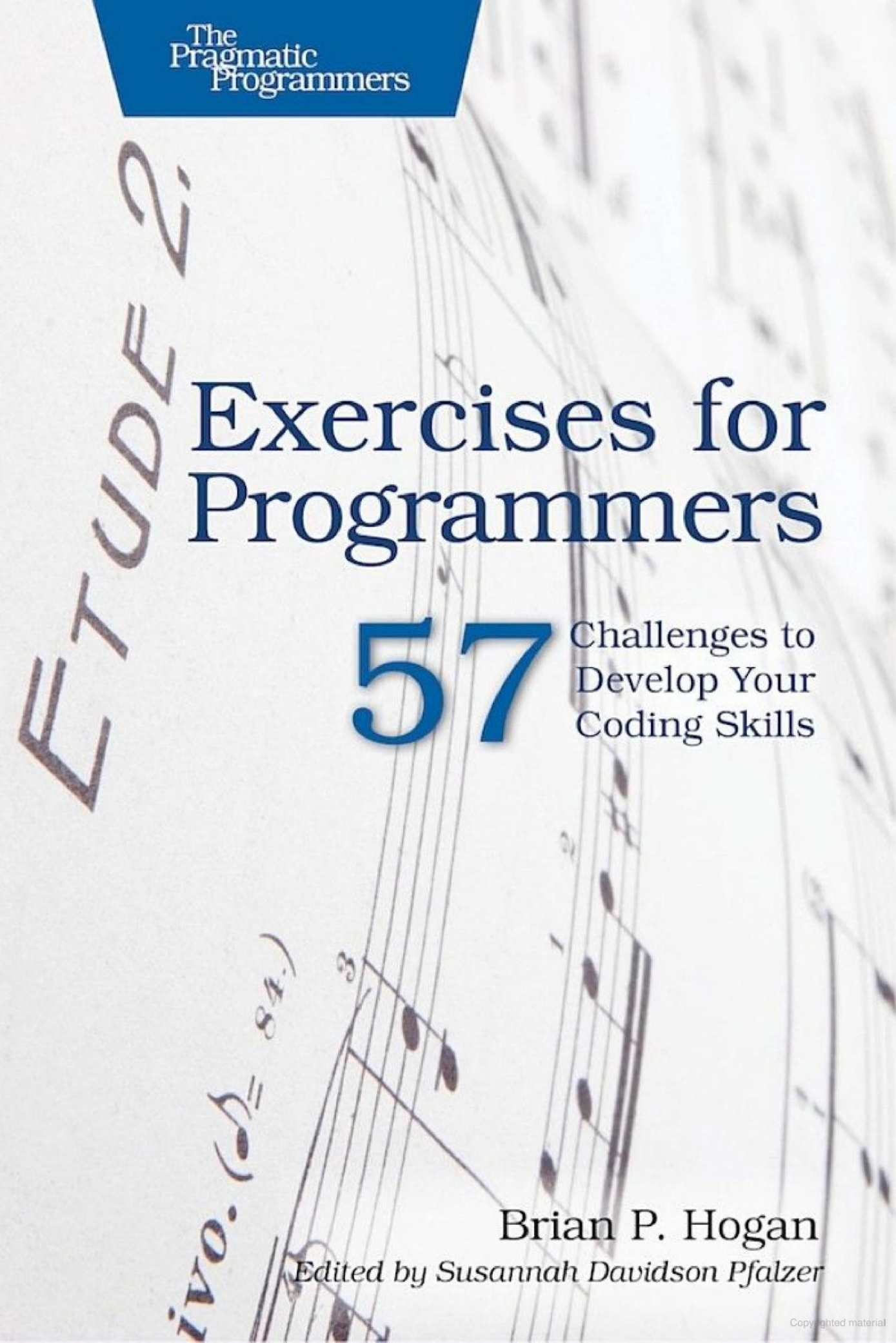

Table of Contents

Early praise for *Exercises for Programmers*

If you're looking to pick up a new programming language, you should also pick up this book. You'll learn how to solve problems from first principles, developing a stronger foundation to build on top of. I learned a lot. I expect you will too.

→ Stephen Orr

Senior software engineer, Impact Applications

A wonderful resource for learning new languages using the most effective method: practice. Because the book is language agnostic, it has almost endless replay value, which is a rare quality among technical books.

