

MORE FROM

NO STARCH PRESS

LEARN TO PROGRAM WITH MINECRAFT

CRAIG RICHARDSON



THINK LIKE A PROGRAMMER, PYTHON EDITION

V. ANTON SPRAUL



THE RUST PROGRAMMING LANGUAGE

STEVE KLABNIK AND CAROL NICHOLS,
WITH CONTRIBUTIONS FROM THE RUST COMMUNITY



THE BOOK OF R

TILMAN M. DAVIES



LEARN JAVA THE EASY WAY

BRYSON PAYNE



ELOQUENT JAVASCRIPT, 2ND EDITION

MARIJN HAVERBEKE



UNDERSTANDING ECMASCRIPT 6

NICHOLAS C. ZAKAS



WICKED COOL SHELL SCRIPTS, 2ND EDITION

DAVE TAYLOR AND BRANDON PERRY



THE LINUX COMMAND LINE

WILLIAM E. SHOTTS, JR.

LEARN TO PROGRAM WITH MINECRAFT®

TRANSFORM YOUR WORLD
WITH THE POWER OF PYTHON

CRAIG RICHARDSON



MISSION #38: THE MIDAS TOUCH

Midas is a king of legend. Everything he touched turned to gold. Your mission is to write a program that changes every block below the player to gold—except for air and water, of course, or you’d be in real trouble! Recall that the gold block has a value of 41, still water is 9, and air is 0.

midas.py

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()

air = 0
water = 9

❶ # Add an infinite while loop here
   pos = mc.player.getTilePos()
   blockBelow = mc.getBlock(pos.x, pos.y - 1, pos.z)

❷ # Add if statement here
   mc.setBlock(pos.x, pos.y - 1, pos.z, 41)
```

Open IDLE and create a new file. Save the file as *midas.py* in the *whileLoops* folder. You need to add a bit more to the program so it can do what you need it to do. First, you’ll add an infinite while loop ❶. Remember that an infinite while loop has a condition that is always True. You also need to add an if statement that checks whether the block below the player is not equal to air and not equal to still water ❷. The value of the block below the player is stored in the `blockBelow` variable, and the values for air and water are stored in the `air` and `water` variables.

When you’ve completed the program, save it and run it. The player should leave a trail of gold behind them. When you jump in water or fly in the air, the blocks below you should not change. Figure 7-7 shows the program in action.



Figure 7-7: Every block I walk on turns to gold.

To exit the infinite loop, go to **Shell ▶ Restart Shell** in your IDLE shell or click in the shell and press CTRL-C.

BONUS OBJECTIVE: I'M A PLOWMAN

You can change *midas.py* to serve a variety of purposes. How would you change it so it automatically changes dirt blocks to hoed farmland? How about changing dirt blocks to grass blocks?

ENDING A WHILE LOOP WITH BREAK

With while loops, you have complete control over how and when the loop ends. So far you've only used conditions to end loops, but you can also use a break statement. The break statement lets your code immediately exit a while loop. Let's look at this concept!

One way to use break statements is to put them in an if statement nested in the loop. Doing so immediately stops the loop when the if statement's condition is True. The following code continually asks for user input until they type "exit":

```
❶ while True:
❷     userInput = input("Enter a command: ")
❸     if userInput == "exit":
❹         break
        print(userInput)
❺ print("Loop exited")
```

This is an infinite loop because it uses `while True` ❶. Each time the loop repeats, it asks for the user to enter a command ❷. The program checks whether the input is "exit" ❸ using an if statement. If the input meets the condition, the break statement stops the loop from repeating ❹, and the program continues on the line immediately after the body of the loop, printing "Loop exited" to the Python shell ❺.

MISSION #39: CREATE A PERSISTENT CHAT WITH A LOOP

In Mission #13 (page 76), you created a program that posts the user's message to chat using strings, input, and output. Although this program was useful, it was quite limited because you had to rerun the program every time you wanted to post a new message.

In this mission, you'll improve your chat program using a while loop so users can post as many messages as they want without restarting the program.

Open the `userChat.py` file in the `strings` folder and then save it as `chatLoop.py` in the `whileLoops` folder.

To post a new message every time you want to without rerunning the program, add the following to your code:

1. Add an infinite `while` loop to the program.
2. Add an `if` statement to the loop to check whether the user's input is "exit". If the input is "exit", the loop should break.
3. Make sure the `userName` variable is defined before the start of the loop.

When you've added the changes, save your program and run it. A prompt in the Python shell will ask you to type in a username. Do this and press ENTER. The program will then ask you to enter a message. Type a message and then press ENTER. The program will keep asking you to enter a message until you type exit. Figure 7-8 shows my chat program running.

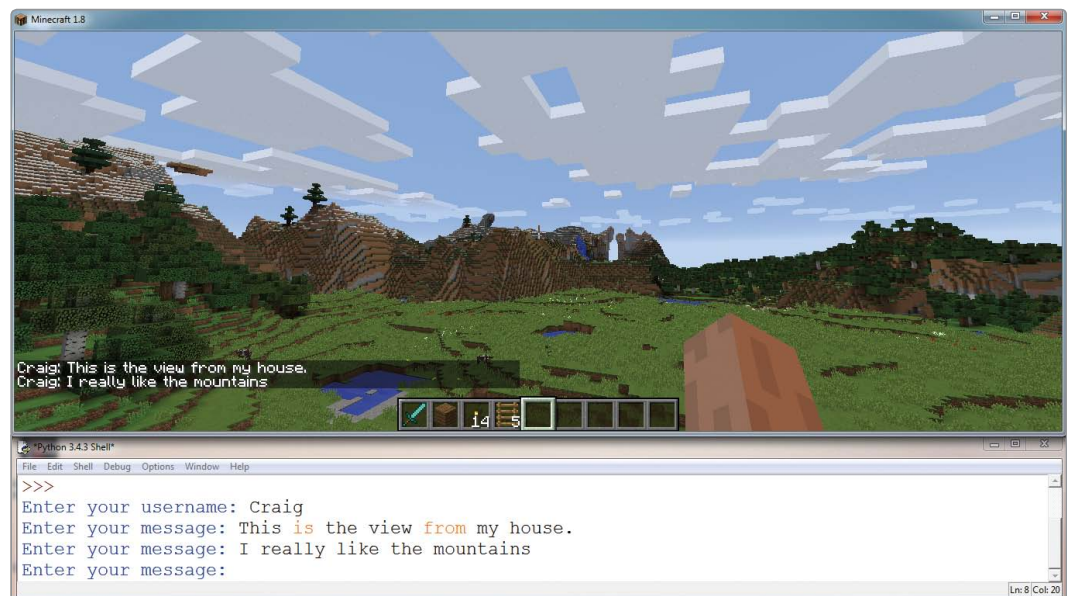


Figure 7-8: I'm chatting with myself.

BONUS OBJECTIVE: BLOCK CHAT

Expand the chat feature so users can create blocks. For example, if the user enters "wool", the program creates a wool block. You can do this by adding `elif` statements to your `if` statement to check user input.

WHILE-ELSE STATEMENTS

Like an if statement, while loops can have secondary conditions triggered by else statements.

The else statement executes when the condition of a while statement is False. Unlike the body of a while statement, the else statement will execute only once, as shown here:

```
message = input("Please enter a message.")

while message != "exit":
    print(message)
    message = input("Please enter a message.")
else:
    print("User has left the chat.")
```

This loop repeats as long as the message entered is not equal to "exit". If the message is "exit", the loop will stop repeating, and the body of the else statement will print "User has left the chat."

If you use a break statement in the while statement, the else isn't executed. The following code is similar to the preceding example but includes a nested if statement and a break statement. When the user types abort instead of exit, the chat loop will exit without printing the "User has left the chat." message to the chat.

```
message = input("Please enter a message.")

