

How to Think Like a Computer Scientist

Java Version

How to Think Like a Computer Scientist

Java Version

Allen B. Downey

Version 4.1

April 23, 2008

Copyright © 2003, 2008 Allen Downey.

Permission is granted to copy, distribute, and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with Invariant Sections being “Preface”, with no Front-Cover Texts, and with no Back-Cover Texts. A copy of the license is included in the appendix entitled “GNU Free Documentation License.”

The GNU Free Documentation License is available from www.gnu.org or by writing to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307, USA.

The original form of this book is \LaTeX source code. Compiling this \LaTeX source has the effect of generating a device-independent representation of the book, which can be converted to other formats and printed.

The \LaTeX source for this book is available from

`thinkapjava.com`

This book was typeset using \LaTeX . The illustrations were drawn in xfig. All of these are free, open-source programs.

Preface

“As we enjoy great Advantages from the Inventions of others, we should be glad of an Opportunity to serve others by any Invention of ours, and this we should do freely and generously.”

—Benjamin Franklin, quoted in *Benjamin Franklin* by Edmund S. Morgan.

Why I wrote this book

This is the fourth edition of a book I started writing in 1999, when I was teaching at Colby College. I had taught an introductory computer science class using the Java programming language, but I had not found a textbook I was happy with. For one thing, they were all too big! There was no way my students would read 800 pages of dense, technical material, even if I wanted them to. And I didn’t want them to. Most of the material was too specific—details about Java and its libraries that would be obsolete by the end of the semester, and that obscured the material I really wanted to get to.

The other problem I found was that the introduction to object oriented programming was too abrupt. Many students who were otherwise doing well just hit a wall when we got to objects, whether we did it at the beginning, middle or end.

So I started writing. I wrote a chapter a day for 13 days, and on the 14th day I edited. Then I sent it to be photocopied and bound. When I handed it out on the first day of class, I told the students that they would be expected to read one chapter a week. In other words, they would read it seven times slower than I wrote it.

The philosophy behind it

Here are some of the ideas that made the book the way it is:

- Vocabulary is important. Students need to be able to talk about programs and understand what I am saying. I tried to introduce the minimum number of terms, to define them carefully when they are first used, and

to organize them in glossaries at the end of each chapter. In my class, I include vocabulary questions on quizzes and exams, and require students to use appropriate terms in short-answer responses.

- In order to write a program, students have to understand the algorithm, know the programming language, and they have to be able to debug. I think too many books neglect debugging. This book includes an appendix on debugging and an appendix on program development (which can help avoid debugging). I recommend that students read this material early and come back to it often.
- Some concepts take time to sink in. Some of the more difficult ideas in the book, like recursion, appear several times. By coming back to these ideas, I am trying to give students a chance to review and reinforce or, if they missed it the first time, a chance to catch up.
- I try to use the minimum amount of Java to get the maximum amount of programming power. The purpose of this book is to teach programming and some introductory ideas from computer science, not Java. I left out some language features, like the `switch` statement, that are unnecessary, and avoided most of the libraries, especially the ones like the AWT that have been changing quickly or are likely to be replaced.

The minimalism of my approach has some advantages. Each chapter is about ten pages, not including the exercises. In my classes I ask students to read each chapter before we discuss it, and I have found that they are willing to do that and their comprehension is good. Their preparation makes class time available for discussion of the more abstract material, in-class exercises, and additional topics that aren't in the book.

But minimalism has some disadvantages. There is not much here that is intrinsically fun. Most of my examples demonstrate the most basic use of a language feature, and many of the exercises involve string manipulation and mathematical ideas. I think some of them are fun, but many of the things that excite students about computer science, like graphics, sound and network applications, are given short shrift.

The problem is that many of the more exciting features involve lots of details and not much concept. Pedagogically, that means a lot of effort for not much payoff. So there is a tradeoff between the material that students enjoy and the material that is most intellectually rich. I leave it to individual teachers to find the balance that is best for their classes. To help, the book includes appendices that cover graphics, keyboard input and file input.

Object-oriented programming

Some books introduce objects immediately; others warm up with a more procedural style and develop object-oriented style more gradually. This book is probably the extreme of the “objects late” approach.

Many of Java's object-oriented features are motivated by problems with previous languages, and their implementations are influenced by this history. Some of these features are hard to explain if students aren't familiar with the problems they solve.

It wasn't my intention to postpone object-oriented programming. On the contrary, I got to it as quickly as I could, limited by my intention to introduce concepts one at a time, as clearly as possible, in a way that allows students to practice each idea in isolation before adding the next. It just happens that it takes 13 steps.