→ Jason Pike

Software developer, theswiftlearner.com

This is a wonderful book for anyone who wants to start fresh in a new language. Programmers new and old will greatly benefit from this repository of exercises. This book offers comfort for beginners and challenges for advanced programmers.

→ Alex Henry

Software engineer quality assurance, JAMF Software

Acknowledgments

First, thank you. You're awesome. No, you really are, because you've picked up this book and made a commitment to improving your skills as a software developer. I wrote this book for people just like you, so thank you for reading.

Second, thank you, Dave Thomas, for believing in this idea and for your guidance over the years. It's been an honor and a privilege to learn from you. Your encouragement on this book means a lot, and I appreciate your generosity with your time as you reviewed the exercises and offered suggestions. You and Andy continue to make the world better for programmers, and I'm grateful to be able to contribute to that in my small way.

A special thank you to Susannah Pfalzer. You always make my books better than they started out. You seem to catch all the right details, and you guide me to focus on what really matters. This is the sixth book you've helped me with, and I'm a better writer because of all your guidance over the years.

Next, thank you, Andy Hunt, Mike Reilly, Michael Swaine, Fahmida Rashid, and Bruce Tate, for your encouragement when I proposed this idea.

The programs in this book are ones I've been using to teach programming over the last ten years. Thank you to Zachary Baxter, Jordan Berg, Luke Chase, Dee Dee Dale, Jacob Donahoe, Alex Eckblad, Arrio Farugie, Emily Mikl, Aaron Miller, Eric Mohr, Zachary Solofra, Darren Sopiarz, Ashley Stevens, Miah Thalacker, Andrew Walley, and all the other students who've come through my classes and training sessions over the years. The feedback you've provided on my approach to teaching has helped me immensely. And thank you, Kyle Loewenhagen, Jon Cooley, and George Andrews, for helping me grow as a teacher with your feedback and insights.

Thank you, Deb Walsh, for your encouragement and incredible ideas on how to get the best out of students. We share core beliefs about teaching and learning, and I learn so much from our conversations. Thank you for sharing your experience and expertise with me and for your support of my teaching methods.

This book of exercises flows much better and is clarified by the fantastic

feedback from a great mix of new and veteran software developers. Each reviewer put an incredible amount of time and effort into working through these problems in their favorite programming language, helping me identify things that didn't make sense or needed improvement. Thank you, Chris C., Alex Henry, Jessica Janiuk, Chris Johnson, Aaron Kalair, Sean Lindsay, Matthew Oldham, Stephen Orr, Jason Pike, Jessica Stodola, Andrew Vahey, and Mitchell Volk, for donating your valuable time to test these exercises and provide suggestions and feedback.

Thank you to my business associates Mitch Bullard, Kevin Gisi, Chris Johnson, Jeff Holland, Erich Tesky, Myles Steinhauser, Chris Warren, and Mike Weber for your support.

Thank you, Carissa, my wonderful wife and best friend. Your love and support make this all possible. I am forever grateful for all you do for me and our girls.

Finally, thank you, Ana, for being awesome, and thank you, Lisa, for all the hugs and text messages while I was writing. And for keeping me company on the couch while I wrote this.

How to Use This Book

Practice makes permanent.

A concert pianist practices many hours a day, learning music, practicing drills, and honing her skills. She practices the same piece of music over and over, learning every little detail to get it just right. Because when she performs, she wants to deliver a performance she is proud of for the people who spent their time and money to hear it.

A pro football player spends hours in the gym lifting, running, jumping, and doing drills over and over until he masters them. And then he practices the sport. He'll study plays and watch old game videos. And, of course, he'll play scrimmage and exhibition games to make sure he's ready to perform during the real contest.

A practitioner of karate spends a lifetime doing *kata*, a series of movements that imitate a fight or battle sequence, learning how to breathe and flex the right muscles at the right time. She may do the same series of movements thousands of times, getting better and better with each repetition.

The best software developers I've ever met approach their craft the same way. They don't go to work every day and practice on the employer's dime. They invest personal time in learning new languages and perfecting techniques in others. Of course they learn new things on the job, but because they're getting paid, there's an expectation that they are there to perform, not practice.

This book is all about practicing your craft as a programmer. Flip to a page in this book, crack open your text editor, and hammer out the program. Make your own variations on it. Do it in a language you've never used before. And get better and better each time you do it.

Who This Book Is For

This book is targeted at two main groups of programmers.