while message != "exit":
    print(message)
    message = input("Please enter a message.")
    if message == "abort":
        break
else:
    print("User has left the chat.")
```

The if statement checks whether the message entered is "abort". If this is True, the break statement runs and the loop will exit. Because the break statement was used, the body of the else statement will not run, and "User has left the chat." will not be printed.

MISSION #40: HOT AND COLD

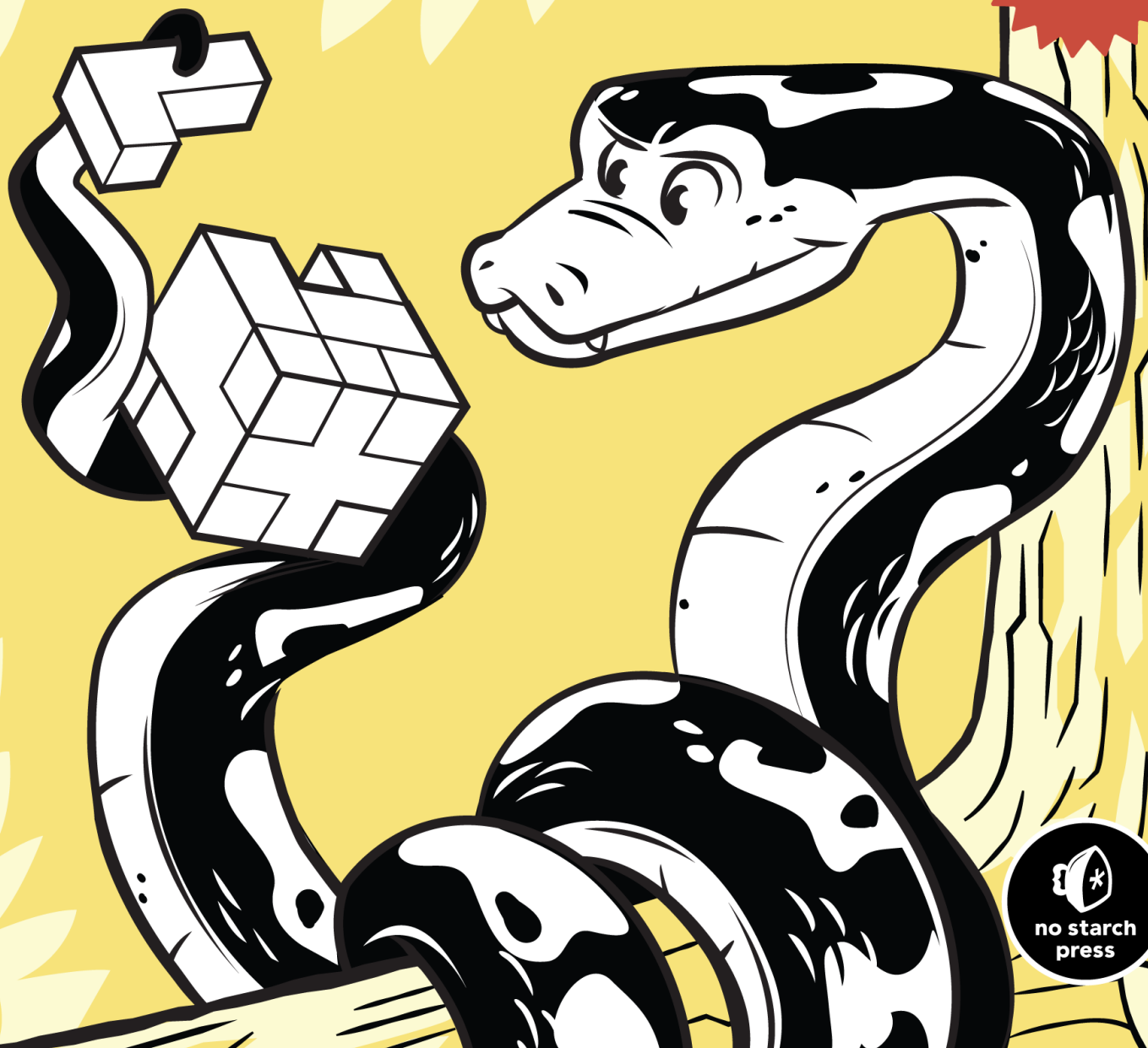
In this mission, we'll create a Hot and Cold game in Minecraft. If you've never played, the idea is that your friend hides an object and you have to find it. Your friend gives you hints based on how far away from the object you are. If you're close, your friend says "Hot," and if you're far away, they'll say "Cold." When you're right next to the object, they'll say "You're on fire!" and if you're very far away, they'll say "Freezing!"

THINK LIKE A PROGRAMMER

A BEGINNER'S GUIDE TO
PROGRAMMING AND PROBLEM SOLVING

V. ANTON SPRAUL

PYTHON
EDITION



3-4. Write a program that reads at-bats and hits and displays the batting average. (Cricket fans can compute batting average from runs and outs if they prefer.)

3-5. This one will probably require a little Googling. Write a program that computes an NCAA quarterback rating: read the relevant statistics and display the answer.

Problem Solving With Variables and Mathematics

Now we're ready to look at problems that require real problem solving. The programming itself is just as simple as the previous examples, using the same basic combination of input, calculations, and output, and the resulting programs are just as short. The difference is in the thought process that must occur before the code is written.

An important note: while we'll be solving these problems using the Python we've seen to this point, some elements in these problems could be solved more directly using techniques we'll cover in later chapters. This pattern will repeat throughout this book. We'll try to push each language concept as far as it can go, even though there may be other ways to solve the same problem around the corner.

Why do this? For one thing, we don't want to wait until we've covered most of the language to start learning problem solving. This early exposure to problem solving will develop your problem-solving skills better than waiting. A chef who first learns to create a variety of tasty dishes using a single pan and a few, simple ingredients will be better off than a chef who can't cope with anything less than a fully-stocked kitchen. In the same way, solving problems with restricted programming syntax will allow you to fully understand the capabilities of each element of programming and help you to unlock all your creative potential as a problem solver. You'll also gain a deeper appreciation of why programming languages have the features they do when you explore the limitations of earlier features.

As explained in the introduction, the Python community encourages a set of concepts they call Pythonic programming, and one Pythonic concept is that there is one "right" general approach for each particular problem. Because of this, a solution that deliberately avoids using more advanced language features will not be Pythonic. But remember that the point of this book is for you to learn to think like a programmer—to learn how to solve programming problems on your own. The only way to know what the Pythonic solution is for a particular problem is to ask someone, and you can't learn to solve problems on your own when someone else is giving you the answers.

So we'll write programs using the syntax we know at each point, and trust that our solutions will become more Pythonic as we become more knowledgeable and proficient.

Packs and Cans

Our first problem involves monetary calculations, but there's more to it than that:

Problem: Efficient Soda Buying

A local store sells six-packs of soda for \$3.29. Individual cans can be bought for 90 cents a can. Write a program to read the total number of soda cans desired and display how many packs should be bought to result in the lowest cost.

At first glance, this problem might seem just as straightforward as those seen previously, but it isn't. As a rule, and especially as a beginning programmer, never assume anything is trivial in programming. Always have a plan, and don't just jump into coding. If someone wanted to buy 22 cans, for example, the right number of packs and individual cans is not immediately obvious, and even less obvious is how we can produce general formulas for the answers.

Making a Table

Making a table of sample input and output is a good way to start when the right output isn't clear. Table 3-4 shows a range of input (the desired number of soda cans) from 1–12, the number of packs and individual cans that should be bought to result in the lowest cost, and that cost.

Table 3-4: Sample Input and Output for Efficient Soda Buying

Cans Needed (Input)	Six-Packs (Output)	Individual Cans	Total Cost
1	0	1	\$0.90
2	0	2	\$1.80
3	0	3	\$2.70
4	1	0	\$3.29
5	1	0	\$3.29
6	1	0	\$3.29
7	1	1	\$4.19
8	1	2	\$5.09
9	1	3	\$5.99
10	2	0	\$6.58
11	2	0	\$6.58
12	2	0	\$6.58

As you can see, buying individual cans is better when buying 1–3 cans, then it becomes more economical to buy a six-pack, and this pattern repeats as the number of cans increases. Creating the table provides data we can use to test our program once it is written. Also, the table may give us some

hints in writing our program. The output, shown in the second column of the table, has a definite pattern, but how to produce that pattern is not immediately clear.

Guessing and Testing

Because the six-pack holds six cans, it's logical to think that figuring out the number of packs we should buy will involve dividing by six. In fact, if someone wanted to buy an exact multiple of six cans, simply dividing the number of cans by six would be the right answer. That's not the right answer here, but let's see how wrong it actually is.

To do that, let's augment the previous table with a new column that shows the result of dividing the total number of cans by six. Because we can't buy part of a six-pack, we'll use floor division, Python's `//` operator. The result is shown as Table 3-5.

Table 3-5: Floor-Division Results

Cans Needed (Input)	Six-Packs (Output)	Cans // 6
1	0	0
2	0	0
3	0	0
4	1	0
5	1	0
6	1	1
7	1	1
8	1	1
9	1	1
10	2	1
11	2	1
12	2	2

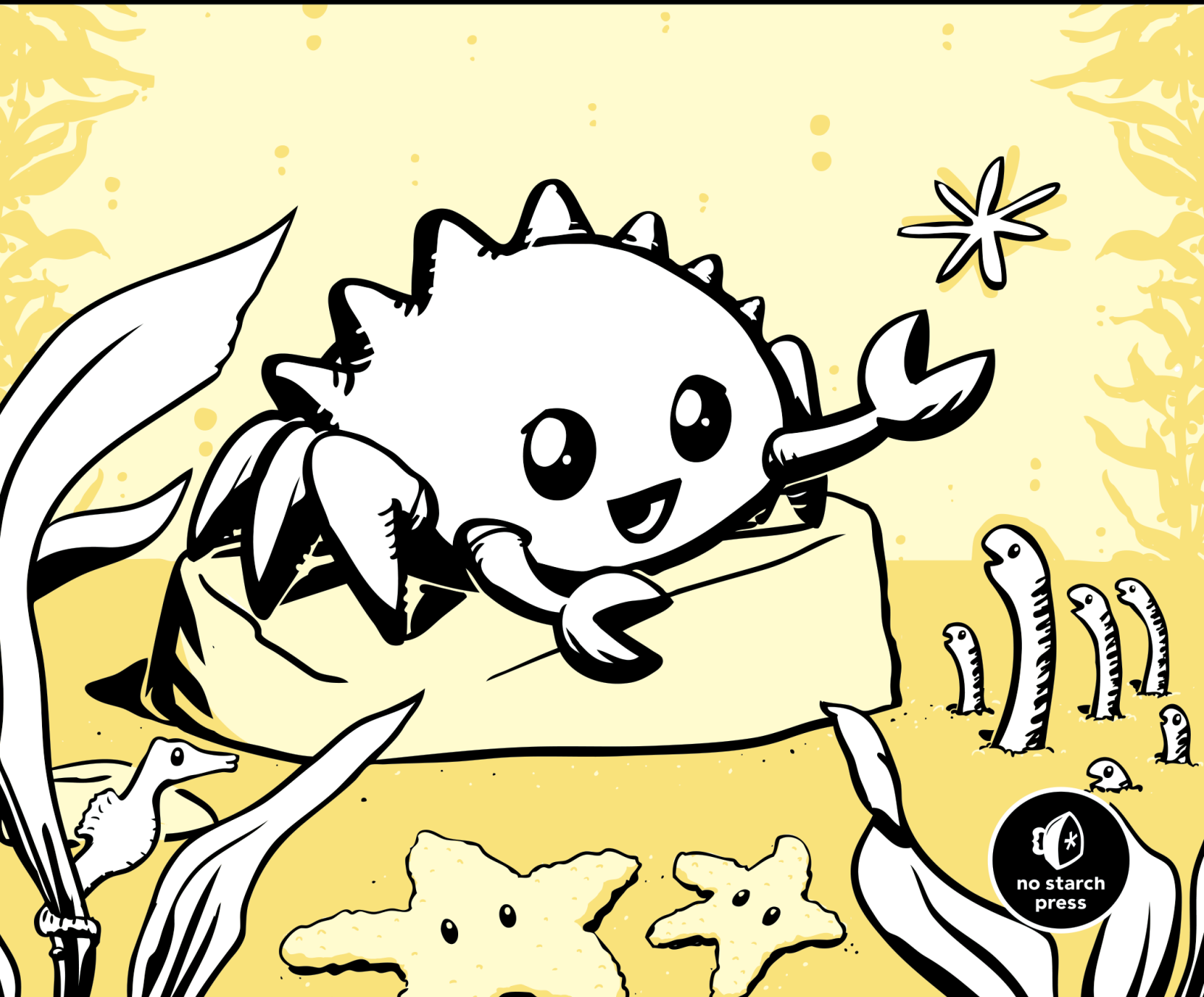
This is what I call *guessing and testing*; when you aren't sure about a mathematical formula, make an educated guess, compare the results to what you need, and see if you can bring the two together. In this case, the floor-division produces the right results, but two rows below where we would like them. In other words, if we could just shift the third column up two rows, it would match the desired output.

A Formula for the Pattern

Maybe that's a clue to a solution. What if we used addition to shift the results? If we added 2 to the number of cans before the floor division, then, for example, an input of 4 would produce the results on row 6 of the third column. Let's apply this idea to our table to make sure it works (Table 3-6).

THE RUST PROGRAMMING LANGUAGE

STEVE KLABNIK AND CAROL NICHOLS,
WITH CONTRIBUTIONS FROM THE RUST COMMUNITY



Setting Up a New Project

To set up a new project, go to the *projects* directory that you created in Chapter 1, and make a new project using Cargo, like so:

```
$ cargo new guessing_game --bin
$ cd guessing_game
```

The first command, `cargo new`, takes the name of the project (`guessing_game`) as the first argument. The `--bin` flag tells Cargo to make a binary project, similar to the one in Chapter 1. The second command changes to the new project’s directory.

Look at the generated *Cargo.toml* file:

```
Filename: Cargo.toml [package]
name = "guessing_game"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]

[dependencies]
```

If the author information that Cargo obtained from your environment is not correct, fix that in the file and save it again.

As you saw in Chapter 1, `cargo new` generates a “Hello, world!” program for you. Check out the *src/main.rs* file:

```
Filename: src/
main.rs
fn main() {
    println!("Hello, world!");
}
```

Now let’s compile this “Hello, world!” program and run it in the same step using the `cargo run` command:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
   Running `target/debug/guessing_game`
Hello, world!
```

The `run` command comes in handy when you need to rapidly iterate on a project, and this game is such a project: we want to quickly test each iteration before moving on to the next one.