Data structures

In Fall 2000 I taught the second course in the introductory sequence, called Data Structures, and wrote additional chapters covering lists, stacks, queues, trees, and hashtables.

Each chapter presents the interface for a data structure, one or more algorithms that use it, and at least one implementation. In most cases there is also an implementation in the `java.utils` package, so teachers can decide on a case-by-case basis whether to discuss the implementation, and whether students will build an implementation as an exercise. For the most part I present data structures and interfaces that are consistent with the implementation in `java.utils`.

The Computer Science AP Exam

During Summer 2001 I worked with teachers at the Maine School of Science and Mathematics on a version of the book that would help students prepare for the Computer Science Advanced Placement Exam, which used C++ at the time. The translation went quickly because, as it turned out, the material I covered was almost identical to the AP Syllabus.

Naturally, when the College Board announced that the AP Exam would switch to Java, I made plans to update the Java version of the book. Looking at the proposed AP Syllabus, I saw that their subset of Java was all but identical to the subset I had chosen.

During January 2003, I worked on the Fourth Edition of the book, making these changes:

- I added a new chapter covering Huffman codes.
- I revised several sections that I had found problematic, including the transition to object-oriented programming and the discussion of heaps.
- I improved the appendices on debugging and program development.
- I added a few sections to improve coverage of the AP syllabus.

- I collected the exercises, quizzes, and exam questions I had used in my classes and put them at the end of the appropriate chapters. I also made up some problems that are intended to help with AP Exam preparation.

Free books!

Since the beginning, this book and its descendents have been available under the GNU Free Documentation License. Readers are free to download the book in a variety of formats and print it or read it on screen. Teachers are free to send the book to a short-run printer and make as many copies as they need. And, maybe most importantly, anyone is free to customize the book for their needs. You can download the L^AT_EX source code, and then add, remove, edit, or rearrange material, and make the book that is best for you or your class.

People have translated the book into other computer languages (including Python and Eiffel), and other natural languages (including Spanish, French and German). Many of these derivatives are also available under the GNU FDL.

This approach to publishing has a lot of advantages, but there is one drawback: my books have never been through a formal editing and proofreading process and, too often, it shows. Motivated by Open Source Software, I have adopted the philosophy of releasing the book early and updating it often. I do my best to minimize the number of errors, but I also depend on readers to help out.

The response has been great. I get messages almost every day from people who have read the book and liked it enough to take the trouble to send in a “bug report.” Often I can correct an error and post an updated version almost immediately. I think of the book as a work in progress, improving a little whenever I have time to make a revision, or when readers take the time to send feedback.

Oh, the title

I get a lot of grief about the title of the book. Not everyone understands that it is—mostly—a joke. Reading this book will probably not make you think like a computer scientist. That takes time, experience, and probably a few more classes.

But there is a kernel of truth in the title: this book is not about Java, and it is only partly about programming. If it is successful, this book is about a way of thinking. Computer scientists have an approach to problem-solving, and a way of crafting solutions, that is unique, versatile and powerful. I hope that this book gives you a sense of what that approach is, and that at some point you will find yourself thinking like a computer scientist.

Allen Downey
Needham, Massachusetts
March 6, 2003

Contributors List

When I started writing free books, it didn't occur to me to keep a contributors list. When Jeff Elkner suggested it, it seemed so obvious that I am embarrassed by the omission. This list starts with the 4th Edition, so it omits many people who contributed suggestions and corrections to earlier versions.

- Tania Passfield pointed out that the glossary of Chapter 4 has some left-over terms that no longer appear in the text.
- Elizabeth Wiethoff noticed that my series expansion of e^{-x^2} was wrong. She is also working on a Ruby version of the book!
- Matt Crawford sent in a whole patch file full of corrections!
- Chi-Yu Li pointed out a typo and an error in one of the code examples.