First, it's for beginning programming students to get additional practice beyond the classroom. You can't hone your skills just by doing your assignments. Your future employer will want you to be able to demonstrate critical thinking and problem-solving skills, and you need practice to develop those. This book gives you that practice in the form of real-world problems that many developers face but that are geared toward your abilities. Each chapter covers a fundamental component of programming and is a little more complex than the previous one, building on what you've learned and preparing you for the challenges that lie ahead, both in and out of the classroom.

Many beginning programmers are used to being told exactly how to solve a problem. They often learn a language by following a written tutorial that has some code they can type. And this is a great way to start writing code. But these programmers struggle when faced with open-ended problems that don't have the solution available. And as anyone who has experience can tell you, software development is full of open-ended problems. The exercises in this book help you develop those problem-solving skills so that you build the confidence to attack even larger problems—maybe even ones that nobody else has solved yet.

But this book is also for experienced programmers looking to get better at what they do. When I learned Go and Elixir, I used programs like the ones in this book. When I tried my hand at iOS development, I tried to write these programs. And every once in a while, I do these programs in a language I already know. I'm fluent in JavaScript and Ruby, and it's a great challenge to see if I can tackle one of these programs in a different way, using a different algorithm or pattern. When I started teaching Ruby and JavaScript full time, these programs helped me discover and explain the unique features of the languages I knew how to use but didn't quite fully understand. And so if you're an experienced developer, I encourage you to do the same. Try one of these programs in Haskell. Or try to write one of these programs in every language you know and compare the results. Challenge your coworkers to do one of these exercises a week and compare your solutions. Or use these programs to mentor the new junior developer on your team.

A Note for Educators

If you teach introductory programming at the high school or college level, you may find the exercises in this book useful in your class. I don't recommend using these as summative assessments though; people reading this book are

encouraged to share their solutions with others. But I do recommend using these as in-class exercises where students can work together. These exercises work well in a problem-based learning environment.

What's in This Book (And What's Not)

This book is written first and foremost to provide beginners with challenging problems they might face when first learning to program. Therefore, most of the problems are relatively simple in the beginning and gradually get more complex. The progression of exercises in this book makes practicing the fundamentals of programming challenging but fun and can accelerate the process of picking up a new language. In the first section, the programs simply take some input and manipulate the data into different output, giving you experience with how computer programs handle input and output operations. They're the kind of programs you'd do in your first week as a beginning programmer.

Next, you'll be challenged by writing programs that have you do calculations. Some of them are as simple as calculating the area of a room. But others involve financial and medical calculations similar to ones you may find on the job.

Then you'll increase the complexity of your programs by including decision logic and repetition logic, and you'll incorporate functions into them.

After that you'll find some problems that need to be solved using data structures like arrays and maps. These programs also require you to draw on some of the other problems you've solved before.

And, of course, no collection of programs would be complete without a bit of file input and output, so you'll get to practice reading data from files, processing it, and writing it back out.

Modern programs often talk with external services, so you'll find a few programs that have you work with data using third-party APIs.

Finally, a few larger programs at the end will require you to put together all the things you've learned.

In addition, each exercise includes some constraints that you'll have to follow when building the program as well as some challenges that ask you to build on the program. If you've never programmed before, you may want to skip the challenges and revisit them when you improve your skills. But if you've got some experience under your belt, you may want to accept these challenges right away if you think the program is too simple. Some of the challenges will be difficult depending on the programming language you've chosen. For example, if you're creating these programs with JavaScript and HTML, making a GUI version of the program will be easy. If you're doing this with Java, it will be a lot more work. So feel free to modify the challenges as you see fit.

However, what you *won't* find in this book are the solutions to the programs. If you think hard enough and use all of the resources at your disposal, you'll be able to figure out how to solve these problems on your own, which is the point of this book.

One last thing: you won't find the infamous interview questions here. There's no FizzBuzz. You won't need to invert binary trees, nor will you need to write a quicksort algorithm (unless you want to as part of a solution). If you're looking for things like that, you'll have to look elsewhere. Those kinds of problems have value but are often more difficult to do because it's not clear why you're doing them. That makes them unapproachable, which creates a barrier to learning.

The problems in this book are simple, real-world problems that you can easily relate to and that will help you practice solving problems with code.

