allows you to pick a file, then it creates a sound object from the full file name, and finally it 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 chapter 8. 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
```

Notice that we named the `String` returned from `FileChooser.pickAFile`. We can use that name many times and each time it will have the same value (until we change it).

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. Executing the code:

```
Picture myPicture = new Picture(myFileName);
```

causes the exact same picture to be created as if we had executed:

```
Picture myPicture = new Picture(FileChooser.pickAFile());1
```

because we set `myFileName` to be equal to the result of `FileChooser.pickAFile()`. The values get substituted for the names when the expression is evaluated. The code `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 method 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.



#### Common Bug: Backslashes and Slashes

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

```
> 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
```

<sup>1</sup>Assuming, of course, that you picked the same file.

```
360 width 381
```

Or even substitute for the name.

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



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

We can substitute a value, a name assigned to that value (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 call statements to the computer that are telling it to do things *commands*. `System.out.println(aPicture);` is a command. So is `String myFileName = FileChooser.pickAFile();`, and `aPicture.show();`. These are more than expressions: They're telling the computer to *do* something.

### 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.

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

Here is an example of invoking an object method:

```
> turtle1.turnLeft();
```

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.

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

Here is an example of invoking a class method:

```
> System.out.println(Math.abs(-3));  
3
```

Class methods are used for general methods like absolute value. 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. This is also called creating an *instance* of a class or *instantiating* an object.

```
new ClassName(parameterList)
```

Here is an example of creating an object of the class `World`:

```
> World worldObj = new World();
```

### 3.7.4 Creating new Methods

To create a method in a class, 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:

```
public returnType methodName(parameterList)
{
    // statements 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.

Here is an example method in the `Turtle` class:

```
/**
 * Method to draw a square with a width and height
 * of 30
 */
public void drawSquare()
{
    this.turnRight();
    this.forward(30);
    this.turnRight();
    this.forward(30);
    this.turnRight();
    this.forward(30);
    this.turnRight();
    this.forward(30);
}
```

## OBJECTS AND METHODS SUMMARY

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

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

Here are the methods introduced in this chapter:

Section 3.7 Concepts Summary 77

<code>Character.getNumericValue(Character character)</code>	Returns the equivalent numeric value in Unicode for the input character.
<code>FileChooser.pickAFile()</code>	Lets the user pick a file and returns the complete path name as a string.
<code>Math.abs(int number)</code>	Takes a number and returns the absolute value of it.
<code>show()</code>	Shows the <code>Picture</code> object that it is invoked on. No return value.
<code>play()</code>	Plays the sound object (object of the <code>Sound</code> class) that it is invoked on.
<code>forward(int numberOfSteps)</code>	Asks the <code>Turtle</code> object that it is invoked on to move forward by the passed number of steps. No return value.
<code>setPenDown(boolean value)</code>	Asks the <code>Turtle</code> object that it is invoked on to set the pen up or down depending on the passed value. If you pass in <code>false</code> for value the pen is lifted and no trail will be drawn when the turtle moves. If you pass in <code>true</code> the pen will be put down and the trail will be drawn.
<code>hide()</code>	Asks the <code>Turtle</code> object that it is invoked on to stop showing itself. No return value.
<code>moveTo(int x, int y)</code>	Asks the <code>Turtle</code> object that it is invoked on to move to the specified x and y location. No return value.
<code>penDown()</code>	Asks the <code>Turtle</code> object that it is invoked on to put down the pen and draw the trail of future movements. No return value.
<code>penUp()</code>	Asks the <code>Turtle</code> object that it is invoked on to pick up the pen so you don't see the trail of future movements. No return value.
<code>show()</code>	Asks the <code>Turtle</code> object that it is invoked on to show (draw) itself. No return value.
<code>turn(int angle)</code>	Asks the <code>Turtle</code> 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.
<code>turnLeft()</code>	Asks the <code>Turtle</code> object that it is invoked on to turn left 90 degrees. No return value.
<code>turnRight()</code>	Asks the <code>Turtle</code> object that it is invoked on to turn right 90 degrees. No return value.

## PROBLEMS

3.1. Some computer science concept questions:

- What is a file?
- What is an operating system?
- What does a compile do?
- What does method visibility mean?
- What is a classpath?
- What is a wrapper class?
- What is a hard disk?
- What is a method?
- What creates new objects?
- What does “pass by value” mean?
- What is a primitive variable?
- What is an object variable?

3.2. Test your understanding of Java with the following:

- What does `pictureObj.show()` do?
- What does `soundObj.play()` do?
- What does `FileChooser.pickAFile()` do?
- What does `turtle1.turnLeft()` do?

3.3. Test your understanding of Java with the following:

- What does `turtle1.forward()` do?
- What does `turtle1.turn(-45)` do?
- What does `turtle1.turn(45)` do?
- What does `turtle1.penUp()` do?
- What does `turtle1.hide()` do?

3.4. How do you create new objects in Java? How do you create a `World` object? How do you create a `Turtle` object?

3.5. Which of the following are class methods and which are object methods? How can you tell which are which?

- `Math.abs(-3);`
- `soundObj.play();`
- `FileChooser.pickAFile();`
- `pictureObj.show();`
- `ColorChooser.pickAColor();`
- `turtle1.turnLeft();`

3.6. What does this do? `System.out.println(new Picture());`

3.7. How many and what kind of variables (primitive or object) are created in the code below?

```
> String fileName = FileChooser.pickAFile();  
> Picture p1 = new Picture(fileName);  
> p1.show();
```

- 3.8. How many and what kind of variables (primitive or object) are created in the code below?
- ```
> World worldObj = new World();  
> Turtle turtle1 = new Turtle(worldObj);  
> turtle1.forward(30);  
> Turtle turtle2 = new Turtle(worldObj);  
> turtle2.turnRight();  
> turtle2.forward(30);
```
- 3.9. How many and what kind of variables (primitive or object) are created in the code below?
- ```
> double cost = 19.20;  
> double percentOff = 0.4;  
> double salePrice = cost * (1.0 - percentOff);
```
- 3.10. 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.
- 3.11. Write a method for `Turtle` to draw a rectangle. Pass in the width and height for the rectangle.
- 3.12. Write a method for `Turtle` to draw a hexagon. Pass in the length of the sides.
- 3.13. Write a method for `Turtle` to draw a pentagon. Pass in the length of the sides.
- 3.14. Write a method for `Turtle` to draw an equilateral triangle. Pass in the length of the sides.
- 3.15. Create a `World` object and a `Turtle` object and use the `Turtle` object to draw a star.
- 3.16. Create a `World` object and a `Turtle` object and use the `Turtle` object to draw an arrow.
- 3.17. Create a `World` object and a `Turtle` object and use the `Turtle` object to draw a pyramid.
- 3.18. Create a `World` object and a `Turtle` object and use the `Turtle` object to draw a flower.
- \*3.19. Create a `World` object and a `Turtle` object and use the `Turtle` object to draw a house.
- \*3.20. Create a `World` object and a `Turtle` object and use the `Turtle` object to draw your first name.