Reopen the *src/main.rs* file. You’ll be writing all the code in this file.

Processing a Guess

The first part of the program will ask for user input, process that input, and check that the input is in the expected form. To start, we’ll allow the player to input a guess. Enter the code in Listing 2-1 into *src/main.rs*.

Filename: src/
main.rs

```
use std::io;

fn main() {
    println!("Guess the number!");

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin().read_line(&mut guess)
        .expect("Failed to read line");

    println!("You guessed: {}", guess);
}
```

Listing 2-1: Code to get a guess from the user and print it out

This code contains a lot of information, so let's go over it bit by bit. To obtain user input and then print the result as output, we need to import the `io` (input/output) library from the standard library (which is known as `std`):

```
use std::io;
```

By default, Rust imports only a few types into every program in the *prelude*. If a type you want to use isn't in the prelude, you have to import that type into your program explicitly with a `use` statement. Using the `std::io` library provides you with a number of useful `io`-related features, including the functionality to accept user input.

As you saw in Chapter 1, the `main` function is the entry point into the program:

```
fn main() {
```

The `fn` syntax declares a new function, the `()` indicate there are no arguments, and `{` starts the body of the function.

As you also learned in Chapter 1, `println!` is a macro that prints a string to the screen:

```
println!("Guess the number!");

println!("Please input your guess.");
```

This code is just printing a prompt stating what the game is and requesting input from the user.

Storing Values with Variables

Next, we'll create a place to store the user input, like this:

```
let mut guess = String::new();
```

Now the program is getting interesting! There's a lot going on in this little line. Notice that this is a `let` statement, which is used to create *variables*. Here's another example:

```
let foo = bar;
```

This line will create a new variable named `foo` and bind it to the value `bar`. In Rust, variables are immutable by default. The following example shows how to use `mut` before the variable name to make a variable mutable:

```
let foo = 5; // immutable
let mut bar = 5; // mutable
```

NOTE

The `//` syntax starts a comment that continues until the end of the line. Rust ignores everything in comments.

Now you know that `let mut guess` will introduce a mutable variable named `guess`. On the other side of the equal sign (`=`) is the value that `guess` is bound to, which is the result of calling `String::new`, a function that returns a new instance of a `String`. `String` is a string type provided by the standard library that is a growable, UTF-8 encoded bit of text.

The `::` syntax in the `::new` line indicates that `new` is an *associated function* of the `String` type. An associated function is implemented on a type, in this case `String`, rather than on a particular instance of a `String`. Some languages call this a *static method*.

This `new` function creates a new, empty `String`. You'll find a `new` function on many types, because it's a common name for a function that makes a new value of some kind.

To summarize, the `let mut guess = String::new();` line has created a mutable variable that is currently bound to a new, empty instance of a `String`. Whew!

Recall that we included the input/output functionality from the standard library with `use std::io;` on the first line of the program. Now we'll call an associated function, `stdin`, on `io`:

```
io::stdin().read_line(&mut guess)
    .expect("Failed to read line");
```

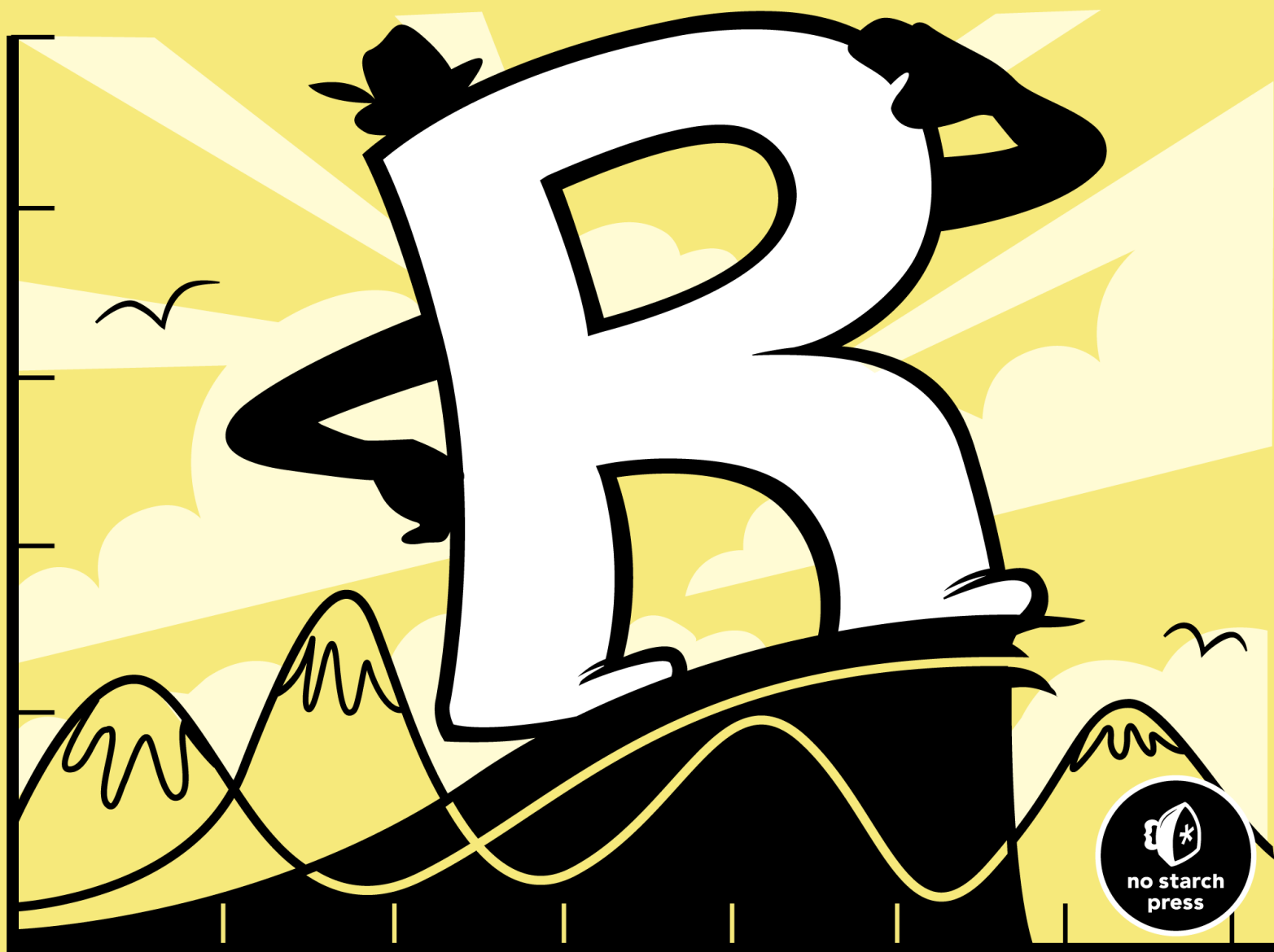
If we hadn't listed the `use std::io` line at the beginning of the program, we could have written this function call as `std::io::stdin`. The `stdin` function returns an instance of `std::io::Stdin`, which is a type that represents a handle to the standard input for your terminal.

The next part of the code, `.read_line(&mut guess)`, calls the `read_line` method on the standard input handle to get input from the user. We're also passing one argument to `read_line`: `&mut guess`.

THE BOOK OF R

A FIRST COURSE IN
PROGRAMMING AND STATISTICS

TILMAN M. DAVIES



2.3 Vectors

Often you'll want to perform the same calculations or comparisons upon multiple entities, for example if you're rescaling measurements in a data set. You could do this type of operation one entry at a time, though this is clearly not ideal, especially if you have a large number of items. R provides a far more efficient solution to this problem with *vectors*.

For the moment, to keep things simple, you'll continue to work with numeric entries only, though many of the utility functions discussed here may also be applied to structures containing non-numeric values. You'll start looking at these other kinds of data in Chapter 4.

2.3.1 Creating a Vector

The vector is the essential building block for handling multiple items in R. In a numeric sense, you can think of a vector as a collection of observations or measurements concerning a single variable, for example, the heights of 50 people or the number of coffees you drink daily. More complicated data structures may consist of several vectors. The function for creating a vector is the single letter `c`, with the desired entries in parentheses separated by commas.

```
R> myvec <- c(1,3,1,42)
R> myvec
[1] 1 3 1 42
```

Vector entries can be calculations or previously stored items (including vectors themselves).

```
R> foo <- 32.1
R> myvec2 <- c(3,-3,2,3.45,1e+03,64^0.5,2+(3-1.1)/9.44,foo)
R> myvec2
[1] 3.000000 -3.000000 2.000000 3.450000 1000.000000 8.000000
[7] 2.201271 32.100000
```

This code created a new vector assigned to the object `myvec2`. Some of the entries are defined as arithmetic expressions, and it's the result of the expression that's stored in the vector. The last element, `foo`, is an existing numeric object defined as `32.1`.

Let's look at another example.

```
R> myvec3 <- c(myvec,myvec2)
R> myvec3
[1] 1.000000 3.000000 1.000000 42.000000 3.000000 -3.000000
[7] 2.000000 3.450000 1000.000000 8.000000 2.201271 32.100000
```

This code creates and stores yet another vector, `myvec3`, which contains the entries of `myvec` and `myvec2` appended together in that order.

2.3.2 Sequences, Repetition, Sorting, and Lengths

Here I'll discuss some common and useful functions associated with R vectors: `seq`, `rep`, `sort`, and `length`.