What You Need

All you need is your favorite development environment—or even one you've never used. This book is programming-language agnostic. Pick a language, grab that language's reference guide, and dive in. Be warned though; the programming language you choose will determine how easy, or difficult, these programs are. For example, if you choose to do this book with Python or Ruby, then developing graphical user interfaces won't be easy. And if you choose to use JavaScript in the browser, then working with external files and web services will be much more complex than with other languages. Your approach to problems will be much different if you choose a functional programming language over an object-oriented one. But that's the real value of these exercises; they'll help you learn a language *and how that language is different from what you already know.*

You should have an Internet connection so you can do some of the programs that use third-party services and participate in the community for this book.

Online Resources

The book's website [1] has a discussion forum where you can discuss the book with other developers. Feel free to post solutions there in your favorite language and discuss your solutions with other readers. One of the most fascinating things about programming is how people approach solving problems differently and how each developer has his or her own style.

Footnotes

[1] http://pragprog.com/titles/bhwb

Chapter 1

Turning Problems into Code

If you're new to programming, you may wonder how experienced developers can look at a problem and turn it into runnable code. It turns out that writing the actual code is only a small part of the process. You have to break down the problem before you can solve it. If you've ever watched an experienced programmer, it may look like they just cracked open their code editor and banged out a solution. But over the years, they've broken down hundreds, if not thousands, of problems, and they can see patterns. If you're just starting out, you might not know how to do that. So in this chapter we'll look at one way to break down problems and turn them into code. And you can use this approach to conquer the problems in the rest of this book.

Understanding the Problem

One of the best ways to figure out what you have to do is to write it down. If I told you that I wanted a tip calculator application, would that be enough information for you to just go and build one? Probably not. You'd probably have to ask me a few questions. This is often called gathering requirements, but I like to think of it as figuring out what features the program should have.

Think of a few questions you could ask me that would let you get a clearer picture of what I want. What do you need to know to build this application?

Got some questions? Great. Here are some you might ask:

- What formula do you want to use? Can you explain how the tip should be calculated?
- What's the tip percentage? Is it 15% or should the user be able to modify it?
- What should the program display on the screen when it starts?
- What should the program display for its output? Do you want to see the tip and the total or just the total?

Once you have the answers to your questions, try writing out a problem statement that explains exactly what you're building. Here's the problem statement for the program we're going to build:

> Create a simple tip calculator. The program should prompt for a bill amount and a tip rate. The program must compute the tip and then display both the tip and the total amount of the bill.

Example output:

```
What is the bill? $200
What is the tip percentage? 15
The tip is $30.00
The total is $230.00
```

Joe asks:

What do I do with complex programs?

Break down the large program into smaller features that are easier to manage. If you do that, you'll have a better chance of success because each feature can be fleshed out. And most complex applications out there are composed of many smaller programs working together. That's how command-line tools in Linux work; one program's output can be another program's input.

If you're ready to open your text editor and hammer out the code, you're jumping way ahead of yourself. You see, if you don't take the time to carefully design the program, you might end up with something that works but isn't good quality. And unfortunately, it's very easy for something like that to get out into the wild. For example, you hammer out your program without testing, planning, or documenting it, and your boss sees it, thinks it's done, and tells you to release it. Now you have untested, unplanned code in production, and you'll probably be asked to make changes to it later. Code that's poorly designed is very hard to maintain or extend. So let's take this tip calculator example and go through a simple process that will help you understand what you're supposed to build.

Discovering Inputs, Processes, and Outputs

Every program has inputs, processes, and outputs, whether it's a simple program like this one or a complex application like Facebook. In fact, large applications are simply a bunch of smaller programs that communicate. The output of one program becomes the input of another.

You can ensure that both small and large programs work well if you take the time to clearly state what these inputs, processes, and outputs are. An easy way to do that, if you have a clear problem statement, is to look at the nouns and verbs in that statement. The nouns end up becoming your inputs and outputs, and the verbs will be your processes. Look at the problem statement for our tip calculator:

> Create a simple tip calculator. The program should prompt for a bill amount and a tip rate. The program must compute the tip and then display both the tip and the total amount of the bill.

First, look for the nouns. Circle them if you like, or just make a list. Here's my list:

- bill amount
- tip rate
- tip
- total amount

Now, what about the verbs?

- prompt
- compute
- display

So we know we have to prompt for inputs, do some calculations, and display some outputs. By looking at the nouns and verbs, we can get an idea of what we're being asked to do.

Of course, the problem statement won't always be clear. For example, the problem statement says we need to calculate the tip, but it then says we need to display the tip and the total. It's implied that we'll need to also add the tip to the original bill amount to get that output. And that's one of the challenges of building software. It isn't spelled out to you 100% of the time. But as you gain more experience, you'll be able to fill in the gaps and read between the lines.