Contents

Preface	v
1 The way of the program	1
1.1 What is a programming language?	1
1.2 What is a program?	3
1.3 What is debugging?	4
1.4 Formal and natural languages	5
1.5 The first program	7
1.6 Glossary	8
1.7 Exercises	10
2 Variables and types	13
2.1 More printing	13
2.2 Variables	14
2.3 Assignment	15
2.4 Printing variables	16
2.5 Keywords	17
2.6 Operators	17
2.7 Order of operations	18
2.8 Operators for Strings	19
2.9 Composition	19
2.10 Glossary	20
2.11 Exercises	20

3	Methods	23
3.1	Floating-point	23
3.2	Converting from <code>double</code> to <code>int</code>	24
3.3	Math methods	25
3.4	Composition	26
3.5	Adding new methods	26
3.6	Classes and methods	28
3.7	Programs with multiple methods	29
3.8	Parameters and arguments	29
3.9	Stack diagrams	31
3.10	Methods with multiple parameters	31
3.11	Methods with results	32
3.12	Glossary	32
3.13	Exercises	33
4	Conditionals and recursion	35
4.1	The modulus operator	35
4.2	Conditional execution	35
4.3	Alternative execution	36
4.4	Chained conditionals	37
4.5	Nested conditionals	37
4.6	The return statement	38
4.7	Type conversion	38
4.8	Recursion	39
4.9	Stack diagrams for recursive methods	40
4.10	Convention and divine law	41
4.11	Glossary	42
4.12	Exercises	43

5	Fruitful methods	47
5.1	Return values	47
5.2	Program development	49
5.3	Composition	51
5.4	Overloading	51
5.5	Boolean expressions	52
5.6	Logical operators	53
5.7	Boolean methods	54
5.8	More recursion	54
5.9	Leap of faith	57
5.10	One more example	57
5.11	Glossary	58
5.12	Exercises	59
6	Iteration	65
6.1	Multiple assignment	65
6.2	Iteration	66
6.3	The while statement	66
6.4	Tables	68
6.5	Two-dimensional tables	69
6.6	Encapsulation and generalization	70
6.7	Methods	71
6.8	More encapsulation	71
6.9	Local variables	72
6.10	More generalization	72
6.11	Glossary	74
6.12	Exercises	75

7	Strings and things	79
7.1	Invoking methods on objects	79
7.2	Length	80
7.3	Traversal	80
7.4	Run-time errors	81
7.5	Reading documentation	81
7.6	The <code>indexOf</code> method	82
7.7	Looping and counting	83
7.8	Increment and decrement operators	83
7.9	Strings are immutable	84
7.10	Strings are incomparable	84
7.11	Glossary	85
7.12	Exercises	86
8	Interesting objects	91
8.1	What's interesting?	91
8.2	Packages	91
8.3	<code>Point</code> objects	92
8.4	Instance variables	92
8.5	Objects as parameters	93
8.6	Rectangles	94
8.7	Objects as return types	94
8.8	Objects are mutable	94
8.9	Aliasing	95
8.10	<code>null</code>	96
8.11	Garbage collection	97
8.12	Objects and primitives	97
8.13	Glossary	98
8.14	Exercises	98

9	Create your own objects	103
9.1	Class definitions and object types	103
9.2	Time	104
9.3	Constructors	105
9.4	More constructors	105
9.5	Creating a new object	106
9.6	Printing an object	107
9.7	Operations on objects	108
9.8	Pure functions	108
9.9	Modifiers	110
9.10	Fill-in methods	111
9.11	Which is best?	111
9.12	Incremental development vs. planning	111
9.13	Generalization	113
9.14	Algorithms	113
9.15	Glossary	114
9.16	Exercises	114
10	Arrays	119
10.1	Accessing elements	119
10.2	Copying arrays	120
10.3	for loops	121
10.4	Arrays and objects	122
10.5	Array length	122
10.6	Random numbers	123
10.7	Array of random numbers	123
10.8	Counting	124
10.9	The histogram	125
10.10	A single-pass solution	126
10.11	Glossary	126
10.12	Exercises	127

11 Arrays of Objects	131
11.1 Composition	131
11.2 Card objects	131
11.3 The <code>printCard</code> method	133
11.4 The <code>sameCard</code> method	134
11.5 The <code>compareCard</code> method	135
11.6 Arrays of cards	136
11.7 The <code>printDeck</code> method	137
11.8 Searching	137
11.9 Decks and subdecks	141
11.10 Glossary	141
11.11 Exercises	142
12 Objects of Arrays	143
12.1 The <code>Deck</code> class	143
12.2 Shuffling	144
12.3 Sorting	145
12.4 Subdecks	146
12.5 Shuffling and dealing	147
12.6 Mergesort	147
12.7 Glossary	149
12.8 Exercises	149
13 Object-oriented programming	153
13.1 Programming languages and styles	153
13.2 Object and class methods	154
13.3 The current object	154
13.4 Complex numbers	154
13.5 A function on <code>Complex</code> numbers	155
13.6 Another function on <code>Complex</code> numbers	156
13.7 A modifier	156