Let's create an equally spaced sequence of increasing or decreasing numeric values. This is something you'll need often, for example when programming loops (see Chapter 10) or when plotting data points (see Chapter 7). The easiest way to create such a sequence, with numeric values separated by intervals of 1, is to use the colon operator.

```
R> 3:27
[1] 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
```

The example `3:27` should be read as “from 3 to 27 (by 1).” The result is a numeric vector just as if you had listed each number manually in parentheses with `c`. As always, you can also provide either a previously stored value or a (strictly parenthesized) calculation when using the colon operator:

```
R> foo <- 5.3
R> bar <- foo:(-47+1.5)
R> bar
[1] 5.3 4.3 3.3 2.3 1.3 0.3 -0.7 -1.7 -2.7 -3.7 -4.7
[12] -5.7 -6.7 -7.7 -8.7 -9.7 -10.7 -11.7 -12.7 -13.7 -14.7 -15.7
[23] -16.7 -17.7 -18.7 -19.7 -20.7 -21.7 -22.7 -23.7 -24.7 -25.7 -26.7
[34] -27.7 -28.7 -29.7 -30.7 -31.7 -32.7 -33.7 -34.7 -35.7 -36.7 -37.7
[45] -38.7 -39.7 -40.7 -41.7 -42.7 -43.7 -44.7
```

Sequences with `seq`

You can also use the `seq` command, which allows for more flexible creations of sequences. This ready-to-use function takes in a `from` value, a `to` value, and a `by` value, and it returns the corresponding sequence as a numeric vector.

```
R> seq(from=3,to=27,by=3)
[1] 3 6 9 12 15 18 21 24 27
```

This gives you a sequence with intervals of 3 rather than 1. Note that these kinds of sequences will always start at the `from` number but will not always include the `to` number, depending on what you are asking R to increase (or decrease) them by. For example, if you are increasing (or decreasing) by even numbers and your sequence ends in an odd number, the final number won't be included. Instead of providing a `by` value, however, you can specify a `length.out` value to produce a vector with that many numbers, evenly spaced between the `from` and `to` values.

```
R> seq(from=3,to=27,length.out=40)
[1] 3.000000 3.615385 4.230769 4.846154 5.461538 6.076923 6.692308
[8] 7.307692 7.923077 8.538462 9.153846 9.769231 10.384615 11.000000
[15] 11.615385 12.230769 12.846154 13.461538 14.076923 14.692308 15.307692
```

```
[22] 15.923077 16.538462 17.153846 17.769231 18.384615 19.000000 19.615385
[29] 20.230769 20.846154 21.461538 22.076923 22.692308 23.307692 23.923077
[36] 24.538462 25.153846 25.769231 26.384615 27.000000
```

By setting `length.out` to 40, you make the program print exactly 40 evenly spaced numbers from 3 to 27.

For decreasing sequences, the use of `by` must be negative. Here's an example:

```
R> foo <- 5.3
R> myseq <- seq(from=foo,to=(-47+1.5),by=-2.4)
R> myseq
 [1]  5.3  2.9  0.5 -1.9 -4.3 -6.7 -9.1 -11.5 -13.9 -16.3 -18.7 -21.1
[13] -23.5 -25.9 -28.3 -30.7 -33.1 -35.5 -37.9 -40.3 -42.7 -45.1
```

This code uses the previously stored object `foo` as the value for `from` and uses the parenthesized calculation `(-47+1.5)` as the `to` value. Given those values (that is, with `foo` being greater than `(-47+1.5)`), the sequence can progress only in negative steps; directly above, we set `by` to be `-2.4`. The use of `length.out` to create decreasing sequences, however, remains the same (it would make no sense to specify a “negative length”). For the same `from` and `to` values, you can create a decreasing sequence of length 5 easily, as shown here:

```
R> myseq2 <- seq(from=foo,to=(-47+1.5),length.out=5)
R> myseq2
 [1]  5.3 -7.4 -20.1 -32.8 -45.5
```

There are shorthand ways of calling these functions, which you'll learn about in Chapter 9, but in these early stages I'll stick with the explicit usage.

Repetition with `rep`

Sequences are extremely useful, but sometimes you may want simply to repeat a certain value. You do this using `rep`.

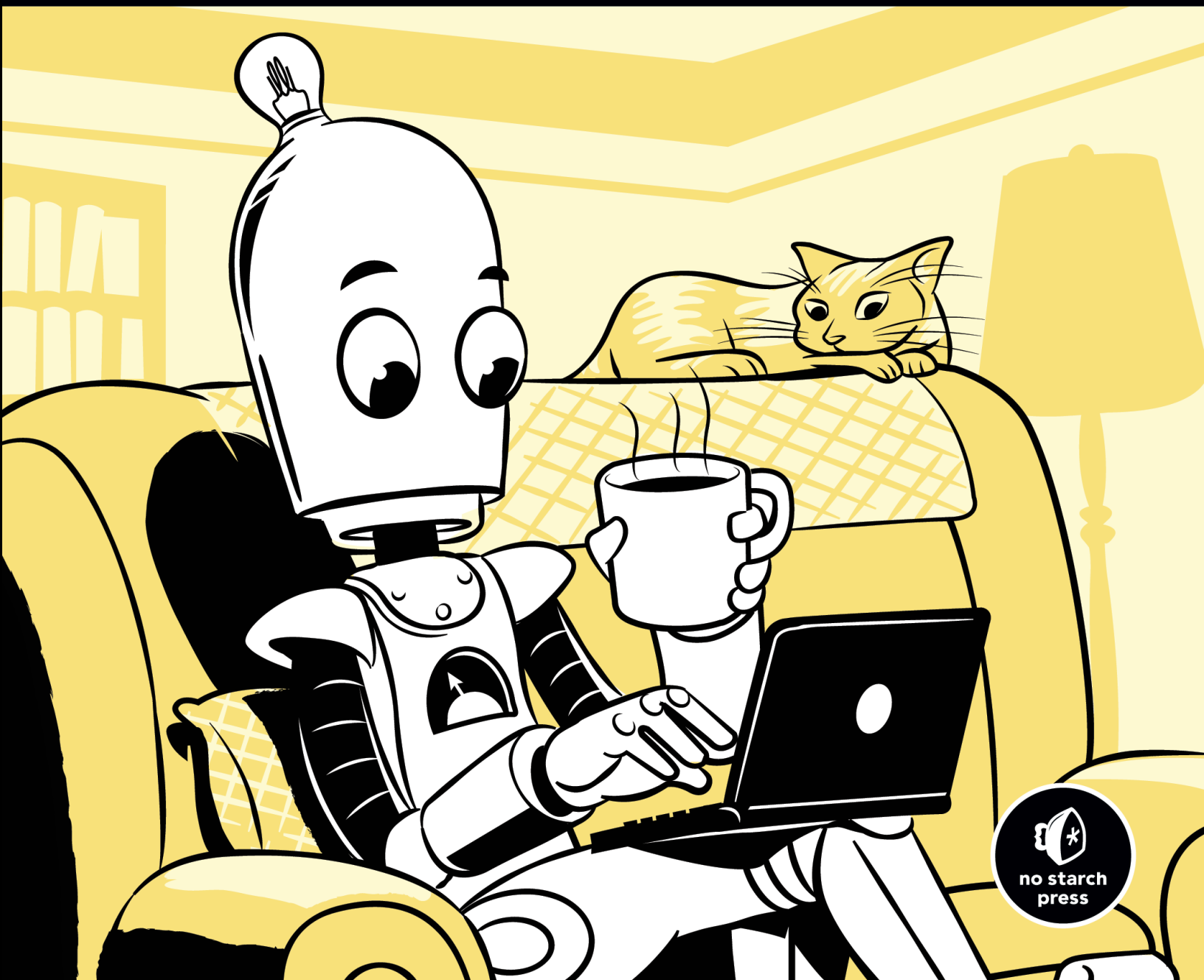
```
R> rep(x=1,times=4)
 [1] 1 1 1 1
R> rep(x=c(3,62,8.3),times=3)
 [1]  3.0 62.0  8.3  3.0 62.0  8.3  3.0 62.0  8.3
R> rep(x=c(3,62,8.3),each=2)
 [1]  3.0  3.0 62.0 62.0  8.3  8.3
R> rep(x=c(3,62,8.3),times=3,each=2)
 [1]  3.0  3.0 62.0 62.0  8.3  8.3  3.0  3.0 62.0 62.0  8.3  8.3  3.0  3.0 62.0
[16] 62.0  8.3  8.3
```

The `rep` function is given a single value or a vector of values as its argument `x`, as well as a value for the arguments `times` and `each`. The value for `times` provides the number of times to repeat `x`, and `each` provides the

LEARN JAVA THE EASY WAY

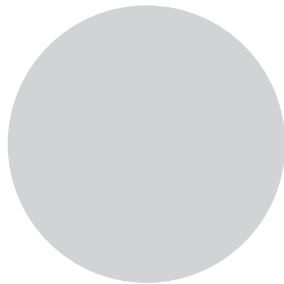
A HANDS-ON INTRODUCTION
TO PROGRAMMING

BRYSON PAYNE

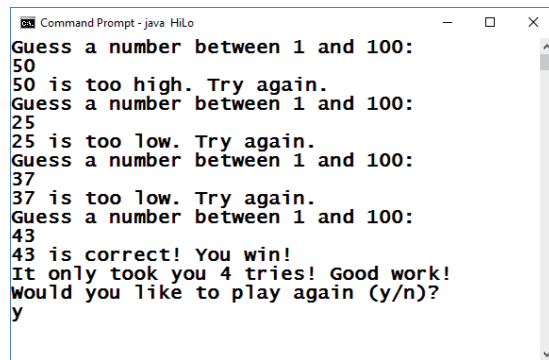


2

BUILD A HI-LO GUESSING GAME APP!



Let's begin by coding a fun, playable game in Java: the Hi-Lo guessing game. We'll program this game as a *command-line application*, which is just a fancy way of saying it's text-based (see Figure 2-1). When the program runs, the prompt will ask the user to guess a number between 1 and 100. Each time they guess, the program will tell them whether the guess is too high, too low, or correct.



```
Command Prompt - java HiLo
Guess a number between 1 and 100:
50
50 is too high. Try again.
Guess a number between 1 and 100:
25
25 is too low. Try again.
Guess a number between 1 and 100:
37
37 is too low. Try again.
Guess a number between 1 and 100:
43
43 is correct! You win!
It only took you 4 tries! Good work!
Would you like to play again (y/n)?
y
```

Figure 2-1: A text-based Hi-Lo guessing game

Now that you know how the game works, all you have to do is code the steps to play it. We'll start by mapping out the app at a high level, and then code a very simple version of the game. By starting out with a goal in mind and understanding how to play the game, you'll be able to pick up coding skills more easily, and you'll learn them with a purpose. You can also enjoy the game immediately after you finish coding it.