So with a little bit of sleuthing, we determine that our inputs, processes, and outputs for this program look like this:

- Inputs: bill amount, tip rate
- Processes: calculate the tip
- Outputs: tip amount, total amount

Are we ready to start producing some code? Not just yet.

Driving Design with Tests

One of the best ways to design and develop software is to think about the result you want to get right from the start. Many professional software developers do this using a formal process called *test-driven development*, or *TDD*. In TDD, you write bits of code that test the outputs of your program or the outputs of the individual programs that make up a larger program. This process of testing as you go guides you toward good design and helps you think about the issues your program might have.

TDD does require some knowledge about the language you're using and a little more experience than the beginning developer has out of the gate.

However, the essence of TDD is to think about what the expected result of the program is ahead of time and then work toward getting there. And if you do that before you write code, it'll make you think beyond what the initial requirements say. So if you're not quite comfortable doing formal TDD, you can still get many of the benefits by creating simple test plans. A test plan lists the program's inputs and its expected result.

Here's what a test plan looks like:

```
Inputs:
Expected result:
Actual result:
```

You list the program inputs and then write out what the program's output should be. And then you run your program and compare the expected result with the actual result your program gives out.

Let's put this into practice by thinking about our tip calculator. How will we know what the program's output should be? How will we know if we calculate it correctly?

Well, let's *define* how we want things to work by using some test plans. We'll do a very simple test plan first.

```
Inputs:
  bill amount: 10
  tip rate: 15
Expected result:
  Tip: $1.50
  Total: $11.50
```

That test plan tells us a couple things. First, it tells us that we'll take in two inputs: a bill amount of `10` and a tip rate of `15`. So we'll need to handle

converting the tip rate from a whole number to a decimal when we do the math. It also tells us we'll print out the tip and total formatted as currency. So we know that we'd better do some conversions in our program.

Now, one test isn't enough. What if we used **11.25** as an input? Using a test plan, what should the output be? Try it out. Fill in the following plan:

```
Input:
  bill amount: 11.25
  tip rate: 15
Expected result:
  Tip: ???
  Total: ???
```

I assume you just went and used a calculator to figure out the tip. If you ran the calculation, your calculator probably said the tip should be **1.6875.**

But is that realistic? Probably not. We would probably round up to the nearest cent. So our test plan would look like this:

```
Input:
  bill amount: 11.25
  tip rate: 15
Expected result:
  Tip: $1.69
  Total: $12.94
```

We just used a test to design the functionality of our program; we determined that our program will need to round up the answer.

When you're going through the exercises in this book, take the time to develop at least four test plans for every program, and try to think of as many scenarios as you can for how people might break the program. And as you get into the more complicated problems, you may need a lot more test plans.

If you're an experienced software developer who wants to get started with TDD, you should use the exercises in this book to get acquainted with the libraries and tools your favorite language has to offer. You can find a list of testing frameworks for many programming languages at Wikipedia.[2] You can read Kent Beck's *Test-Driven Development: By Example* to gain more insight into how to design code with tests, or you can investigate any number of more language-specific resources on TDD.

So now that we have a clearer picture of the features the program will have, we can start putting together the *algorithm* for the program.

Writing the Algorithm in Pseudocode

An algorithm is a step-by-step set of operations that need to be performed. If you take an algorithm and write code to perform those operations, you end up with a computer program.

If you're new to programming and not entirely comfortable with a programming language's syntax yet, you should consider writing out the algorithm using *pseudocode*, an English-like syntax that lets you think about the logic without having to worry about paper. Pseudocode isn't just for beginners; experienced programmers will occasionally write some pseudocode on a whiteboard when working with teammates to solve problems, or even by themselves.

There's no "right way" to write pseudocode, although there are some widely used terms. You might use **Initialize** to state that you're setting an initial value, **Prompt** to say that you're prompting for input, and **Display** to indicate what you're displaying on the screen.

Here's how our tip calculator might look in pseudocode:

```
TipCalculator
  Initialize billAmount to 0
  Initialize tip to 0
  Initialize tipRate to 0
  Initialize total to 0

  Prompt for billAmount with "What is the bill amount?"
  Prompt for tipRate with "What is the tip rate?"

  convert billAmount to a number
  convert tipRate to a number

  tip = billAmount * (tipRate / 100)
  round tip up to nearest cent
  total = billAmount + tip

  Display "Tip: $" + tip
  Display "Total: $" + total
End
```

That's a rough stab at how our program's algorithm will look. We'll have to set up some variables, make some decisions based on the input, do some conversions, and put some output on the screen. I recommend including details like variable names and text you'll display on the screen in pseudocode, because it helps you think more clearly about the end result of the program.