13.8	The <code>toString</code> method	157
13.9	The <code>equals</code> method	157
13.10	Invoking one object method from another	158
13.11	Oddities and errors	159
13.12	Inheritance	159
13.13	Drawable rectangles	160
13.14	The class hierarchy	161
13.15	Object-oriented design	161
13.16	Glossary	161
13.17	Exercises	162
14	Linked lists	163
14.1	References in objects	163
14.2	The <code>Node</code> class	163
14.3	Lists as collections	165
14.4	Lists and recursion	166
14.5	Infinite lists	166
14.6	The fundamental ambiguity theorem	167
14.7	Object methods for nodes	168
14.8	Modifying lists	168
14.9	Wrappers and helpers	169
14.10	The <code>IntList</code> class	170
14.11	Invariants	171
14.12	Glossary	171
14.13	Exercises	172
15	Stacks	175
15.1	Abstract data types	175
15.2	The Stack ADT	176
15.3	The Java Stack Object	176
15.4	Wrapper classes	177

15.5	Creating wrapper objects	178
15.6	Creating more wrapper objects	178
15.7	Getting the values out	178
15.8	Useful methods in the wrapper classes	179
15.9	Postfix expressions	179
15.10	Parsing	180
15.11	Implementing ADTs	181
15.12	Array implementation of the Stack ADT	181
15.13	Resizing arrays	182
15.14	Glossary	184
15.15	Exercises	185
16	Queues and Priority Queues	187
16.1	The queue ADT	187
16.2	Veneer	189
16.3	Linked Queue	190
16.4	Circular buffer	192
16.5	Priority queue	195
16.6	Metaclass	195
16.7	Array implementation of Priority Queue	196
16.8	A Priority Queue client	197
16.9	The <code>Golfer</code> class	198
16.10	Glossary	200
16.11	Exercises	201
17	Trees	203
17.1	A tree node	203
17.2	Building trees	204
17.3	Traversing trees	204
17.4	Expression trees	205
17.5	Traversal	206

Contents	xix
17.6 Encapsulation	207
17.7 Defining a metaclass	207
17.8 Implementing a metaclass	208
17.9 The <code>Vector</code> class	209
17.10 The <code>Iterator</code> class	210
17.11 Glossary	211
17.12 Exercises	212
18 Heap	215
18.1 Array implementation of a tree	215
18.2 Performance analysis	218
18.3 Analysis of mergesort	220
18.4 Overhead	221
18.5 Priority Queue implementations	222
18.6 Definition of a Heap	223
18.7 Heap <code>remove</code>	224
18.8 Heap <code>add</code>	225
18.9 Performance of heaps	226
18.10 Heapsort	227
18.11 Glossary	227
18.12 Exercises	228
19 Maps	229
19.1 Arrays, Vectors and Maps	229
19.2 The Map ADT	230
19.3 The built-in <code>HashMap</code>	230
19.4 A <code>Vector</code> implementation	232
19.5 The <code>List</code> metaclass	234
19.6 <code>HashMap</code> implementation	234
19.7 Hash Functions	235
19.8 Resizing a hash map	236
19.9 Performance of resizing	237
19.10 Glossary	237
19.11 Exercises	238

20 Huffman code	241
20.1 Variable-length codes	241
20.2 The frequency table	242
20.3 The Huffman Tree	243
20.4 The super method	245
20.5 Decoding	246
20.6 Encoding	247
20.7 Glossary	248
 A Program development plan	 249
 B Debugging	 255
B.1 Compile-time errors	255
B.2 Run-time errors	258
B.3 Semantic errors	261
 C Input and Output in Java	 267
 D Graphics	 269
D.1 Slates and Graphics objects	269
D.2 Invoking methods on a Graphics object	269
D.3 Coordinates	270
D.4 A lame Mickey Mouse	271
D.5 Other drawing commands	272
D.6 The Slate Class	273

Chapter 1

The way of the program

The goal of this book, and this class, is to teach you to think like a computer scientist. I like the way computer scientists think because they combine some of the best features of Mathematics, Engineering, and Natural Science. Like mathematicians, computer scientists use formal languages to denote ideas (specifically computations). Like engineers, they design things, assembling components into systems and evaluating tradeoffs among alternatives. Like scientists, they observe the behavior of complex systems, form hypotheses, and test predictions.