Planning the Game Step-by-Step

Let's think about all the steps we'll need to code in order to get the Hi-Lo guessing game to work. A basic version of the game will need to do the following:

1. Generate a random number between 1 and 100 for the user to guess.
2. Display a *prompt*, or a line of text, asking the user to guess a number in that range.
3. Accept the user's guess as input.
4. Compare the user's guess to the computer's number to see if it's too high, too low, or correct.
5. Display the results on the screen.
6. Prompt the user to guess another number until they guess correctly.
7. Ask the user if they'd like to play again.

We'll start with this basic structure. In Programming Challenge #2, you'll try adding an extra feature, to tell the user how many tries it took to guess the number correctly.

Creating a New Java Project

The first step in coding a new Java app in Eclipse is creating a project. On the menu bar in Eclipse, go to **File** ▶ **New** ▶ **Java Project** (or select **File** ▶ **New** ▶ **Project**, then **Java** ▶ **Java Project** in the New Project wizard). The Create a Java Project dialog should pop up, as shown in Figure 2-2.

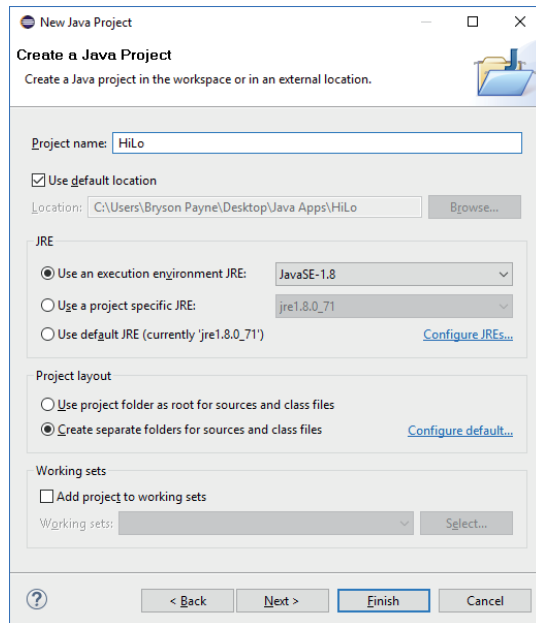


Figure 2-2: The New Java Project window for the Hi-Lo guessing game app

Type **HiLo** into the Project name text box. Note that uppercase and lowercase letters are important in Java, and we'll get in the habit of using uppercase letters to start all of our project, file, and class names, which is a common Java practice. Leave all the other settings unchanged, and click **Finish**. Depending on your version of Eclipse, you may be asked if you want to open the project using the Java Perspective. A *perspective* in Eclipse is a workspace set up for coding in a specific language. Click **Yes** to tell Eclipse you'd like the workspace set up for convenient coding in Java.

Creating the HiLo Class

Java is an *object-oriented programming language*. Object-oriented programming languages use *classes* to design reusable pieces of programming code. Classes are like templates that make it easier to create *objects*, or instances of that class. If you think of a class as a cookie cutter, objects are the cookies. And, just like a cookie cutter, classes are reusable, so once we've built a useful class, we can reuse it over and over again to create as many objects as we want.

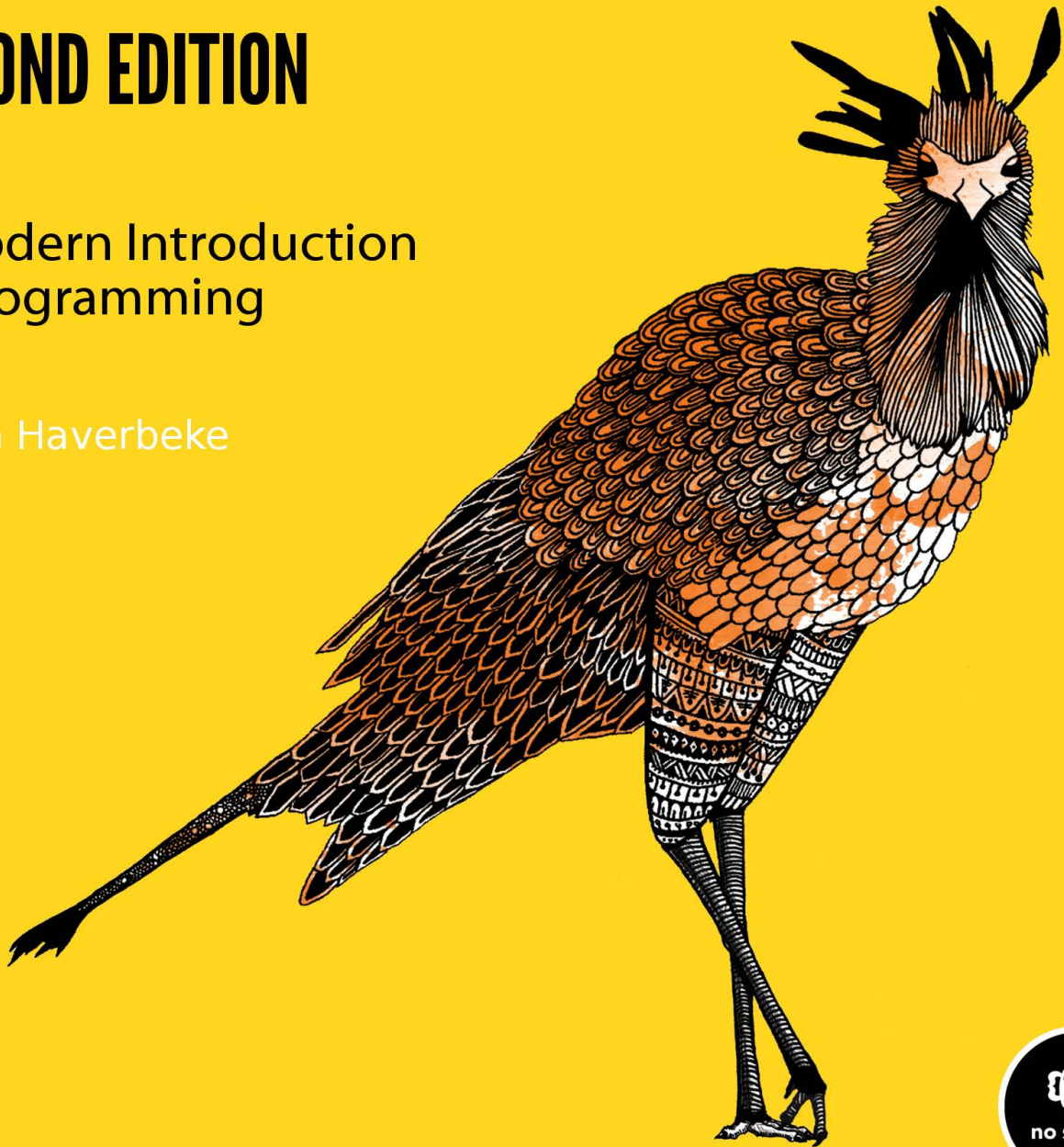
The Hi-Lo guessing game will have a single class file that creates a guessing game object with all the code needed to play the game. We'll call our new class **HiLo**. The capitalization matters, and naming the class **HiLo** follows several Java naming conventions. It's common practice to start all class names with an uppercase letter, so we use a capital **H** in **HiLo**. Also, there should be no spaces, hyphens, or special characters between words in a class name. Finally, we use *camel case* for class names with multiple words, beginning each new word with a capital letter, as in **HiLo**, **GuessingGame**, and **BubbleDrawApp**. The words look like they have humps in the middle, just like a camel.

ELOQUENT JAVASCRIPT

SECOND EDITION

A Modern Introduction
to Programming

Marijn Haverbeke



6

THE SECRET LIFE OF OBJECTS

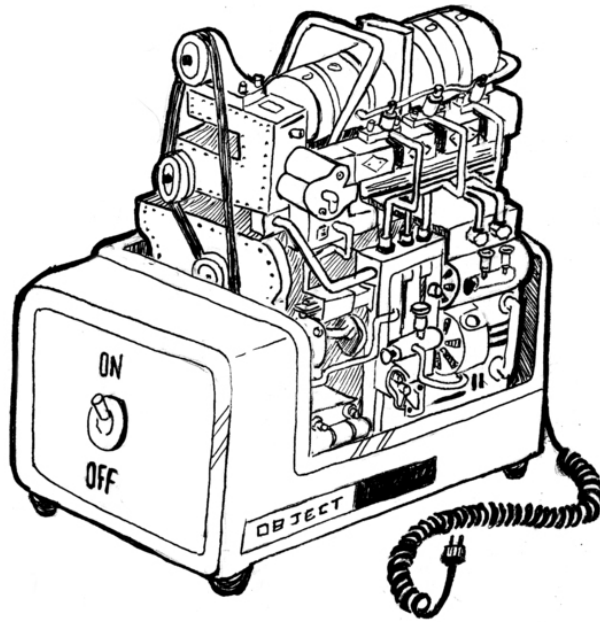
When a programmer says “object,” this is a loaded term. In my profession, objects are a way of life, the subject of holy wars, and a beloved buzzword that still hasn’t quite lost its power.

To an outsider, this is probably a little confusing. Let’s start with a brief history of objects as a programming construct.

History

This story, like most programming stories, starts with the problem of complexity. One philosophy is that complexity can be made manageable by separating it into small compartments that are isolated from each other. These compartments have ended up with the name *objects*.

An object is a hard shell that hides the gooey complexity inside it and instead offers us a few knobs and connectors (such as methods) that present an *interface* through which the object is to be used. The idea is that the interface is relatively simple and all the complex things going on *inside* the object can be ignored when working with it.



As an example, you can imagine an object that provides an interface to an area on your screen. It provides a way to draw shapes or text onto this area but hides all the details of how these shapes are converted to the actual pixels that make up the screen. You'd have a set of methods—for example, `drawCircle`—and those are the only things you need to know in order to use such an object.

These ideas were initially worked out in the 1970s and 1980s and, in the 1990s, were carried up by a huge wave of hype—the object-oriented programming revolution. Suddenly, there was a large tribe of people declaring that objects were the *right* way to program—and that anything that did not involve objects was outdated nonsense.

That kind of zealotry always produces a lot of impractical silliness, and there has been a sort of counter-revolution since then. In some circles, objects have a rather bad reputation nowadays.

I prefer to look at the issue from a practical, rather than ideological, angle. There are several useful concepts, most importantly that of *encapsulation* (distinguishing between internal complexity and external interface), that the object-oriented culture has popularized. These are worth studying.

This chapter describes JavaScript's rather eccentric take on objects and the way they relate to some classical object-oriented techniques.