## 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* [18] 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

## C H A P T E R 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.

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.

If we want to write programs to manipulate JPEG images we need to understand how they are stored and displayed. To do this we need to understand arrays, matrices, pixels, and color.

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. Why is the index based on the distance? Array values are stored one after the other in memory. This makes it easy to find any element of the array by multiplying the size of each element by the index and adding it to the address of the beginning of the array. If you are looking for the element at index 3 in an array and the size of each element is 4 bytes long and the array starts at memory location 26 then the 3rd element is at  $(3 * 4 + 26 = 12 + 26 = 38)$ .

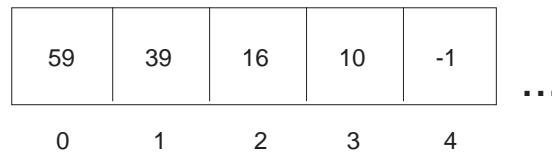


FIGURE 4.1: A depiction of the first five elements in an array

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 hundreds of 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

84 Chapter 4 Modifying Pictures using Loops

access the last element in the array use `arrayName[arrayName.length - 1]`.

To declare an array in Java you specify the type and then use open and close square brackets followed by a name for the array.

```
> double[] grades;  
> System.out.println(grades);  
null
```

or you could have specified the square brackets after the variable name:

```
> double grades[];  
> System.out.println(grades);  
null
```

The above code declares an array of doubles with the name `grades`. Notice though that this just declared an object reference and set it to null. *It didn't create* the array. In Java you can create an array and specify the values for it at the same time:

```
> double[] gradeArray = {80, 90.5, 88, 92, 94.5};  
> System.out.println(gradeArray.length);  
5  
> System.out.println(gradeArray[0]);  
80.0  
> System.out.println(gradeArray[4]);  
94.5
```



**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 can talk about an element at row *r* and column *c*, that is, `matrix[r][c]`. This is called *row-major order*.

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 4.2). Have you ever gone to a play? Usually your ticket has a row and seat number. These are both examples of row-major two-dimensional arrays.

Another way to specify a location in a two-dimensional array is *column-major order* which specifies the column first and then the row: `matrix[c][r]`. This is



FIGURE 4.2: The top left corner of a battleship guess board with a miss at B-3.

how we normally talk about pictures by using an  $x$  for the horizontal location and a  $y$  for the vertical location such as `matrix[x][y]`. Picture data is represented as a column-major two-dimensional array.

Java actually creates multi-dimensional arrays as arrays of arrays. When you have a two-dimensional array the first index is the location in the outer array and the second is the location in the inner array. You can either think of the outer array as being the rows or the outer array as being the columns. So, Java isn't row-major or column-major, but you will create and work with your arrays in either row-major or column-major fashion. Just be sure to be consistent.

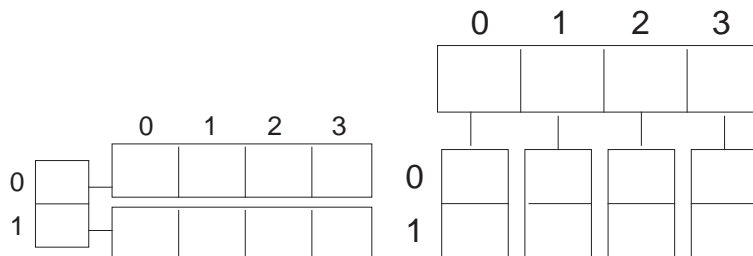


FIGURE 4.3: Picturing a 2-d array as row-major or column-major

In Figure 4.4, you see an example matrix. Using column-major order for the *coordinates*  $(0,0)$  (horizontal, vertical), you'll find the matrix element whose value is 15. The element at  $(1,1)$  is 7,  $(2,1)$  is 43, and  $(3,1)$  is 23. We will often refer to these coordinates as  $(x,y)$  (*horizontal, vertical*).

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 4.5 was generated by capturing as an image the top left part of the DrJava window and then magnifying it 600%. It's made up of many, many dots. When

	0	1	2	3
0	15	12	13	10
1	9	7	43	23
2	6	13	15	16

FIGURE 4.4: An example matrix (two-dimensional array) of numbers

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.

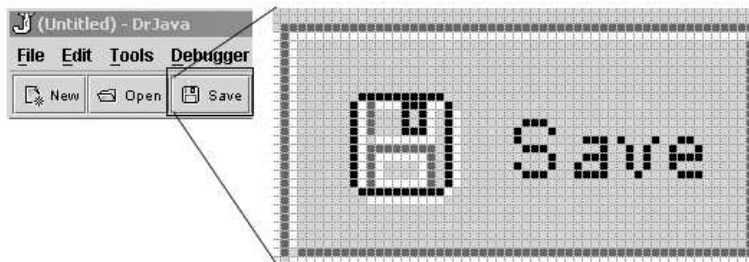


FIGURE 4.5: Upper left corner of DrJava window with part magnified 600%

You can get a similar view of individual pixels using the picture explorer, which is discussed later in this chapter. The picture explorer allows you to zoom a picture up to 500% so that each individual pixel is visible (Figure 4.6).

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

That lack of resolution in human vision is what makes it possible to digitize pictures. Animals that perceive greater details than humans (e.g., eagles or cats) may actually see the individual pixels. We break up the picture into smaller elements (pixels), but there are enough of them and they are small enough that the picture doesn't look choppy when viewed from a normal viewing distance. 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

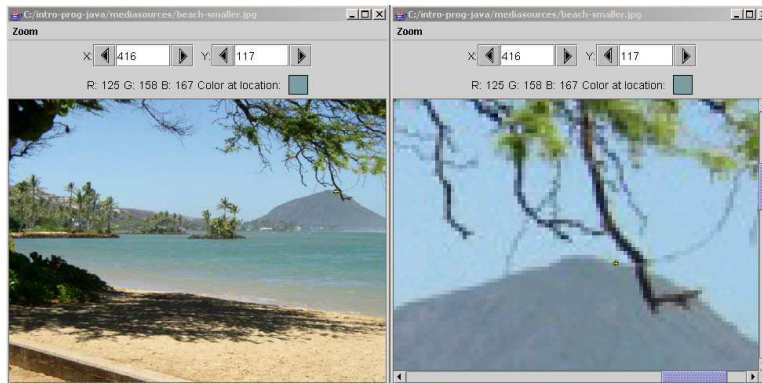


FIGURE 4.6: Image shown in the picture explorer: 100% image on left and 500% on right (close-up of the branch over the mountain)

process becomes obvious.