The single most important skill for a computer scientist is **problem-solving**. By that I mean the ability to formulate problems, think creatively about solutions, and express a solution clearly and accurately. As it turns out, the process of learning to program is an excellent opportunity to practice problem-solving skills. That's why this chapter is called "The way of the program."

On one level, you will be learning to program, which is a useful skill by itself. On another level you will use programming as a means to an end. As we go along, that end will become clearer.

1.1 What is a programming language?

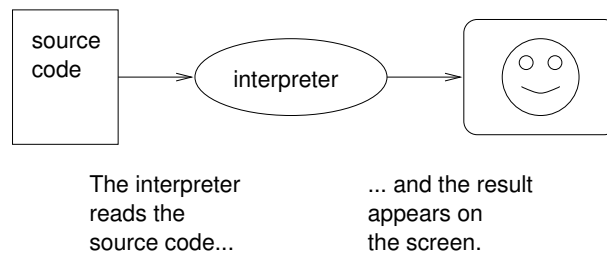
The programming language you will be learning is Java, which is relatively new (Sun released the first version in May, 1995). Java is an example of a **high-level language**; other high-level languages you might have heard of are Pascal, C, C++ and FORTRAN.

As you might infer from the name "high-level language," there are also **low-level languages**, sometimes referred to as machine language or assembly language. Loosely-speaking, computers can only execute programs written in low-level languages. Thus, programs written in a high-level language have to be translated before they can run. This translation takes some time, which is a small disadvantage of high-level languages.

But the advantages are enormous. First, it is *much* easier to program in a high-level language; by “easier” I mean that the program takes less time to write, it’s shorter and easier to read, and it’s more likely to be correct. Secondly, high-level languages are **portable**, meaning that they can run on different kinds of computers with few or no modifications. Low-level programs can only run on one kind of computer, and have to be rewritten to run on another.

Due to these advantages, almost all programs are written in high-level languages. Low-level languages are only used for a few special applications.

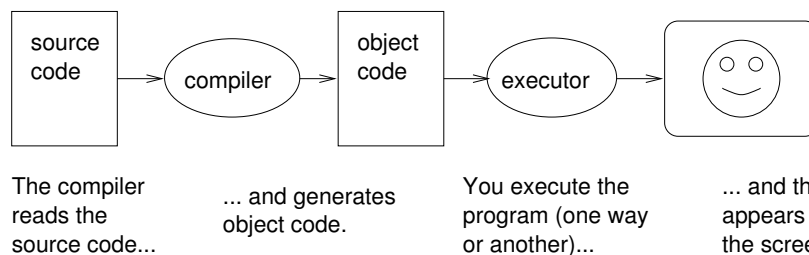
There are two ways to translate a program; **interpreting** or **compiling**. An interpreter is a program that reads a high-level program and does what it says. In effect, it translates the program line-by-line, alternately reading lines and carrying out commands.



A compiler is a program that reads a high-level program and translates it all at once, before executing any of the commands. Often you compile the program as a separate step, and then execute the compiled code later. In this case, the high-level program is called the **source code**, and the translated program is called the **object code** or the **executable**.

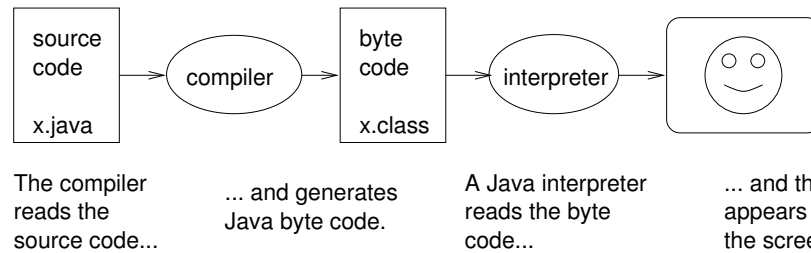
As an example, suppose you write a program in C. You might use a text editor to write the program (a text editor is a simple word processor). When the program is finished, you might save it in a file named **program.c**, where “program” is an arbitrary name you make up, and the suffix **.c** is a convention that indicates that the file contains C source code.

Then, depending on what your programming environment is like, you might leave the text editor and run the compiler. The compiler would read your source code, translate it, and create a new file named **program.o** to contain the object code, or **program.exe** to contain the executable.



The Java language is unusual because it is both compiled and interpreted. Instead of translating Java programs into machine language, the Java compiler

generates Java byte code. Byte code is easy (and fast) to interpret, like machine language, but it is also portable, like a high-level language. Thus, it is possible to compile a Java program on one machine, transfer the byte code to another machine over a network, and then interpret the byte code on the other machine. This ability is one of the advantages of Java over many other high-level languages.