Is this the best way we could write the program? Probably not. But that's not the

point. By writing pseudocode, we've created something we can show to another developer to get feedback, and it didn't take long to throw it together.

Best of all, we can use this as a blueprint to code this up in any programming language. Notice that our pseudocode makes no assumptions about the language we might end up using, but it does guide us as to what the variable names will be and what the output to the end user will look like.

Once you write your initial version of the program and get it working, you can start tweaking your code to improve it. For example, you may split the program into functions, or you may do the numerical conversions inline instead of as separate steps. Just think of pseudocode as a planning tool.

Writing the Code

Now it's your turn. Using what you've learned, can you write the code for this program? Give it a try. Just keep these constraints in mind as you do so:

Constraints

- Enter the tip as a percentage. For example, a 15% tip would be entered as `15`, not `0.15`. Your program should handle the division.
- Round fractions of a cent up to the next cent.

If you can't figure out how to enforce these constraints, write the program without them and come back to it later. The point of these exercises is to practice and improve.

And if this program is too challenging for you right now, jump ahead and do some of the easier programs in this book first, and then come back to this one.

Challenges

When you've finished writing the basic version of the program, try tackling some additional challenges:

- Ensure that the user can enter only numbers for the bill amount and the tip rate. If the user enters non-numeric values, display an appropriate message and exit the program. Here's a test plan as an example:

```
Input:
  bill amount: abcd
  tip rate: 15
Expected result: Please enter a valid number for
                 the bill amount.
```

- Instead of displaying an error message and exiting the program, keep asking the user for correct input until it is provided.
- Don't allow the user to enter a negative number.
- Break the program into functions that do the computations.
- Implement this program as a GUI program that automatically updates the values when any value changes.
- Instead of the user entering the value of the tip as a percentage, have the user drag a slider that rates satisfaction with the server, using a range between 5% and 20%.

Onward!

Try to tackle each problem in the book using this strategy to get the most out of the experience. Discover your inputs, processes, and outputs. Develop some test plans, come up with some pseudocode, and write the program. Then accept the various challenges after each program. Or go in your own direction. Or write the program in as many languages as you can.

But most of all, have fun and enjoy learning.

Footnotes

[2] https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks

---

Chapter 2

Input, Processing, and Output

Getting input from the user and converting it to something meaningful is one of the fundamental pieces of programming. Software developers are always turning data into information that can be used to make decisions. That data may come from the keyboard, a mouse, a touch, a swipe, or even a game controller. The computer has to react to it, process it, and do something useful.

The exercises in this chapter will help you get acquainted with how to get input from the user and process it to produce output. You'll build up strings, do a little math, and get your feet wet with the programming language you're using. They're simple problems, but they'll help you build up your confidence as a programmer; the problems in the later chapters of the book are more complex.

Each exercise has additional challenges you can do if you feel up to the task. If you're new to programming, some of the challenges will ask you to use techniques you might not be familiar with yet. Feel free to skip them; you can always come back and do those challenges later.

Ready? Set? Go!

1 Saying Hello

The "Hello, World" program is the first program you learn to write in many languages, but it doesn't involve any input.

So create a program that prompts for your name and prints a greeting using your name.

Example Output

```
What is your name? Brian
Hello, Brian, nice to meet you!
```

Constraint

- Keep the input, string concatenation, and output separate.

Challenges

- Write a new version of the program without using any variables.
- Write a version of the program that displays different greetings for different people. This would be a good challenge to try after you've completed the exercises in Chapter 4, *Making Decisions* and Chapter 7, *Data Structures*.

2 Counting the Number of Characters

Create a program that prompts for an input string and displays output that shows the input string and the number of characters the string contains.

Example Output

```
What is the input string? Homer
Homer has 5 characters.
```

Constraints

- Be sure the output contains the original string.
- Use a single output statement to construct the output.
- Use a built-in function of the programming language to determine the length of the string.

Challenges

- If the user enters nothing, state that the user must enter something into the program.
- Implement this program using a graphical user interface and update the character counter every time a key is pressed. If your language doesn't have a particularly friendly GUI library, try doing this exercise with HTML and JavaScript instead.