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

#### 4.1.1 Color Representations

Visible light is 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 color we “see” 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 4.7). 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

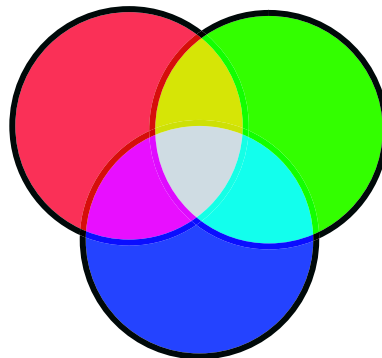


FIGURE 4.7: Merging red, green, and blue to make new colors

“lighter” or “darker” map cleanly to it, e.g., you simply change the saturation (Figure 4.8). 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.

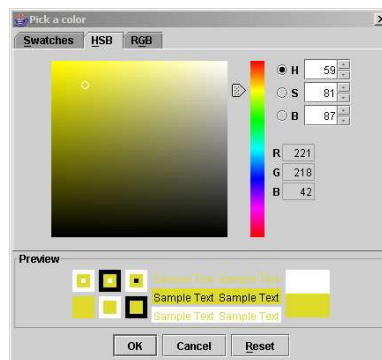


FIGURE 4.8: Picking colors using the HSB color model

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 ( $2^8$ ): 0000000, 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 4.9 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 4.9: 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 4.10). The color chooser in Java offers a similar set of sliders (Figure 4.11).

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 4.12 is a representation of pixel RGB triplets in a matrix representation. In column-major order 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

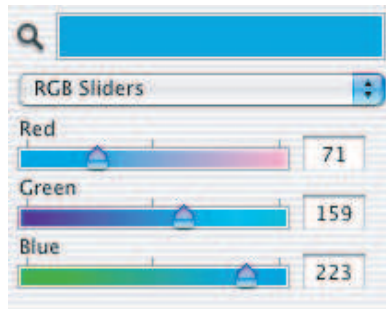


FIGURE 4.10: The Macintosh OS X RGB color picker

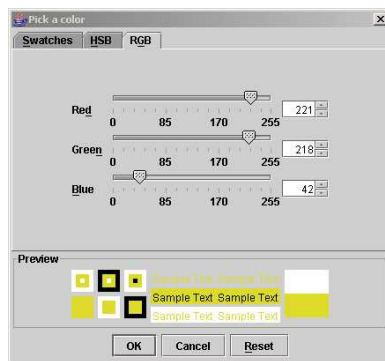


FIGURE 4.11: Picking a color using RGB sliders from Java

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 wide by 240 pixels high, 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

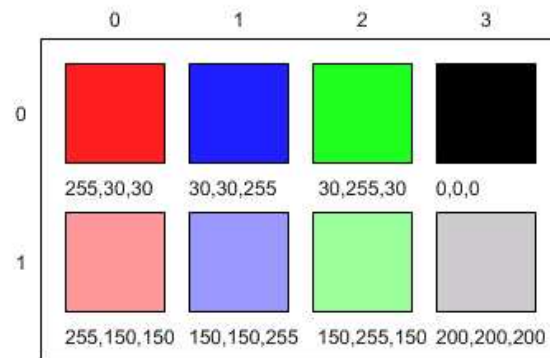



FIGURE 4.12: RGB triplets in a matrix representation

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

	320x240 image	640x480	1024x768
24-bit color	230,400 bytes	921,600 bytes	2,359,296 bytes
32-bit color	307,200 bytes	1,228,800 bytes	3,145,728 bytes

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 have used it to mean 2 to the 10th 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 20th power and a megabyte is 1,000,000 bytes (one million bytes). A gibibyte is defined to be 2 raised to the 30th 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

92 Chapter 4 Modifying Pictures using Loops

`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()`.

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

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

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

```
> 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. This returns an object of the class `Pixel` which knows the picture it is from and the `x` and `y` position of the pixel in that picture. 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.

```
> 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
```

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

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

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

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

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

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

variables (fields) can be accessed using *ClassName.fieldName*.



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

Color is a Java class in the package java.awt. A package is a group of related classes. 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 Color class is java.awt.Color. You can always use the fully qualified name instead of importing but people don't usually want to type that much. To import all classes in the package java.awt use `import java.awt.*;`. To import just the Color class from the package java.awt use `import java.awt.Color;`. Importing doesn't make your class larger, it is just used to determine what class you mean.




#### **Debugging Tip: Undefined Class Error**

If you get the message “Error: Undefined class Color” it means that you didn't import the class Color. You must either import classes that are in packages other than java.lang or fully qualify them.

```
> 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.

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



**Common Bug: Not 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. Asking the picture to show itself again `pictureObject.show()` will also repaint it.

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 methods do it for you.)

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

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

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

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

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

```
> pictureObject.write("newPicture.jpg");
```



**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 methods 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.

```
> import java.awt.Color;  
> String fName = "C:/intro-prog-java/mediasources/caterpillar.jpg";  
> Picture picture = new Picture(fName);  
> 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 4.13. The black line is 100 pixels down, and the pixels 10 through 19 from the left edge have been turned black.

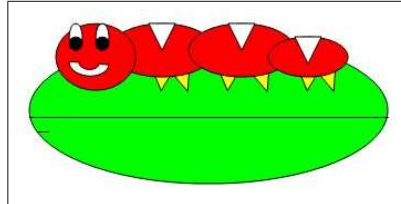


FIGURE 4.13: Directly modifying the pixel colors via commands: Note the small black line on the left under the line across the leaf

### 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 4.14 and the *MediaTools* application appears in Figure 4.15. 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.

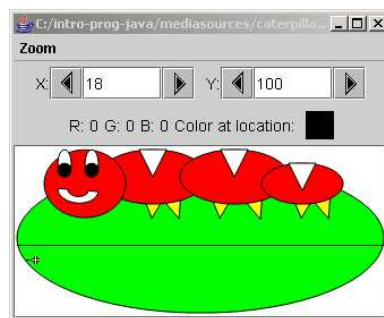


FIGURE 4.14: Exploring the caterpillar with the line

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

the pixel that you want to examine. You can also type in the x and y values and press 'Enter' to see the pixel information for a particular pixel.

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

- The red, green, and blue values will be displayed for the pixel you're pointing at. This is useful when you want to get a sense of how the colors in your picture map to numeric red, green, and blue values. It's also helpful if you're going to be doing some computation on the pixels and want to check the values.
- The x and y position will be displayed for the pixel you're pointing at. This is useful when you want to figure out regions of the screen, e.g., if you want to process only part of the picture. If you know the range of x and y coordinates where you want to process, you can tune your program to reach just those sections.
- Finally, a magnifier is available to let you see the pixels magnified. (The magnifier can be clicked and dragged around.)



FIGURE 4.15: Using the MediaTools image exploration tools on barbara.jpg

### 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 a For-Each Loop

We know that we can use the `getPixels()` method to get an array of `Pixel` objects from a `Picture` object. We can use the `getRed()` method to get the red value from a `Pixel` object, then we can multiply it by 0.5 to decrease the red value, and then we can use `setRed()` to set the red value of a `Pixel` object.

We will need to cast back to integer after we multiply the red value by 0.5. Remember that if the computer sees you using a `double` value it assumes that the result should be a `double`. However, pixel color values must be integers. We could write the code to change the first three pixels like this:

```
> String fName = "C:/intro-prog-java/mediasources/caterpillar.jpg";
> Picture pict = new Picture(fName);
> pict.show();
> Pixel[] pixelArray = pict.getPixels();
> Pixel pixelObj = pixelArray[0];
> int red = pixelObj.getRed();
> red = (int) (red * 0.5);
> pixelObj.setRed(red);
> pixelObj = pixelArray[1];
> red = pixelObj.getRed();
> red = (int) (red * 0.5);
> pixelObj.setRed(red);
> pixelObj = pixelArray[2];
> red = pixelObj.getRed();
> red = (int) (red * 0.5);
> pixelObj.setRed(red);
> pict.explore();
```

This only changes the first three pixels. We don’t want to write out statements like this to change *all* of the pixels in the array even for a small picture. We need some way to repeat the statements that get the red value, change it, and then set the red value for each pixel in the array. As of Java 5.0 (1.5) we can do that using a for-each *loop*. A loop is a way to repeat a statement or a block of statements. The syntax for a for-each loop is

```
for (Type variableName : array)
```

You can read this as “first declare a variable that will be used in the body of the loop” then “for each element in the array execute the body of the loop.” The

100 Chapter 4 Modifying Pictures using Loops

body of the loop can be either one statement or a series of statements inside of an open curly brace '{' and a close curly brace '}'. The statements in the body of the loop are indented to show that they are part of the loop. A method that will loop through all the pixels in the current picture and set the red value in each to half the original value is:

```
public void decreaseRed()
{
    Pixel[] pixelArray = this.getPixels();
    int value = 0;

    // loop through all the pixels in the array
    for (Pixel pixelObj : pixelArray)
    {

        // get the red value
        value = pixelObj.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
        pixelObj.setRed(value);
    }
}
```

If you are using Java 5.0 (1.5) or above add the `decreaseRed()` method to the `Picture.java` file before the last closing curly brace '}'. Then click the `COMPILE ALL` button in DrJava to compile the file. You can try this method out by typing the following in the interactions pane.

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

You can compare the original picture with the changed picture. Use the picture explorers to check that the amount of red was decreased.

When you execute `pict.decreaseRed()` the Java runtime checks the `Picture` class to see if it has a `decreaseRed()` method. The `Picture` class does have this method so it will execute that method and implicitly pass in the `Picture` object the method was invoked on. The keyword `this` is used to refer to the object the method was invoked on (the one referred to by the variable `pict`).

The first time through the loop the `pixelObj` will refer to the first element of the array (the one at index 0). The second time through the loop the `pixelObj` will refer to the second element of the array (the one at index 1). The last time through the loop the `pixelObj` will refer to the last element of the array (the one at index `(length - 1)`).

For-each loops are very useful for looping through each of the elements in an array. If you are using Java 1.4 you can't use a for-each loop. You can use a

`while` loop instead. Even if you are using Java 5.0 `while` loops can help you solve problems that for-each loops can't solve.

### 4.3.2 Using While Loops

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

The syntax for a `while` loop is:

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

Let's talk through the pieces here.

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

Tell someone to clap their hands 12 times. Did they do it right? How do you know? In order to tell if they did it right you would have to count each time they clapped and when they stopped clapping your count would be 12 if they did it right. A 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.

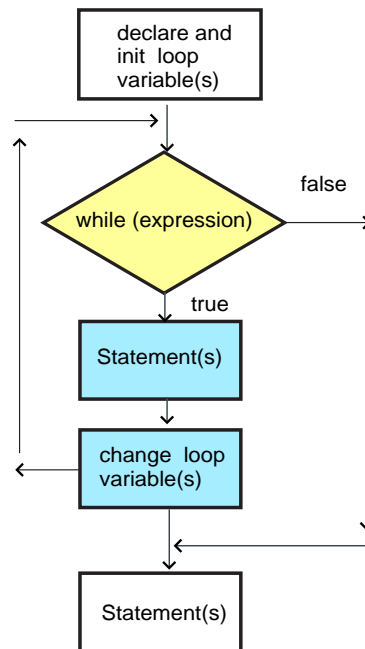


FIGURE 4.16: Flowchart of a while loop



### Computer Science Idea: Flowcharts

Figure 4.16 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.
```

	<p><b>Debugging Tip: Stopping an Infinite Loop</b></p> <p>If you forget to increment the count in the body of the <code>while</code> loop, or if you close the body of the <code>while</code> loop before the count is incremented you will have an <i>infinite loop</i>. 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.</p>
--	--

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.

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

104 Chapter 4 Modifying Pictures using Loops

```
    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 stored in it:  
`C:/intro-prog-java/mediasources/caterpillar.jpg.`
- The variable `pict` is created and refers to the new `Picture` object created from the picture information in the file named by the variable `fileName`.
- We tell the `Picture` object to show (display) itself using `pict.show();`
- Next we declare a variable `pixelArray` that references an array of `Pixel` objects (`Pixel[]`). We get the array of `Pixel` objects by asking the `Picture` object for them using the `getPixels()` method.
- We declare an object variable, `Pixel pixel`, that will refer to a pixel object but initialize it to `null` to show that it isn’t referring to 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

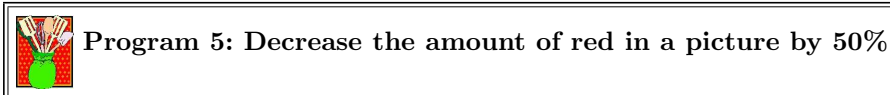


without writing thousands of individual lines, let’s do something useful with this.

### 4.3.3 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.