Although this process may seem complicated, in most programming environments (sometimes called development environments), these steps are automated for you. Usually you will only have to write a program and press a button or type a single command to compile and run it. On the other hand, it is useful to know what the steps are that are happening in the background, so that if something goes wrong you can figure out what it is.

1.2 What is a program?

A program is a sequence of instructions that specifies how to perform a computation. The computation might be something mathematical, like solving a system of equations or finding the roots of a polynomial, but it can also be a symbolic computation, like searching and replacing text in a document or (strangely enough) compiling a program.

The instructions, which we will call **statements**, look different in different programming languages, but there are a few basic operations most languages can perform:

input: Get data from the keyboard, or a file, or some other device.

output: Display data on the screen or send data to a file or other device.

math: Perform basic mathematical operations like addition and multiplication.

testing: Check for certain conditions and execute the appropriate sequence of statements.

repetition: Perform some action repeatedly, usually with some variation.

That's pretty much all there is to it. Every program you've ever used, no matter how complicated, is made up of statements that perform these operations. Thus, one way to describe programming is the process of breaking a large, complex task up into smaller and smaller subtasks until eventually the subtasks are simple enough to be performed with one of these basic operations.

1.3 What is debugging?

Programming is a complex process, and since it is done by human beings, it often leads to errors. For whimsical reasons, programming errors are called **bugs** and the process of tracking them down and correcting them is called **debugging**.

There are a few different kinds of errors that can occur in a program, and it is useful to distinguish between them in order to track them down more quickly.

1.3.1 Compile-time errors

The compiler can only translate a program if the program is syntactically correct; otherwise, the compilation fails and you will not be able to run your program. **Syntax** refers to the structure of your program and the rules about that structure.

For example, in English, a sentence must begin with a capital letter and end with a period. this sentence contains a syntax error. So does this one

For most readers, a few syntax errors are not a significant problem, which is why we can read the poetry of e e cummings without spewing error messages.

Compilers are not so forgiving. If there is a single syntax error anywhere in your program, the compiler will print an error message and quit, and you will not be able to run your program.

To make matters worse, there are more syntax rules in Java than there are in English, and the error messages you get from the compiler are often not very helpful. During the first few weeks of your programming career, you will probably spend a lot of time tracking down syntax errors. As you gain experience, though, you will make fewer errors and find them faster.

1.3.2 Run-time errors

The second type of error is a run-time error, so-called because the error does not appear until you run the program. In Java, run-time errors occur when the interpreter is running the byte code and something goes wrong.

The good news for now is that Java tends to be a **safe** language, which means that run-time errors are rare, especially for the simple sorts of programs we will be writing for the next few weeks.

Later on in the semester, you will probably start to see more run-time errors, especially when we start talking about objects and references (Chapter 8).

In Java, run-time errors are called **exceptions**, and in most environments they appear as windows or dialog boxes that contain information about what happened and what the program was doing when it happened. This information is useful for debugging.

1.3.3 Logic errors and semantics

The third type of error is the **logical** or **semantic** error. If there is a logical error in your program, it will compile and run successfully, in the sense that the computer will not generate any error messages, but it will not do the right thing. It will do something else. Specifically, it will do what you told it to do.

The problem is that the program you wrote is not the program you wanted to write. The meaning of the program (its semantics) is wrong. Identifying logical errors can be tricky, since it requires you to work backwards by looking at the output of the program and trying to figure out what it is doing.

1.3.4 Experimental debugging

One of the most important skills you will acquire in this class is debugging. Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming.

In some ways debugging is like detective work. You are confronted with clues and you have to infer the processes and events that lead to the results you see.

Debugging is also like an experimental science. Once you have an idea what is going wrong, you modify your program and try again. If your hypothesis was correct, then you can predict the result of the modification, and you take a step closer to a working program. If your hypothesis was wrong, you have to come up with a new one. As Sherlock Holmes pointed out, “When you have eliminated the impossible, whatever remains, however improbable, must be the truth.” (from A. Conan Doyle’s *The Sign of Four*).

For some people, programming and debugging are the same thing. That is, programming is the process of gradually debugging a program until it does what you want. The idea is that you should always start with a working program that does *something*, and make small modifications, debugging them as you go, so that you always have a working program.