3 Printing Quotes

Quotation marks are often used to denote the start and end of a string. But sometimes we need to print out the quotation marks themselves by using *escape characters*.

Create a program that prompts for a quote and an author. Display the quotation and author as shown in the example output.

Example Output

```
What is the quote? These aren't the droids you're looking for.
Who said it? Obi-Wan Kenobi
Obi-Wan Kenobi says, "These aren't the droids
you're looking for."
```

Constraints

- Use a single output statement to produce this output, using appropriate string-escaping techniques for quotes.
- If your language supports string interpolation or string substitution, don't use it for this exercise. Use string concatenation instead.

Challenge

- In Chapter 7, *Data Structures*, you'll practice working with lists of data. Modify this program so that instead of prompting for quotes from the user, you create a structure that holds quotes and their associated attributions and then display all of the quotes using the format in the example. An array of maps would be a good choice.

4 Mad Lib

Mad libs are a simple game where you create a story template with blanks for words. You, or another player, then construct a list of words and place them into the story, creating an often silly or funny story as a result.

Create a simple mad-lib program that prompts for a noun, a verb, an adverb, and an adjective and injects those into a story that you create.

Example Output

```
Enter a noun: dog
Enter a verb: walk
Enter an adjective: blue
Enter an adverb: quickly
Do you walk your blue dog quickly? That's hilarious!
```

Constraints

- Use a single output statement for this program.
- If your language supports string interpolation or string substitution, use it to build up the output.

Challenges

- Add more inputs to the program to expand the story.
- Implement a branching story, where the answers to questions determine how the story is constructed. You'll explore this concept more in the problems in Chapter 4, *Making Decisions*.

5 Simple Math

You'll often write programs that deal with numbers. And depending on the programming language you use, you'll have to convert the inputs you get to numerical data types.

Write a program that prompts for two numbers. Print the sum, difference, product, and quotient of those numbers as shown in the example output:

Example Output

```
What is the first number? 10
What is the second number? 5
10 + 5 = 15
10 - 5 = 5
10 * 5 = 50
10 / 5 = 2
```

Constraints

- Values coming from users will be strings. Ensure that you convert these values to numbers before doing the math.
- Keep the inputs and outputs separate from the numerical conversions and other processing.
- Generate a single output statement with line breaks in the appropriate spots.

Challenges

- Revise the program to ensure that inputs are entered as numeric values. Don't allow the user to proceed if the value entered is not numeric.
- Don't allow the user to enter a negative number.
- Break the program into functions that do the computations. You'll explore functions in Chapter 5, *Functions*.
- Implement this program as a GUI program that automatically updates the values when any value changes.

6 Retirement Calculator

Your computer knows what the current year is, which means you can incorporate that into your programs. You just have to figure out how your programming language can provide you with that information.

Create a program that determines how many years you have left until retirement and the year you can retire. It should prompt for your current age and the age you want to retire and display the output as shown in the example that follows.

Example Output

```
What is your current age? 25
At what age would you like to retire? 65
You have 40 years left until you can retire.
It's 2015, so you can retire in 2055.
```

Constraints

- Again, be sure to convert the input to numerical data before doing any math.
- Don't hard-code the current year into your program. Get it from the system time via your programming language.

Challenge

- Handle situations where the program returns a negative number by stating that the user can already retire.

What You Learned

These problems were pretty simple, but hopefully they got you thinking about keeping input, processing, and output separate. When programs are simple, it's tempting to just do some math or string concatenation inside the program's output statements, but as your programs get more complex, you'll find you need to break things into reusable components. You'll be glad you were disciplined from the start.

Head on over to the next chapter. It's time to do some more serious math.

Chapter 3

Calculations

You've done some basic math already, but now it's time to dive into more complex math. The exercises in this chapter are a little more challenging. You'll work with formulas for numerical conversion and you'll create some real-world financial programs, too.

These programs will test your knowledge of the order of operations. "Please Excuse My Dear Aunt Sally," or *PEMDAS*, is a common way to remember the order of operations:

- *P*arentheses
- *E*xponents
- *M*ultiplication
- *D*ivision
- *A*ddition
- *S*ubtraction

The computer will always follow these rules, even if you don't want it to. So the exercises in this chapter will have you thinking about adding parentheses to your programs to ensure the output comes out correctly.