```

/**
 * 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 COMPILER 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.




**Common Bug: Methods with the same name**  
If you added the method `decreaseRed` with a for-each loop in it to your `Picture.java` source code you will get an error when you add this `decreaseRed` method and compile. You can't have two methods with the same name and parameter list in a class. Just rename the first `decreaseRed` method to `decreaseRedForEach` and compile again.

Unlike some other computer languages Java doesn't compile into *machine code* which is the language for the machine it is running on. When we compile Java source code we compile it into a language for a *virtual machine* which is a machine that doesn't necessarily exist.

When we successfully compile a `ClassName.java` file the compiler outputs a `ClassName.class` file which contains the instructions that a Java virtual machine can understand. If our compile is not successful we will get error messages that explain what is wrong. We have to fix the errors and compile again before we can try out our new method.

When we execute a Java class the Java virtual machine will read the compiled code and map the instructions for the virtual machine to the machine it is currently executing on. This allows you to compile Java programs on one type of computer and run them on another without having to recompile.



**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:

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

Section 4.3 Changing color values 107

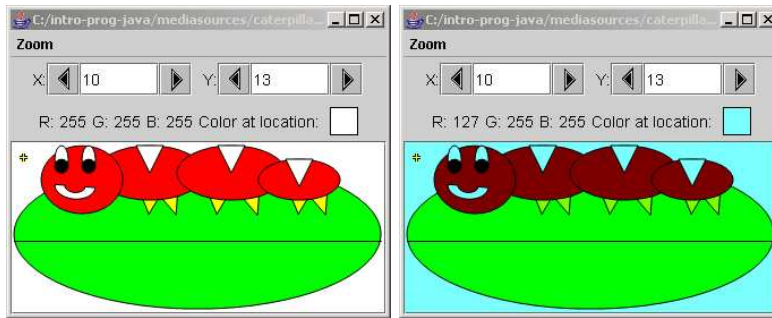


FIGURE 4.17: The original picture (left) and red-decreased version (right)

```
> picture.show();  
> picture.decreaseRed();  
> picture.repaint();
```



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

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

The original picture and its red-decreased version appear in Figure 4.17. 50% is obviously a *lot* of red to reduce! The picture looks like it was taken through a blue filter.



**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 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()`

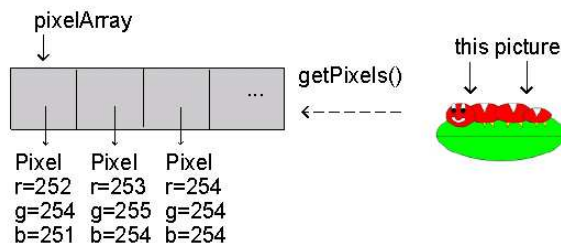
```
> String fileN = "C:/intro-prog-java/mediasources/caterpillar.jpg";  
> Picture picture = new Picture(fileN);  
> picture.show();  
> picture.decreaseRed();
```

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`.

The first line we execute in Program 5 (page 105) is `Pixel[] pixelArray = this.getPixels()`. Let's break this down.

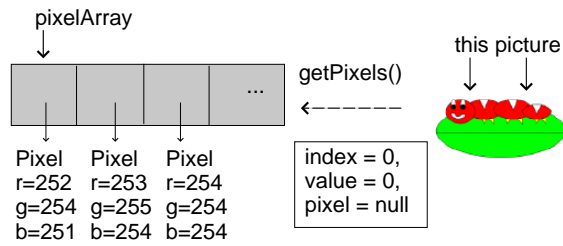
- The `Pixel[] pixelArray` is a declaration of a variable `pixelArray` that references an array of `Pixel` objects. The '=' means that the variable `pixelArray` will be initialized to the result of the right side expression which is a call to the method `this.getPixels()` which returns a one-dimensional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



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 elements are indexed starting with 0 and ending at the length of the array minus one.

Variables that you declare inside methods are not automatically initialized for you so you **should** initialize them when you declare them.



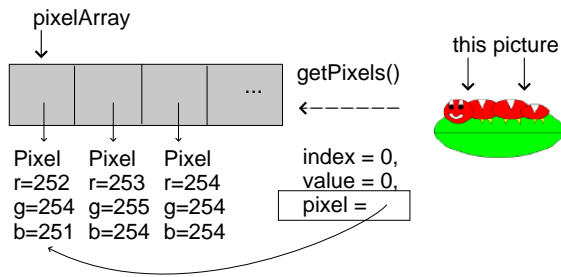
### Computer Science Idea: Scope

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

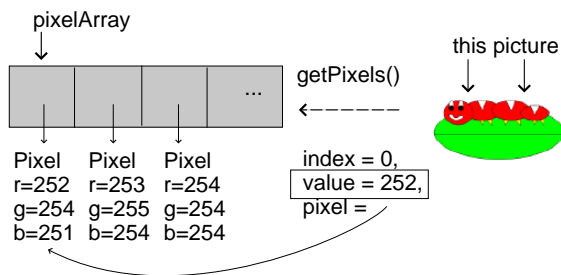
Next comes the loop `while (index < pixelArray.length)`. This tests if the value of the variable `index` is less than the length of the array of pixels referred to by `pixelArray`. If the test is true the body of the loop will be executed. The body of the loop is all the code between the open and close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

Section 4.3 Changing color values 111



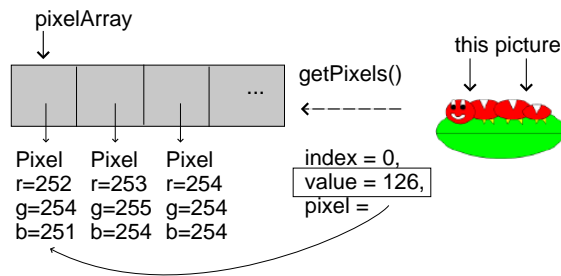
Next in the body of the loop is `value = pixel.getRed();`. This sets the variable `value` to the amount of red in the current pixel. Remember that the amount of red can vary from a minimum of 0 to a maximum of 255.