Methods

Methods are simply properties that hold function values. This is a simple method:

```
var rabbit = {};  
rabbit.speak = function(line) {  
  console.log("The rabbit says '" + line + "'");  
};
```



```
rabbit.speak("I'm alive.");  
// > The rabbit says 'I'm alive.'
```

Usually a method needs to do something with the object it was called on. When a function is called as a method—looked up as a property and immediately called, as in `object.method()`—the special variable `this` in its body will point to the object that it was called on.

```
function speak(line) {  
  console.log("The " + this.type + " rabbit says '" +  
    line + "'");  
}  
var whiteRabbit = {type: "white", speak: speak};  
var fatRabbit = {type: "fat", speak: speak};  
  
whiteRabbit.speak("Oh my ears and whiskers, " +  
  "how late it's getting!");  
// > The white rabbit says 'Oh my ears and whiskers, how  
//   late it's getting!'  
fatRabbit.speak("I could sure use a carrot right now.");  
// > The fat rabbit says 'I could sure use a carrot  
//   right now.'
```

The code uses the `this` keyword to output the type of rabbit that is speaking. Recall that the `apply` and `bind` methods both take a first argument that can be used to simulate method calls. This first argument is in fact used to give a value to `this`.

There is a method similar to `apply`, called `call`. It also calls the function it is a method of but takes its arguments normally, rather than as an array. Like `apply` and `bind`, `call` can be passed a specific `this` value.

```
speak.apply(fatRabbit, ["Burp!"]);  
// > The fat rabbit says 'Burp!'  
speak.call({type: "old"}, "Oh my.");  
// > The old rabbit says 'Oh my.'
```

Prototypes

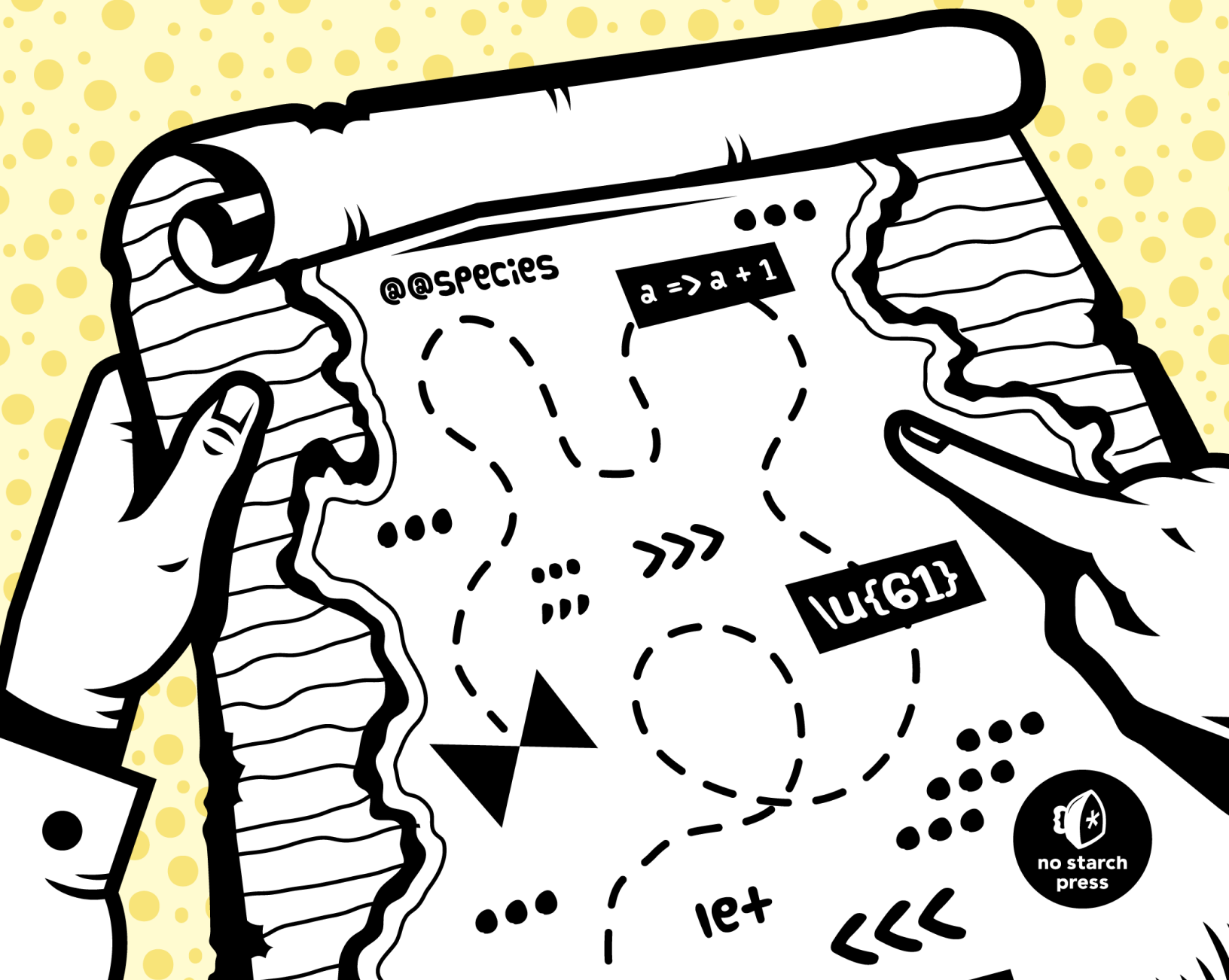
Watch closely.

```
var empty = {};  
console.log(empty.toString());  
// > function toString(){...}  
console.log(empty.toString());  
// > [object Object]
```

UNDERSTANDING ECMAScript 6

THE DEFINITIVE GUIDE FOR
JAVASCRIPT DEVELOPERS

NICHOLAS C. ZAKAS



Class-Like Structures in ECMAScript 5

As mentioned, in ECMAScript 5 and earlier, JavaScript had no classes. The closest equivalent to a class was creating a constructor and then assigning methods to the constructor's prototype, an approach typically called creating a custom type. For example:

```
function PersonType(name) {
    this.name = name;
}

PersonType.prototype.sayName = function() {
    console.log(this.name);
};

var person = new PersonType("Nicholas");
person.sayName(); // outputs "Nicholas"

console.log(person instanceof PersonType); // true
console.log(person instanceof Object);     // true
```

In this code, `PersonType` is a constructor function that creates a single property called `name`. The `sayName()` method is assigned to the prototype so the same function is shared by all instances of the `PersonType` object. Then, a new instance of `PersonType` is created via the `new` operator. The resulting `person` object is considered an instance of `PersonType` and of `Object` through prototypal inheritance.

This basic pattern underlies many of the class-mimicking JavaScript libraries, and that's where ECMAScript 6 classes start.

Class Declarations

The simplest class form in ECMAScript 6 is the class declaration, which looks similar to classes in other languages.

A Basic Class Declaration

Class declarations begin with the `class` keyword followed by the name of the class. The rest of the syntax looks similar to concise methods in object literals but doesn't require commas between the elements of the class. Here's a simple class declaration:

```
class PersonClass {

    // equivalent of the PersonType constructor
    constructor(name) {
        this.name = name;
    }

    // equivalent of PersonType.prototype.sayName
    sayName() {
```

```

        console.log(this.name);
    }
}

let person = new PersonClass("Nicholas");
person.sayName(); // outputs "Nicholas"

console.log(person instanceof PersonClass); // true
console.log(person instanceof Object); // true

console.log(typeof PersonClass); // "function"
console.log(typeof PersonClass.prototype.sayName); // "function"

```

The class declaration for `PersonClass` behaves similarly to `PersonType` in the previous example. But instead of defining a function as the constructor, class declarations allow you to define the constructor directly inside the class using the special constructor method name. Because class methods use the concise syntax, there's no need to use the `function` keyword. All other method names have no special meaning, so you can add as many methods as you want.

Own properties, properties that occur on the instance rather than the prototype, can only be created inside a class constructor or method. In this example, `name` is an own property. I recommend creating all possible own properties inside the constructor function so a single place in the class is responsible for all of them.

Interestingly, class declarations are just syntactic sugar on top of the existing custom type declarations. The `PersonClass` declaration actually creates a function that has the behavior of the constructor method, which is why `typeof PersonClass` gives "function" as the result. The `sayName()` method also ends up as a method on `PersonClass.prototype` in this example, similar to the relationship between `sayName()` and `PersonType.prototype` in the previous example. These similarities allow you to mix custom types and classes without worrying too much about which you're using.

NOTE

Class prototypes, such as `PersonClass.prototype` in the preceding example, are read-only. That means you cannot assign a new value to the prototype like you can with functions.

Why Use the Class Syntax?

Despite the similarities between classes and custom types, you need to keep some important differences in mind:

- Class declarations, unlike function declarations, are not hoisted. Class declarations act like `let` declarations, so they exist in the temporal dead zone until execution reaches the declaration.
- All code inside class declarations runs in strict mode automatically. There's no way to opt out of strict mode inside classes.
- All methods are nonenumerable. This is a significant change from custom types, where you need to use `Object.defineProperty()` to make a method nonenumerable.

- All methods lack an internal `[[Construct]]` method and will throw an error if you try to call them with `new`.
- Calling the class constructor without `new` throws an error.
- Attempting to overwrite the class name within a class method throws an error.

