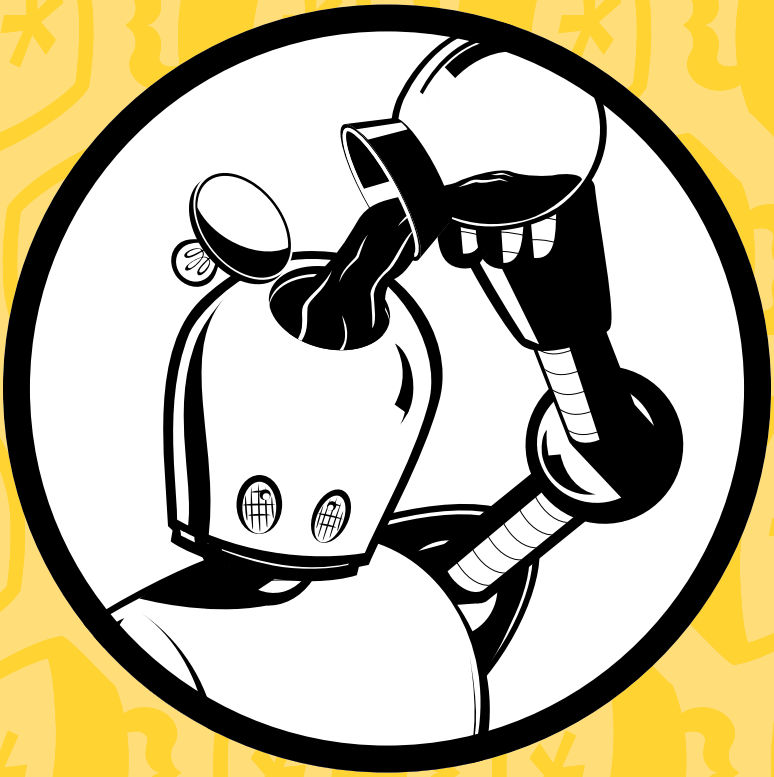
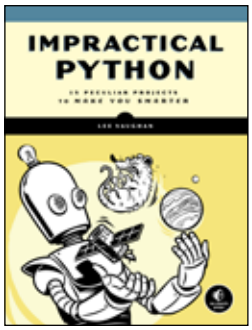


MORE FROM

NO STARCH PRESS

COMING SOON FROM NO STARCH PRESS!



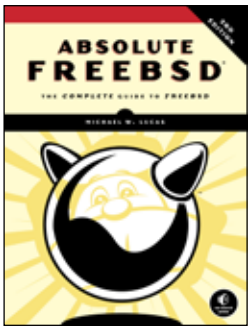
IMPRACTICAL PYTHON

15 Peculiar Projects to Make You Smarter

by LEE VAUGHAN
SUMMER 2018, 504 PP., $29.95
ISBN 978-1-59327-890-8

C++ CRASH COURSE

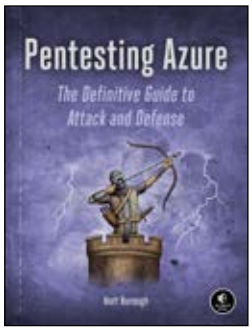
by JOSH LOSPINOSO
SUMMER 2018, 524 PP., $49.95
ISBN 978-1-59327-888-5



ABSOLUTE FREEBSD, 3RD EDITION

The Complete Guide to FreeBSD

by MICHAEL W. LUCAS
SUMMER 2018, 832 PP., $59.95
ISBN 978-1-59327-892-2



PENTESTING AZURE

The Definitive Guide to Attack and Defense

by MATT BURROUGH
SPRING 2018, 200 PP., $39.95
ISBN 978-1-59327-863-2

ROOTKITS AND BOOTKITS

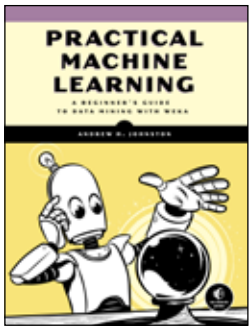
Reversing Modern Malware and Next Generation Threats

by ALEX MATROSOV, EUGENE RODIONOV, AND SERGEY BRATUS
SUMMER 2018, 504 PP., $49.95
ISBN 978-1-59327-716-1

MALWARE DATA SCIENCE

Attack Detection and Attribution

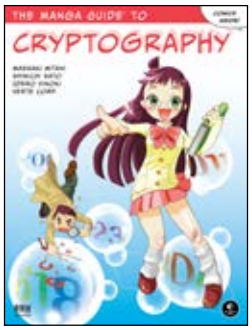
by JOSHUA SAXE WITH HILLARY SANDERS
SUMMER 2018, 400 PP., $49.95
ISBN 978-1-59327-859-5