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

112 Chapter 4 Modifying Pictures using Loops

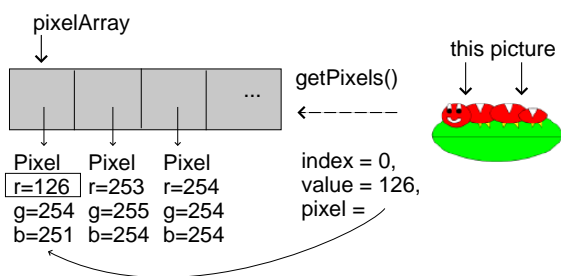
an integer number. Any numbers after the decimal point will be discarded. We do this because colors are represented as integers. The `(int) (value * 0.5)` is needed because the variable `value` is declared of type `int` and yet the calculation of `(value * 0.5)` contains a floating point number and so 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`.



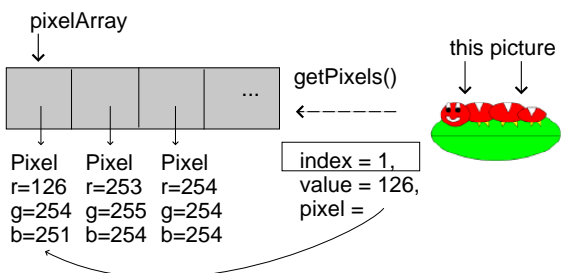
The next step in the body of the loop is `pixel.setRed(value);`. This changes the amount of red in the current pixel to be the same as what is stored in variable `value`. The current pixel is the first one so we see that the red value has changed from 252 to 126 after this line of code is executed.



Section 4.3 Changing color values 113



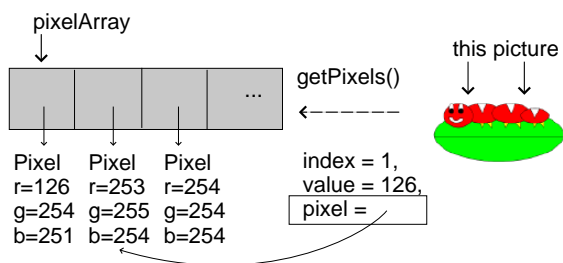
After the statements in the body of the loop are executed the `index = index + 1;` will be executed which will add one to the current value of `index`. Since `index` was initialized to 0 this will result in `index` holding the value 1.



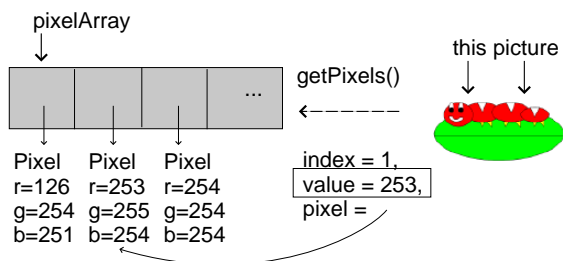
What happens next is very important. The loop starts over again. The continuation test will again check that the value in variable `index` is less than the length of the array of pixels and since the value of `index` is less than the length of the array, the statements in the body of the loop will be executed again. The

114 Chapter 4 Modifying Pictures using Loops

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`.



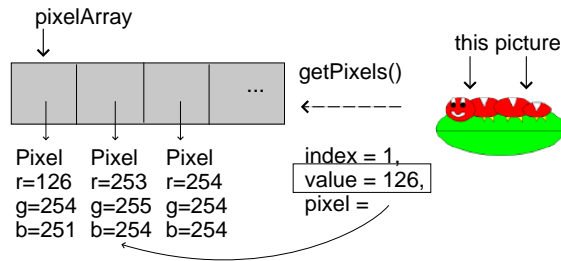
The variable `value` will be set to the red amount in the current pixel referred to by the variable `pixel`, which is 253.



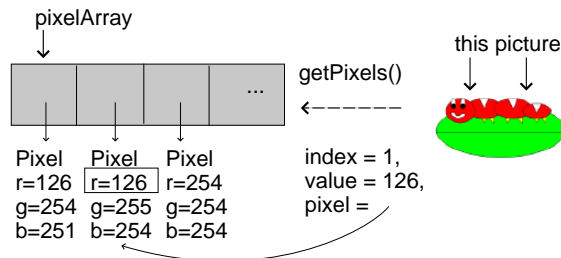
The variable `value` will be set to the result of casting to integer the result of multiplying the amount in `value` by 0.5. This results in  $(253 * 0.5) = 126.5$  and after we drop the digits after the decimal this is 126. We drop the digits after the

Section 4.3 Changing color values 115

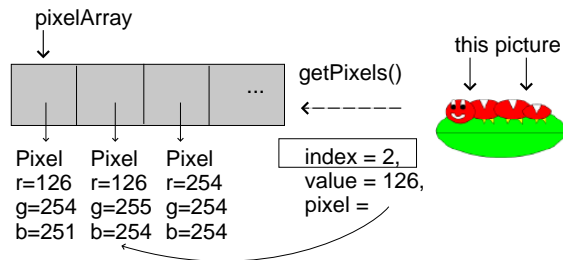
decimal point because of the cast to the type `int` (integer). We cast to integer because colors are represented as integer values from 0 to 255.



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 4.17 (and at Figure 4.18). We keep going through all the pixels in the sequence and changing all the red values.

**Testing the program: Did that really work?.**

How do we know that that really worked? Sure, *something* happened to the picture, but did we really decrease the red? By 50%?



**Making it Work Tip: Don't just trust your programs!**

It's easy to mislead yourself that your programs worked. After all, you told the computer to do a particular thing, you shouldn't be surprised if the computer did what you wanted. But computers are really stupid—they can't figure out what you want. They only do what you 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 methods that we know in the Interactions pane to check

Section 4.3 Changing color values 117

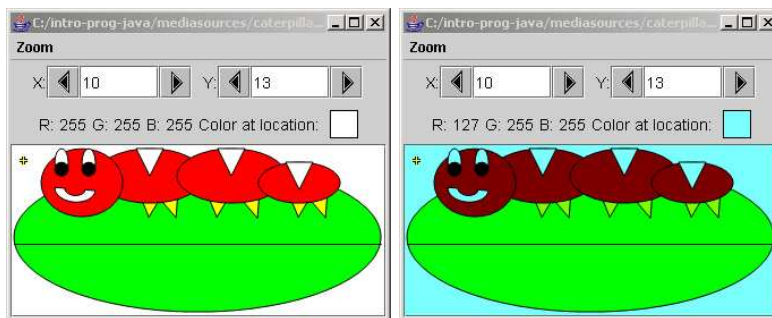


FIGURE 4.18: Using the picture explorer to convince ourselves that the red was decreased

the red values of individual pixels.

```
> String fName = "C:/intro-prog-java/mediasources/caterpillar.jpg";
> Picture pict = new Picture(fName);
> 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 4.19).

**Program 6: Increase the red component by 30%**

```
/**
 * 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 way 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 value.