With all of these differences in mind, the `PersonClass` declaration in the previous example is directly equivalent to the following code, which doesn't use the class syntax:

```
// direct equivalent of PersonClass
let PersonType2 = (function() {

    "use strict";

    const PersonType2 = function(name) {

        // make sure the function was called with new
        if (typeof new.target === "undefined") {
            throw new Error("Constructor must be called with new.");
        }

        this.name = name;
    }

    Object.defineProperty(PersonType2.prototype, "sayName", {
        value: function() {

            // make sure the method wasn't called with new
            if (typeof new.target !== "undefined") {
                throw new Error("Method cannot be called with new.");
            }

            console.log(this.name);
        },
        enumerable: false,
        writable: true,
        configurable: true
    });

    return PersonType2;
})();
```

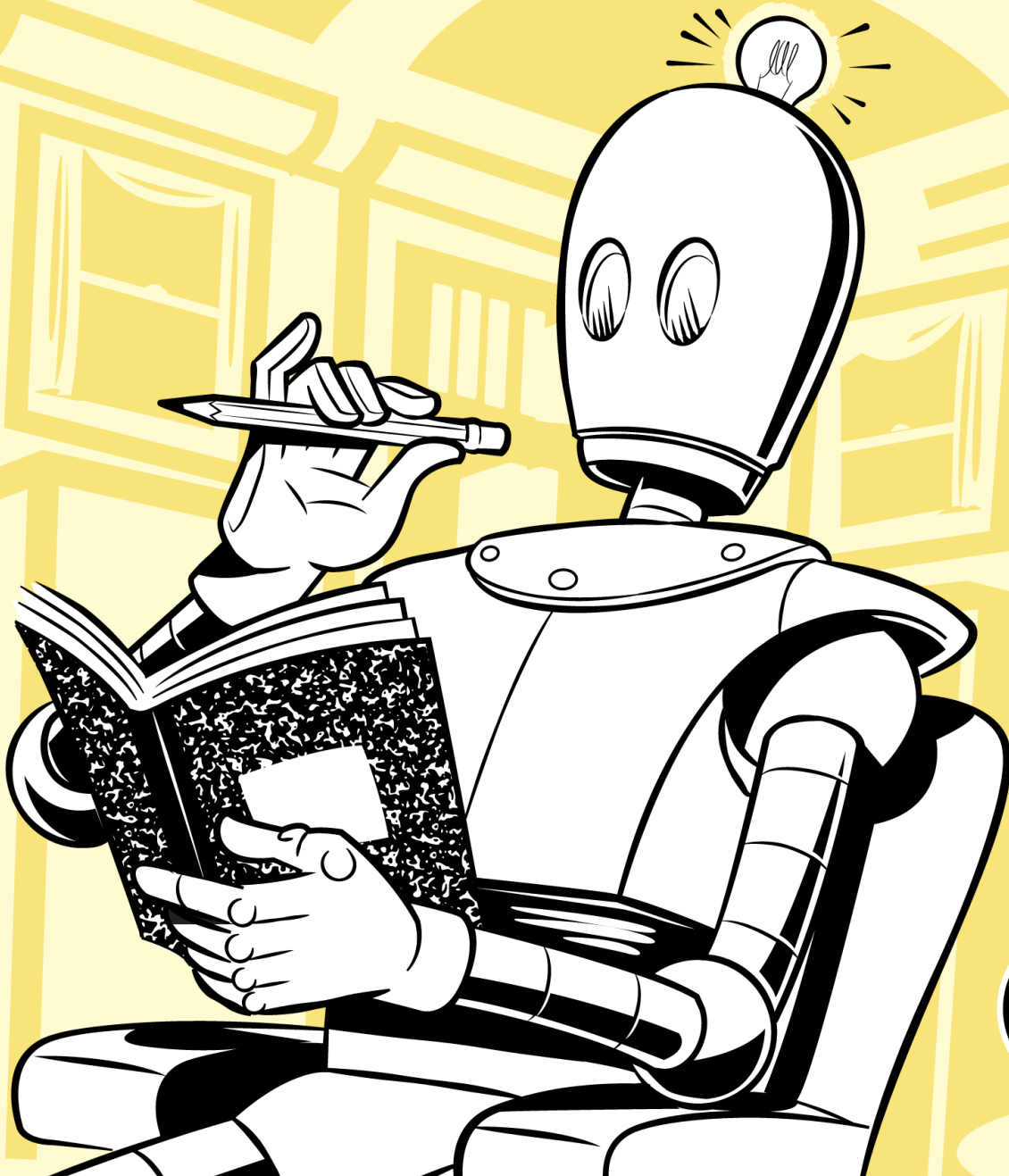
First, notice that there are two `PersonType2` declarations: a `let` declaration in the outer scope and a `const` declaration inside the immediately invoked function expression (IIFE)—this is how class methods are forbidden from overwriting the class name while code outside the class is allowed to do so. The constructor function checks `new.target` to ensure that it's being called with `new`; if not, an error is thrown. Next, the `sayName()` method is defined as nonenumerable, and the method checks `new.target` to ensure that it wasn't called with `new`. The final step returns the constructor function.

2ND
EDITION

WICKED COOL SHELL SCRIPTS

101 SCRIPTS FOR LINUX, OS X,
AND UNIX SYSTEMS

DAVE TAYLOR AND BRANDON PERRY



```
services.amazon.com
www.6pm.com
www.abebooks.com
www.acx.com
www.afterschool.com
www.alexa.com
```

Amazon doesn't tend to point outside its own site, but there are some partner links that creep onto the home page. Other sites are different, of course.

What if we split the links on the Amazon page into relative and absolute links?

```
$ getlinks -a http://www.amazon.com/ | wc -l
51
$ getlinks -r http://www.amazon.com/ | wc -l
222
```

As you might have expected, Amazon has four times more relative links pointing inside its own site than it has absolute links, which would lead to a different website. Gotta keep those customers on your own page!

Hacking the Script

You can see where `getlinks` could be quite useful as a site analysis tool. For a way to enhance the script, stay tuned: Script #69 on page 217 complements this script nicely, allowing us to quickly check that all hypertext references on a site are valid.

#55 Getting GitHub User Information

GitHub has grown to be a huge boon to the open source industry and open collaboration across the world. Many system administrators and developers have visited GitHub to pull down some source code or report an issue to an open source project. Because GitHub is essentially a social platform for developers, getting to know a user's basic information quickly can be useful. The script in Listing 7-6 prints some information about a given GitHub user, and it gives a good introduction to the very powerful GitHub API.

The Code

```
#!/bin/bash
# githubuser--Given a GitHub username, pulls information about the user

if [ $# -ne 1 ]; then
    echo "Usage: $0 <username>"
    exit 1
fi
```

```

# The -s silences curl's normally verbose output.
❶ curl -s "https://api.github.com/users/$1" | \
    awk -F'"' '
        /\\"name\\":/ {
            print $4 " is the name of the GitHub user."
        }
        /\\"followers\\":/{
            split($3, a, " ")
            sub(/,/,"", a[2])
            print "They have "a[2]" followers."
        }
        /\\"following\\":/{
            split($3, a, " ")
            sub(/,/,"", a[2])
            print "They are following "a[2]" other users."
        }
        /\\"created_at\\":/{
            print "Their account was created on \"$4\"."
        }
    '

exit 0

```

Listing 7-6: The githubuser script

How It Works

Admittedly, this is almost more of an `awk` script than a `bash` script, but sometimes you need the extra horsepower `awk` provides for parsing (the GitHub API returns JSON). We use `curl` to ask GitHub for the user ❶, given as the argument of the script, and pipe the JSON to `awk`. With `awk`, we specify a field separator of the double quotes character, as this will make parsing the JSON much simpler. Then we match the JSON with a handful of regular expressions in the `awk` script and print the results in a user-friendly way.

Running the Script

The script accepts a single argument: the user to look up on GitHub. If the username provided doesn't exist, nothing will be printed.

The Results

When passed a valid username, the script should print a user-friendly summary of the GitHub user, as Listing 7-7 shows.

```

$ githubuser brandonprry
Brandon Perry is the name of the GitHub user.
They have 67 followers.
They are following 0 other users.
Their account was created on 2010-11-16T02:06:41Z.

```

Listing 7-7: Running the githubuser script

Hacking the Script

This script has a lot of potential due to the information that can be retrieved from the GitHub API. In this script, we are only printing four values from the JSON returned. Generating a “résumé” for a given user based on the information provided by the API, like those provided by many web services, is just one possibility.

#56 ZIP Code Lookup

To demonstrate a different technique for scraping the web, this time using `curl`, let’s create a simple ZIP code lookup tool. Give the script in Listing 7-8 a ZIP code, and it’ll report the city and state the code belongs to. Easy enough.

Your first instinct might be to use the official US Postal Service website, but we’re going to tap into a different site, <http://city-data.com/>, which configures each ZIP code as its own web page so information is far easier to extract.

The Code

```
#!/bin/bash

# zipcode--Given a ZIP code, identifies the city and state. Use city-data.com,
#   which has every ZIP code configured as its own web page.

baseURL="http://www.city-data.com/zips"

/bin/echo -n "ZIP code $1 is in "

curl -s -dump "$baseURL/$1.html" | \
  grep -i '<title>' | \
  cut -d\ ( -f2 | cut -d\ ) -f1

exit 0
```

Listing 7-8: The zipcode script

How It Works

The URLs for ZIP code information pages on <http://city-data.com/> are structured consistently, with the ZIP code itself as the final part of the URL.

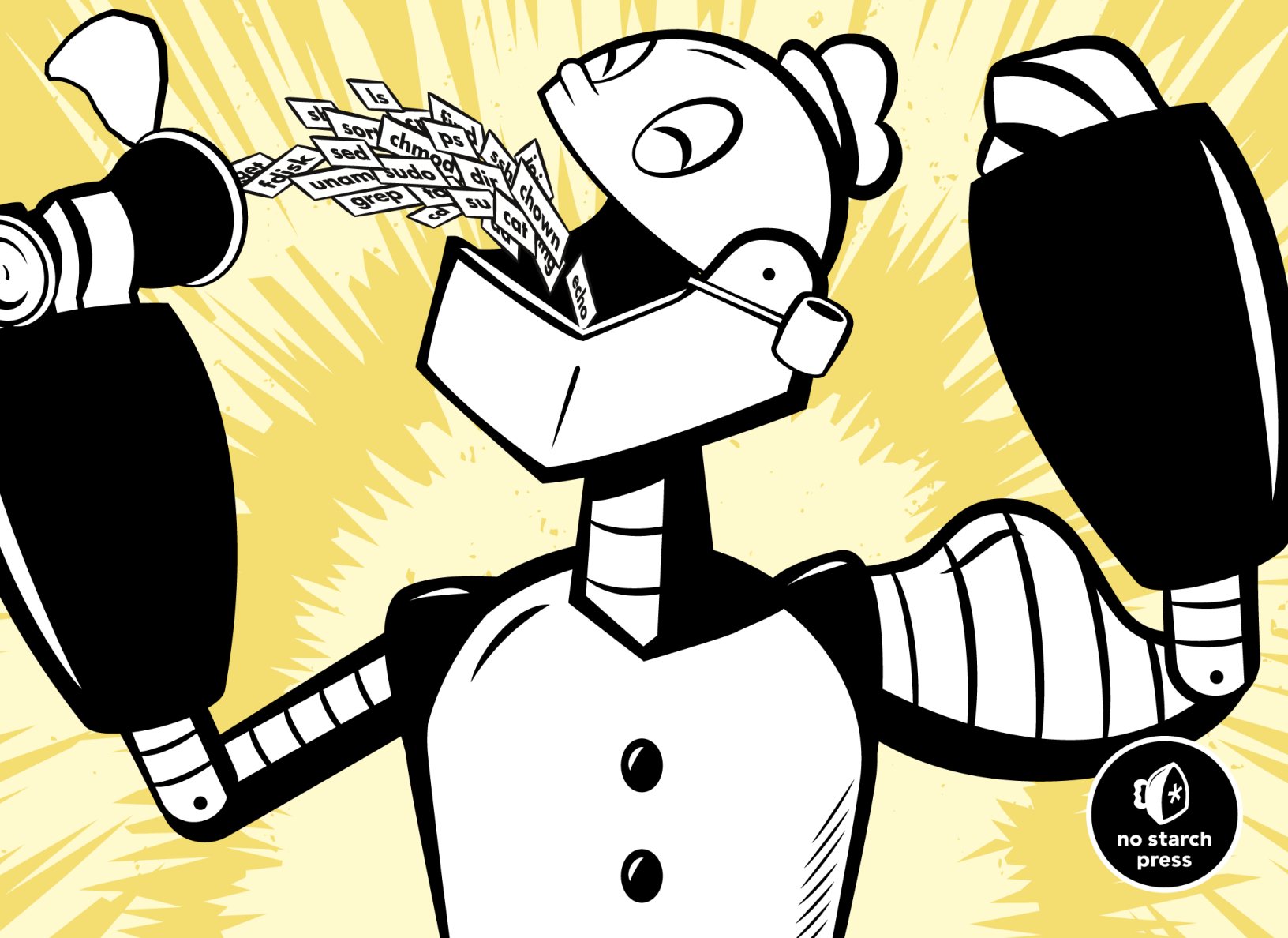
```
http://www.city-data.com/zips/80304.html
```

This consistency makes it quite easy to create an appropriate URL for a given ZIP code on the fly. The resultant page has the city name in the title, conveniently denoted by open and close parentheses, as follows.