PRACTICAL MACHINE LEARNING

A Beginner's Guide to Data Mining with WEKA

by ANDREW H. JOHNSTON
SUMMER 2018, 232 PP., $39.95
ISBN 978-1-59327-876-2



THE MANGA GUIDE TO CRYPTOGRAPHY

by MASAAKI MITANI, SHINICHI SATO, IDERO HINOKI, AND VERTE CORP.
SUMMER 2018, 240 PP., $24.95
ISBN 978-1-59327-742-0

REAL-WORLD BUG HUNTING

A Field Guide to Web Hacking

by PETER YAWORSKI
SUMMER 2018, 256 PP., $39.95
ISBN 978-1-59327-861-8

READ SAMPLE CHAPTERS FROM THESE NO STARCH BOOKS!

THE MANGA GUIDE TO MICROPROCESSORS

MICHIO SHIBUYA, TAKASHI TONAGI, AND OFFICE SAWA

•

THE RUST PROGRAMMING LANGUAGE

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

•

LEARN JAVA THE EASY WAY

BRYSON PAYNE

•

AUTOMATE THE MINECRAFT STUFF

AL SWEIGART

•

CRACKING CODES WITH PYTHON

AL SWEIGART

•

PRACTICAL SQL

ANTHONY DEBARROS

•

SERIOUS CRYPTOGRAPHY

JEAN-PHILIPPE AUMASSON

•

PRACTICAL PACKET ANALYSIS, 3RD EDITION

CHRIS SANDERS

•

CODING IPHONE APPS FOR KIDS

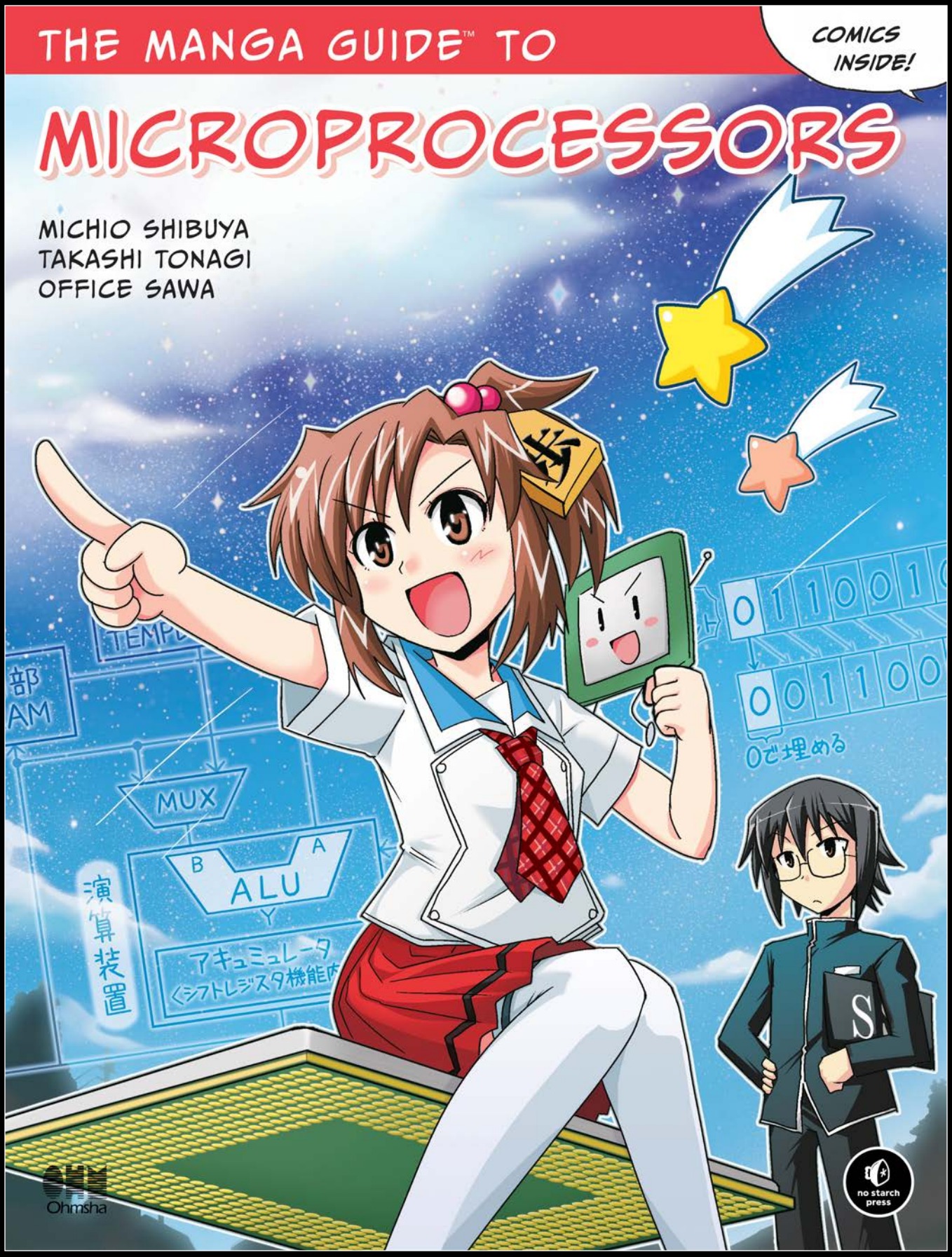
GLORIA WINQUIST AND MATT MCCARTHY

THE MANGA GUIDE™ TO

COMICS INSIDE!

MICROPROCESSORS

MICHIO SHIBUYA
TAKASHI TONAGI
OFFICE SAWA



Ohmsha
no starch press

1

WHAT DOES THE CPU DO?

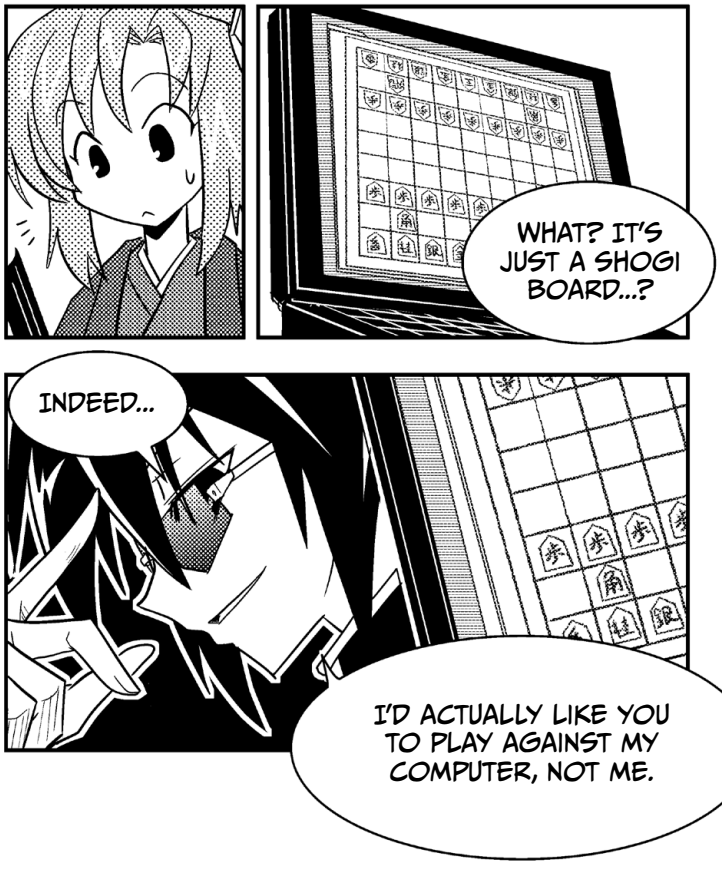

BANG!
BANG!
CULTURAL FESTIVAL
やき
JOIN US IN THE
SHOGI*CLUB!
COME TODAY AND
TRY YOUR STRENGTH
AGAINST NATIONAL
CHAMPION AYUMI
* A JAPANESE BOARD GAME SIMILAR TO CHESS
WOW,
AYUMI...
STRONG AS
EVER, I SEE.
PHEW!
CLACK
HMM, TWENTY WINS
IN A ROW...