For example, Linux is an operating system that contains thousands of lines of code, but it started out as a simple program Linus Torvalds used to explore the Intel 80386 chip. According to Larry Greenfield, “One of Linus’s earlier projects was a program that would switch between printing AAAA and BBBB. This later evolved to Linux” (from *The Linux Users’ Guide Beta Version 1*).

In later chapters I will make more suggestions about debugging and other programming practices.

1.4 Formal and natural languages

Natural languages are the languages that people speak, like English, Spanish, and French. They were not designed by people (although people try to impose some order on them); they evolved naturally.

Formal languages are languages that are designed by people for specific applications. For example, the notation that mathematicians use is a formal language that is particularly good at denoting relationships among numbers and symbols. Chemists use a formal language to represent the chemical structure of molecules. And most importantly:

Programming languages are formal languages that have been designed to express computations.

As I mentioned before, formal languages tend to have strict rules about syntax. For example, $3 + 3 = 6$ is a syntactically correct mathematical statement, but $3 = +6\$$ is not. Also, H_2O is a syntactically correct chemical name, but $_2Zz$ is not.

Syntax rules come in two flavors, pertaining to tokens and structure. Tokens are the basic elements of the language, like words and numbers and chemical elements. One of the problems with $3=+6\$$ is that $\$$ is not a legal token in mathematics (at least as far as I know). Similarly, $_2Zz$ is not legal because there is no element with the abbreviation Zz .

The second type of syntax rule pertains to the structure of a statement; that is, the way the tokens are arranged. The statement $3=+6\$$ is structurally illegal, because you can't have a plus sign immediately after an equals sign. Similarly, molecular formulas have to have subscripts after the element name, not before.

When you read a sentence in English or a statement in a formal language, you have to figure out what the structure of the sentence is (although in a natural language you do this unconsciously). This process is called **parsing**.

For example, when you hear the sentence, "The other shoe fell," you understand that "the other shoe" is the subject and "fell" is the verb. Once you have parsed a sentence, you can figure out what it means, that is, the semantics of the sentence. Assuming that you know what a shoe is, and what it means to fall, you will understand the general implication of this sentence.

Although formal and natural languages have many features in common—tokens, structure, syntax and semantics—there are many differences.

ambiguity: Natural languages are full of ambiguity, which people deal with by using contextual clues and other information. Formal languages are designed to be nearly or completely unambiguous, which means that any statement has exactly one meaning, regardless of context.

redundancy: In order to make up for ambiguity and reduce misunderstandings, natural languages employ lots of redundancy. As a result, they are often verbose. Formal languages are less redundant and more concise.

literalness: Natural languages are full of idiom and metaphor. If I say, "The other shoe fell," there is probably no shoe and nothing falling. Formal languages mean exactly what they say.

People who grow up speaking a natural language (everyone) often have a hard time adjusting to formal languages. In some ways the difference between formal and natural language is like the difference between poetry and prose, but more so:

Poetry: Words are used for their sounds as well as for their meaning, and the whole poem together creates an effect or emotional response. Ambiguity is not only common but often deliberate.

Prose: The literal meaning of words is more important and the structure contributes more meaning. Prose is more amenable to analysis than poetry, but still often ambiguous.

Programs: The meaning of a computer program is unambiguous and literal, and can be understood entirely by analysis of the tokens and structure.

Here are some suggestions for reading programs (and other formal languages). First, remember that formal languages are much more dense than natural languages, so it takes longer to read them. Also, the structure is very important, so it is usually not a good idea to read from top to bottom, left to right. Instead, learn to parse the program in your head, identifying the tokens and interpreting the structure. Finally, remember that the details matter. Little things like spelling errors and bad punctuation, which you can get away with in natural languages, can make a big difference in a formal language.

1.5 The first program

Traditionally the first program people write in a new language is called “Hello, World.” because all it does is display the words “Hello, World.” In Java, this program looks like this:

```
class Hello {  
  
    // main: generate some simple output  
  
    public static void main (String[] args) {  
        System.out.println ("Hello, world.");  
    }  
}
```

Some people judge the quality of a programming language by the simplicity of the “Hello, World.” program. By this standard, Java does not do very well. Even the simplest program contains a number of features that are hard to explain to beginning programmers. We are going to ignore a lot of them for now, but I will explain a few.

All programs are made up of **class definitions**, which have the form:

```
class CLASSNAME {  
  
    public static void main (String[] args) {  
        STATEMENTS  
    }  
}
```

Here `CLASSNAME` indicates an arbitrary name that you make up. The class name in the example is `Hello`.