THE LINUX COMMAND LINE

A COMPLETE INTRODUCTION

WILLIAM E. SHOTTS, JR.



6

REDIRECTION

In this lesson we are going to unleash what may be the coolest feature of the command line: *I/O redirection*. The *I/O* stands for *input/output*, and with this facility you can redirect the input and output of commands to and from files, as well as connect multiple commands to make powerful command *pipelines*. To show off this facility, we will introduce the following commands:

- `cat`—Concatenate files.
- `sort`—Sort lines of text.
- `uniq`—Report or omit repeated lines.
- `wc`—Print newline, word, and byte counts for each file.
- `grep`—Print lines matching a pattern.
- `head`—Output the first part of a file.
- `tail`—Output the last part of a file.
- `tee`—Read from standard input and write to standard output and files.

Standard Input, Output, and Error

Many of the programs that we have used so far produce output of some kind. This output often consists of two types. First, we have the program's results; that is, the data the program is designed to produce. Second, we have status and error messages that tell us how the program is getting along. If we look at a command like `ls`, we can see that it displays its results and its error messages on the screen.

Keeping with the Unix theme of “everything is a file,” programs such as `ls` actually send their results to a special file called *standard output* (often expressed as *stdout*) and their status messages to another file called *standard error* (*stderr*). By default, both standard output and standard error are linked to the screen and not saved into a disk file.

In addition, many programs take input from a facility called *standard input* (*stdin*), which is, by default, attached to the keyboard.

I/O redirection allows us to change where output goes and where input comes from. Normally, output goes to the screen and input comes from the keyboard, but with I/O redirection we can change that.

Redirecting Standard Output

I/O redirection allows us to redefine where standard output goes. To redirect standard output to another file instead of the screen, we use the `>` redirection operator followed by the name of the file. Why would we want to do this? It's often useful to store the output of a command in a file. For example, we could tell the shell to send the output of the `ls` command to the file *ls-output.txt* instead of the screen:

```
[me@linuxbox ~]$ ls -l /usr/bin > ls-output.txt
```

Here, we created a long listing of the */usr/bin* directory and sent the results to the file *ls-output.txt*. Let's examine the redirected output of the command:

```
[me@linuxbox ~]$ ls -l ls-output.txt
-rw-rw-r-- 1 me  me  167878 2012-02-01 15:07 ls-output.txt
```

Good—a nice, large, text file. If we look at the file with `less`, we will see that the file *ls-output.txt* does indeed contain the results from our `ls` command:

```
[me@linuxbox ~]$ less ls-output.txt
```

Now, let's repeat our redirection test but this time with a twist. We'll change the name of the directory to one that does not exist:

```
[me@linuxbox ~]$ ls -l /bin/usr > ls-output.txt
ls: cannot access /bin/usr: No such file or directory
```

We received an error message. This makes sense because we specified the nonexistent directory `/bin/usr`, but why was the error message displayed on the screen rather than being redirected to the file `ls-output.txt`? The answer is that the `ls` program does not send its error messages to standard output. Instead, like most well-written Unix programs, it sends its error messages to standard error. Since we redirected only standard output and not standard error, the error message was still sent to the screen. We'll see how to redirect standard error in just a minute, but first, let's look at what happened to our output file:

```
[me@linuxbox ~]$ ls -l ls-output.txt
-rw-rw-r-- 1 me  me  0 2012-02-01 15:08 ls-output.txt
```

The file now has zero length! This is because, when we redirect output with the `>` redirection operator, the destination file is always rewritten from the beginning. Since our `ls` command generated no results and only an error message, the redirection operation started to rewrite the file and then stopped because of the error, resulting in its truncation. In fact, if we ever need to actually truncate a file (or create a new, empty file) we can use a trick like this:

```
[me@linuxbox ~]$ > ls-output.txt
```

Simply using the redirection operator with no command preceding it will truncate an existing file or create a new, empty file.

So, how can we append redirected output to a file instead of overwriting the file from the beginning? For that, we use the `>>` redirection operator, like so:

```
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
```

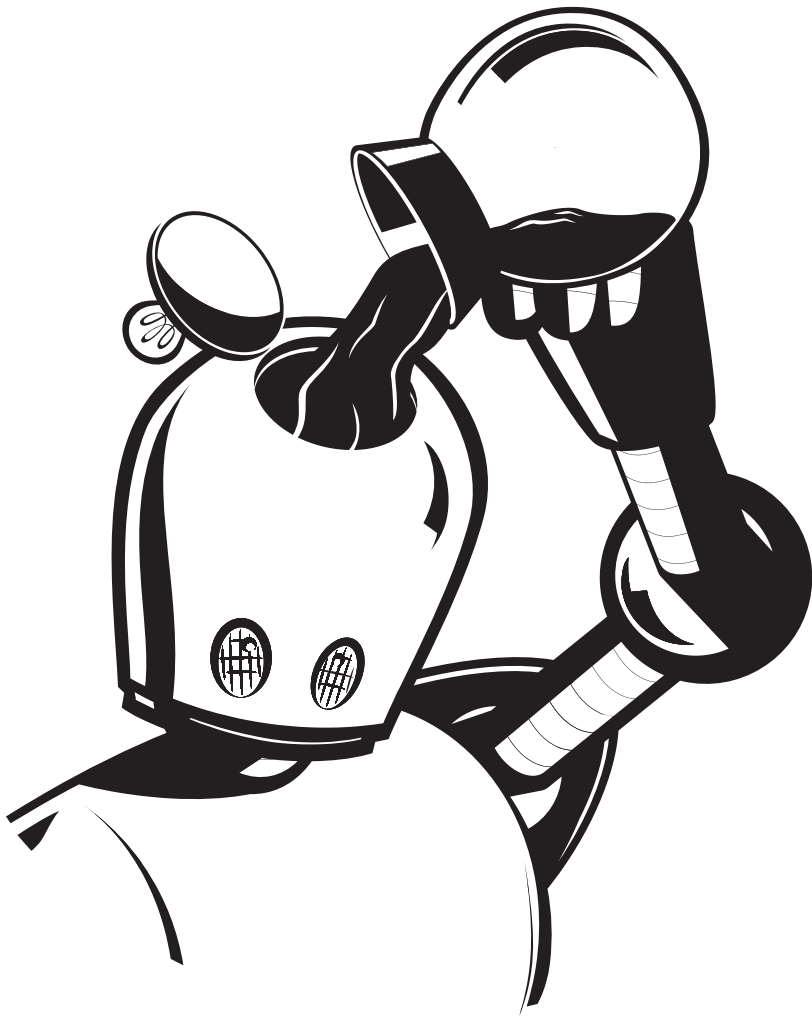
Using the `>>` operator will result in the output being appended to the file. If the file does not already exist, it is created just as though the `>` operator had been used. Let's put it to the test:

```
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
[me@linuxbox ~]$ ls -l ls-output.txt
-rw-rw-r-- 1 me  me  503634 2012-02-01 15:45 ls-output.txt
```

We repeated the command three times, resulting in an output file three times as large.

Redirecting Standard Error

Redirecting standard error lacks the ease of using a dedicated redirection operator. To redirect standard error we must refer to its *file descriptor*. A program can produce output on any of several numbered file streams. While



Founded in 1994, No Starch Press is one of the few remaining independent technical book publishers. We publish the finest in geek entertainment—unique books on technology, with a focus on open source, security, hacking, programming, alternative operating systems, and LEGO. Our titles have personality, our authors are passionate, and our books tackle topics that people care about.

**VISIT WWW.NOSTARCH.COM
FOR A COMPLETE CATALOG.**

NO STARCH PRESS 2017 CATALOG FOR HUMBLE BOOK BUNDLE: PYTHON. COPYRIGHT © 2017 NO STARCH PRESS, INC. ALL RIGHTS RESERVED. LEARN TO PROGRAM WITH MINECRAFT® © CRAIG RICHARDSON. THINK LIKE A PROGRAMMER, PYTHON EDITION © V. ANTON SPRAUL. THE RUST PROGRAMMING LANGUAGE © STEVE KLABNIK AND CAROL NICHOLS, WITH CONTRIBUTIONS FROM THE RUST COMMUNITY. THE BOOK OF R © TILMAN M. DAVIES. LEARN JAVA THE EASY WAY © BRYSON PAYNE. ELOQUENT JAVASCRIPT, 2ND EDITION © MARIJN HAVERBEKE. UNDERSTANDING ECMASCRIPT 6 © NICHOLAS C. ZAKAS. WICKED COOL SHELL SCRIPTS, 2ND EDITION © DAVE TAYLOR AND BRANDON PERRY. THE LINUX COMMAND LINE © WILLIAM E. SHOTTS, JR. NO STARCH PRESS AND THE NO STARCH PRESS LOGO ARE REGISTERED TRADEMARKS OF NO STARCH PRESS, INC. NO PART OF THIS WORK MAY BE REPRODUCED OR TRANSMITTED IN ANY FORM OR BY ANY MEANS, ELECTRONIC OR MECHANICAL, INCLUDING PHOTOCOPYING, RECORDING, OR BY ANY INFORMATION STORAGE OR RETRIEVAL SYSTEM, WITHOUT THE PRIOR WRITTEN PERMISSION OF NO STARCH PRESS, INC.