#### **Making it Work Tip: Shortcuts for Increment and Decrement**

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

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

Section 4.3 Changing color values 119

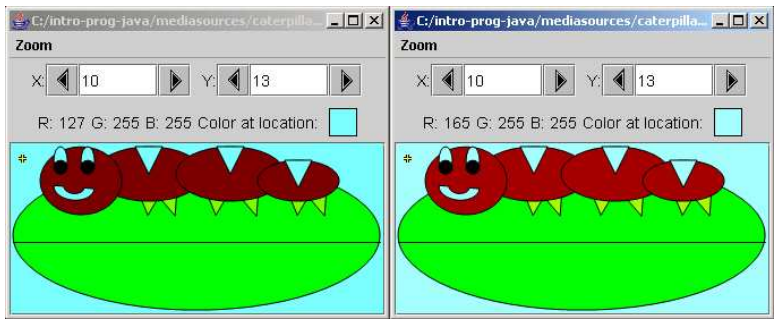


FIGURE 4.19: Overly blue (left) and red increased by 30% (right)

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

We can even get rid of a color completely. The method below erases the blue component from a picture by setting the blue value to 0 in all pixels(Figure 4.20).

 **Program 7: Clear the blue component from a picture**

```
/**
 * 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++;
    }
}
```

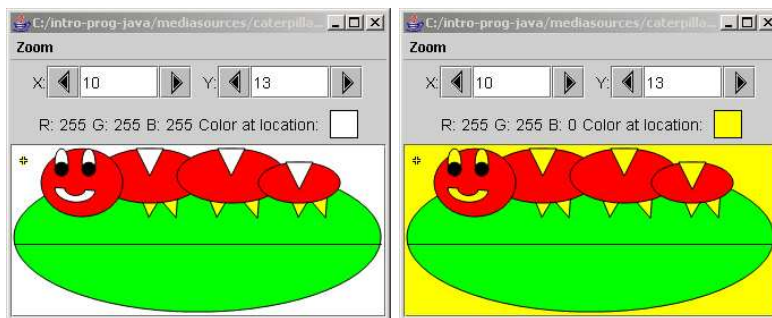


FIGURE 4.20: Original (left) and blue erased (right)

```
}  
}
```

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

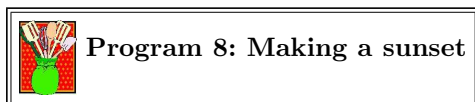
```
> String fName = "C:/intro-prog-java/mediasources/caterpillar.jpg";  
> Picture picture = new Picture(fName);  
> picture.explore();  
> picture.clearBlue();  
> picture.explore();
```

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.

### 4.3.4 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:



```
/**  
 * 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++;
    }
}
```



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

What we see happening in Program 8 (page 120) is that we’re changing both the blue and green channels—reducing each by 30%. The effect works pretty well, as seen in Figure 4.21.

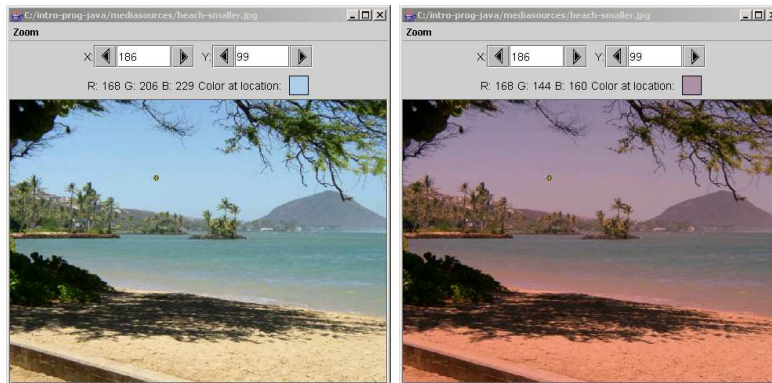


FIGURE 4.21: Original beach scene (left) and at (fake) sunset (right)

### 4.3.5 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 methods, 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 ( Program 8 (page 120)). 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 Program 9 (page 122).



#### Program 9: Making a sunset as three methods

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

```
int i = 0;

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

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

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

    // increment the index
    i++;
}

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

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

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

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

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

```
}
```

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

What’s different is that the method `makeSunset2()` 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 to pick a file twice—once in each method. Because we wrote these methods 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 methods like `makeSunset()`

There is an issue that the new `makeSunset2()` will take twice as long to finish as the original `makeSunset()`, since every pixel gets changed twice. We address that issue in chapter 15 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 `makeSunset2()` before calling the other methods. The methods `decreaseBlue()` and `decreaseGreen()` are completely flexible and reusable again. But the method `makeSunset2()` 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 `makeSunset2()` 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 that just does something to a picture. You can work on user interfaces later.

Even larger methods, like `makeSunset()`, do “one and only one thing.” The method `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. This is also called *top down refinement or problem decomposition*.

### 4.3.6 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 method is by passing it in as input to the method. Within the method, 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 return it so that it can be used. You can do that by using the `return` keyword.

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)`; it will 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 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 one 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)
    {
        // get the current pixel
        pixel = pixelArray[i];

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

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

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

        // increment i
        i++;
    }
}

```

```
}
```

We could use this method as shown below:

```
> String fName = "C:/intro-prog-java/mediasources/beach-smaller.jpg";  
> Picture picture = new Picture(fName);  
> 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 method.

Recall seeing in Program 7 (page 119) this code:

```
/**  
 * Method to clear the blue from the picture (set  
 * the blue to 0 for all pixels)  
 */  
public void clearBlue()  
{  
    Pixel[] pixelArray = this.getPixels();  
    Pixel pixel = null;  
    int index = 0;  
  
    // loop through all the pixels  
    while (index < pixelArray.length)  
    {  
        // get the current pixel  
        pixel = pixelArray[index];  
  
        // set the blue on the pixel to 0  
        pixel.setBlue(0);  
  
        // 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 method 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.

### 4.3.7 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
- 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.

130 Chapter 4 Modifying Pictures using Loops

- 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 statement 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).



**Common Bug: Change Loop Variable in One Place!**  
When you specify how to change the loop variables in the change area of the `for` loop this will actually happen at the end of the body of the loop. So don't also change the loop variables in the loop or you will change them twice and probably not get the desired results.