In the second line, you should ignore the words `public static void` for now, but notice the word `main`. `main` is a special name that indicates the place in the program where execution begins. When the program runs, it starts by executing the first statement in `main` and it continues, in order, until it gets to the last statement, and then it quits.

There is no limit to the number of statements that can be in `main`, but the example contains only one. It is a **print statement**, meaning that it prints a message on the screen. It is a bit confusing that “print” sometimes means “display something on the screen,” and sometimes means “send something to the printer.” In this book I won’t say much about sending things to the printer; we’ll do all our printing on the screen.

The statement that prints things on the screen is `System.out.println`, and the thing between the parentheses is the thing that will get printed. At the end of the statement there is a semi-colon (`;`), which is required at the end of every statement.

There are a few other things you should notice about the syntax of this program. First, Java uses squiggly-braces (`{` and `}`) to group things together. The outermost squiggly-braces (lines 1 and 8) contain the class definition, and the inner braces contain the definition of `main`.

Also, notice that line 3 begins with `//`. This indicates that this line contains a **comment**, which is a bit of English text that you can put in the middle of a program, usually to explain what the program does. When the compiler sees a `//`, it ignores everything from there until the end of the line.

1.6 Glossary

problem-solving: The process of formulating a problem, finding a solution, and expressing the solution.

high-level language: A programming language like Java that is designed to be easy for humans to read and write.

low-level language: A programming language that is designed to be easy for a computer to execute. Also called “machine language” or “assembly language.”

formal language: Any of the languages people have designed for specific purposes, like representing mathematical ideas or computer programs. All programming languages are formal languages.

natural language: Any of the languages people speak that have evolved naturally.

portability: A property of a program that can run on more than one kind of computer.

interpret: To execute a program in a high-level language by translating it one line at a time.

compile: To translate a program in a high-level language into a low-level language, all at once, in preparation for later execution.

source code: A program in a high-level language, before being compiled.

object code: The output of the compiler, after translating the program.

executable: Another name for object code that is ready to be executed.

byte code: A special kind of object code used for Java programs. Byte code is similar to a low-level language, but it is portable, like a high-level language.

statement: A part of a program that specifies an action that will be performed when the program runs. A print statement causes output to be displayed on the screen.

comment: A part of a program that contains information about the program, but that has no effect when the program runs.

algorithm: A general process for solving a category of problems.

bug: An error in a program.

syntax: The structure of a program.

semantics: The meaning of a program.

parse: To examine a program and analyze the syntactic structure.

syntax error: An error in a program that makes it impossible to parse (and therefore impossible to compile).

exception: An error in a program that makes it fail at run-time. Also called a run-time error.

logical error: An error in a program that makes it do something other than what the programmer intended.

debugging: The process of finding and removing any of the three kinds of errors.

1.7 Exercises

Exercise 1.1

Computer scientists have the annoying habit of using common English words to mean something different from their common English meaning. For example, in English, a statement and a comment are pretty much the same thing, but when we are talking about a program, they are very different.

The glossary at the end of each chapter is intended to highlight words and phrases that have special meanings in computer science. When you see familiar words, don't assume that you know what they mean!

- a. In computer jargon, what's the difference between a statement and a comment?
- b. What does it mean to say that a program is portable?
- c. What is an executable?

Exercise 1.2

Before you do anything else, find out how to compile and run a Java program in your environment. Some environments provide sample programs similar to the example in Section 1.5.

- a. Type in the "Hello, world" program, then compile and run it.
- b. Add a second print statement that prints a second message after the "Hello, world!". Something witty like, "How are you?" Compile and run the program again.
- c. Add a comment line to the program (anywhere) and recompile it. Run the program again. The new comment should not affect the execution of the program.

This exercise may seem trivial, but it is the starting place for many of the programs we will work with. In order to debug with confidence, you have to have confidence in your programming environment. In some environments, it is easy to lose track of which program is executing, and you might find yourself trying to debug one program while you are accidentally executing another. Adding (and changing) print statements is a simple way to establish the connection between the program you are looking at and the output when the program runs.

Exercise 1.3

It is a good idea to commit as many errors as you can think of, so that you see what error messages the compiler produces. Sometimes the compiler will tell you exactly what is wrong, and all you have to do is fix it. Sometimes, though, the compiler will produce wildly misleading messages. You will develop a sense for when you can trust the compiler and when you have to figure things out yourself.

- a. Remove one of the open squiggly-braces.
- b. Remove one of the close squiggly-braces.
- c. Instead of `main`, write `mian`.