You'll want to make good use of test plans for these exercises, too, because you're going to be dealing with precision issues. If you work with decimal numbers in many programming languages, you may encounter some interesting, and unexpected, results. For example, if you add `0.1` and `0.2` in Ruby, you'll get this:

```
> 0.1 + 0.2
 => 0.30000000000000004
```

This happens in JavaScript too. And multiplication can make things even more interesting. Look at this code:

```
> 1.25 * 0.055
 => 0.06875
```

Should that answer be rounded down to **0.06** or up to **0.07**? It depends entirely on your business rules. If your answer must be a whole number, you may have to round it up.

Things get even messier with currency. One of the most common issues new programmers face occurs when they try to use floating-point numbers for currency. This will result in precision errors.

One common approach is to represent money using whole numbers. So instead of working with **1.25**, work with **125**. Do the math, and then shift the decimal back when finished. Here's an example, again in Ruby:

```
> cents = 1.25 * 100.0
 => 125.0
> tax = cents * 0.055
 => 6.875
> tax = tax.round / 100.0
 => 0.07
```

You may need to be a lot more precise than this. These floating-point precision issues exist in many programming languages, and so there are libraries that make working with currency much better. For example, Java has the **BigDecimal** data type that even lets you specify what type of "banker's rounding" you need to do. When you're working on these problems, think carefully about how you need to handle precision. When you do problems for real, especially if it's some kind of financial work, learn how the business you're working with rounds numbers.

One last thing before you dive in: the exercises in this chapter might seem to get a little repetitive toward the end if you're experienced. But for beginners, repetition builds up confidence quickly. It's the same reason you do practice drills in sports or practice your scales over and over in music. By doing several similar problems, you build up your problem-solving skills and improve your speed at breaking down problems. And that translates into success on the job.

7 Area of a Rectangular Room

When working in a global environment, you'll have to present information in both metric and Imperial units. And you'll need to know when to do the conversion to ensure the most accurate results.

Create a program that calculates the area of a room. Prompt the user for the length and width of the room in feet. Then display the area in both square feet and square meters.

Example Output

```
What is the length of the room in feet? 15
What is the width of the room in feet? 20
You entered dimensions of 15 feet by 20 feet.
The area is
300 square feet
27.871 square meters
```

The formula for this conversion is

$$m^2 = f^2 \times 0.09290304$$

Constraints

- Keep the calculations separate from the output.
- Use a constant to hold the conversion factor.

Challenges

- Revise the program to ensure that inputs are entered as numeric values. Don't allow the user to proceed if the value entered is not numeric.
- Create a new version of the program that allows you to choose feet or meters for your inputs.
- Implement this program as a GUI program that automatically updates the values when any value changes.

8 Pizza Party

Division isn't always exact, and sometimes you'll write programs that will need to deal with the leftovers as a whole number instead of a decimal.

Write a program to evenly divide pizzas. Prompt for the number of people, the number of pizzas, and the number of slices per pizza. Ensure that the number of pieces comes out even. Display the number of pieces of pizza each person should get. If there are leftovers, show the number of leftover pieces.

Example Output

```
How many people? 8
How many pizzas do you have? 2

8 people with 2 pizzas
Each person gets 2 pieces of pizza.
There are 0 leftover pieces.
```

Challenges

- Revise the program to ensure that inputs are entered as numeric values. Don't allow the user to proceed if the value entered is not numeric.
- Alter the output so it handles pluralization properly, for example:

  Each person gets 2 pieces of pizza.

  or

  Each person gets 1 piece of pizza.

  Handle the output for leftover pieces appropriately as well.

- Create a variant of the program that prompts for the number of people and the number of pieces each person wants, and calculate how many full pizzas you need to purchase.

9 Paint Calculator

Sometimes you have to round up to the next number rather than follow standard rounding rules.

Calculate gallons of paint needed to paint the ceiling of a room. Prompt for the length and width, and assume one gallon covers 350 square feet. Display the number of gallons needed to paint the ceiling as a *whole number.*

Example Output

```
You will need to purchase 2 gallons of
paint to cover 360 square feet.
```

Remember, you can't buy a partial gallon of paint. You must round up to the next whole gallon.

Constraints

- Use a constant to hold the conversion rate.
- Ensure that you round *up* to the next whole number.

Challenges

- Revise the program to ensure that inputs are entered as numeric values. Don't allow the user to proceed if the value entered is not numeric.
- Implement support for a round room.
- Implement support for an L-shaped room.
- Implement a mobile version of this app so it can be used at the hardware store.