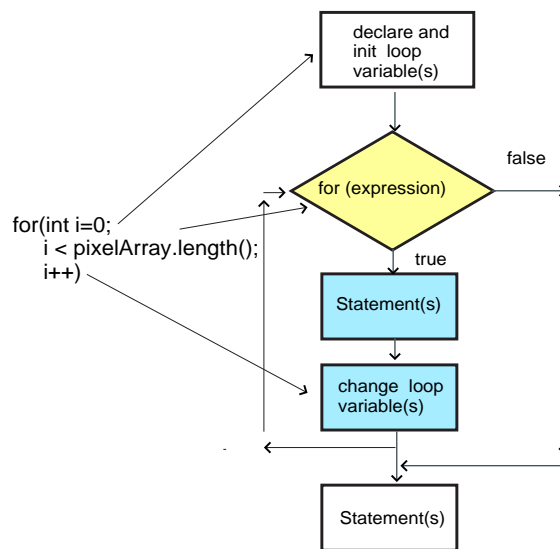


FIGURE 4.22: Flowchart of a for loop

Compare the flowchart (Figure 4.22) for a `for` loop with the flowchart for a `while` loop (Figure 4.16). 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

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

#### 4.3.8 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 4.23 shows the lighter and darker versions of the original picture seen earlier.



### Program 13: Lighten the picture

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

132 Chapter 4 Modifying Pictures using Loops

```

// 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);
    }
}

```

### 4.3.9 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



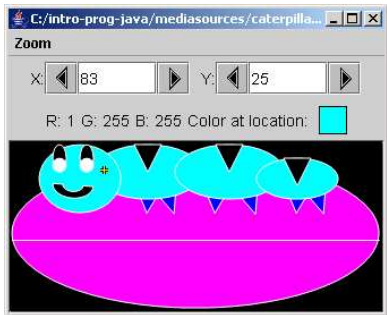


FIGURE 4.24: Negative of the image

### 4.3.10 Converting to Grayscale

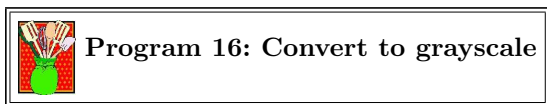
Converting to grayscale is a fun program. It’s short, not hard to understand, and yet has such a nice visual effect. It’s a really nice example of what one can do easily yet powerfully by manipulating pixel color values.

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

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

$$\frac{(red+green+blue)}{3}$$

This leads us to the following simple program and Figure 4.25.



```
/**
 * 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++)
    {
```

Section 4.3 Changing color values 135

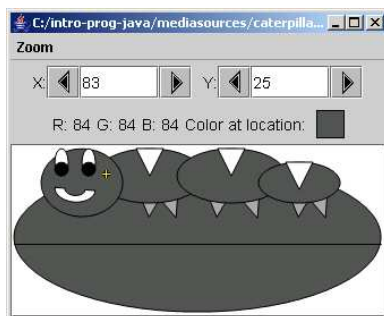


FIGURE 4.25: Color picture converted to grayscale

```


// 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));
}
}
```

## 4.4 CONCEPTS SUMMARY

In this chapter we have introduced arrays, `while` loops, `for` loops, and comments.

### 4.4.1 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.

```
Pixel[] pixelArray;
```

This declares an array of `Pixel` objects. The value stored at each position in the array is a reference to a `Pixel` object.

Arrays are objects and you can find out how large an array by getting its length.

```
i < pixelArray.length
```

Notice that this isn't a method call (there are no parentheses). This accesses a public read-only field.

You can get an element of the array using `arrayReference[index]`. Where the index values can range from 0 to `arrayReference.length-1`.

```
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.

```
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.

```
// 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 `//`.

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

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

```
/* 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 `*/`.

```
/**  
 * 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).

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

### Picture methods

Section 4.4 Concepts Summary 139

getHeight()	This method returns the height of the <b>Picture</b> object in pixels.
getPixel(int x, int y)	This method takes an <i>x</i> position and a <i>y</i> position (two numbers), and returns the <b>Pixel</b> object at that location in the <b>Picture</b> object it is invoked on.
getPixels()	Returns a one-dimensional array of <b>Pixel</b> objects in the <b>Picture</b> object it is invoked on.
getWidth()	This method returns the width of the <b>Picture</b> object in pixels.
writePictureTo(String fileName)	This method takes a file name (a string) as input, then writes the <b>Picture</b> 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.)

**Pixel methods**

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

**Color methods**

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

**ColorChooser methods**

ColorChooser.pickAColor()	Displays a window with ways to pick a color. Find the color you want, and the method will return the Color object that you picked.
---------------------------	--

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. What is meant by each of the following?
- Pixel
  - Kilobyte
  - RGB
  - Loop
  - HSV
  - Flowchart
  - Infinite loop
  - Variable scope
  - Array
  - Matrix
  - JPEG
  - Column-major order
  - Pixelization
  - Luminance
- 4.2. Why don't we see red, green, and blue spots at each position in our picture?
- 4.3. Why is the maximum value of any color channel 255?
- 4.4. 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? Can we represent *enough* colors in RGB?
- 4.5. Program 5 (page 105) is obviously too much color reduction. Write a version that only decreases the red by 10%, and one that reduces red 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.
- 4.6. Each of the below is equivalent to Program 6 (page 117). Test them and convince yourself that they are equivalent. Which do you prefer and why?

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

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

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

- 4.7. 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.
- 4.8. Change a variable name in any of the given methods. Make sure you change all instances of the variable name to the new name. Compile and run the changed method and make sure it still works.
- 4.9. Write new methods like Program 7 (page 119) to clear red and green. For each of these, which would be the most useful in actual practice? How about combinations of these?
- 4.10. Write a method to keep just the blue color. This means to set all the green and red values to zero. Write a method to keep just the red color. Write a method to keep just the green color.

142 Chapter 4 Modifying Pictures using Loops

- 4.11. Write a new method to *maximize* blue (i.e., setting it to 255) instead of clearing it use Program 7 (page 119) as a starting point. Is this useful? Would the red or green versions be useful?
- 4.12. Write a method that modifies the red, green, and blue values of a picture by different amounts. Try it out on different pictures to see if you get any nice results.
- 4.13. How do we get the height from a `Picture` object?
- 4.14. How do we get the width from a `Picture` object?
- 4.15. How many pixels are in a picture with a width of 200 and a height of 100?
- 4.16. How many pixels are in a picture with a width of 640 and a height of 480?
- 4.17. How do you get an array of `Pixel` objects from a `Picture` object?
- 4.18. How do you get the red value from a `Pixel` object?
- 4.19. How do you set the red value in a `Pixel` object?
- 4.20. 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 134) 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 with what your printer produces when you print the image. How do the two pictures differ?

**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 [21].