GOOD JOB!
RATTLE
I BET OUR BOOTH WILL BE THE MOST POPULAR ONE AT THE FESTIVAL!
AND IT'S ALL THANKS TO YOU, AYUMI! MARRY MEEEE!
HUG!
I'M A GOOD SPORT, BUT THAT MIGHT BE A BIT MUCH...
THAT'S MY GIRL! NOW WEAR THIS FOR THE NEXT GAME!
CROWD PULLER
TAKE A BREAK ALREADY. YOU'RE RIDICULOUS.
WELL, YOU'RE UNBEATABLE WHETHER I'M HERE CHEERING OR NOT.
SEE YOU AROUND!
...
AYUMI
UNBEATABLE, HUH?


SHE'S RIGHT! I AM STRONG!
OR ACTUALLY, EVERYONE ELSE IS SO WEAK IT'S BORING ME TO TEARS....
KNOCK KNOCK
EXCUSE ME.
OH, SURE!
I'D LOVE TO!
DO YOU... HAVE TIME FOR A GAME?
I SEE... WELL THEN...


FUHAHAHAHAHAHA!
BEHOLD! UNLEASHED FROM
THE CHASM OF DARKNESS!
SS
Shooting Star
THUNK
THE SHOOTING STAR!!

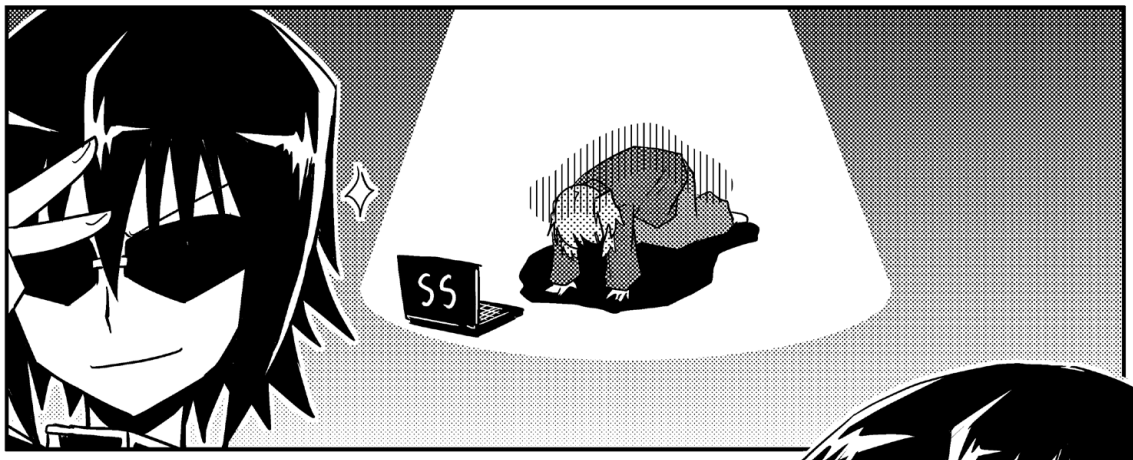

WHAT'S HE SO
EXCITED ABOUT?!
IT'S JUST A BLACK
COMPUTER!!
JUST MY LUCK...
A REAL WEIRDO.

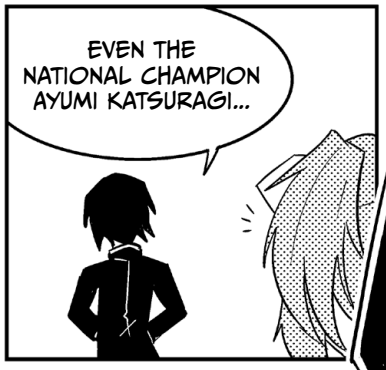

WHAT? IT'S
JUST A SHOGI
BOARD...?
INDEED...
I'D ACTUALLY LIKE YOU
TO PLAY AGAINST MY
COMPUTER, NOT ME.

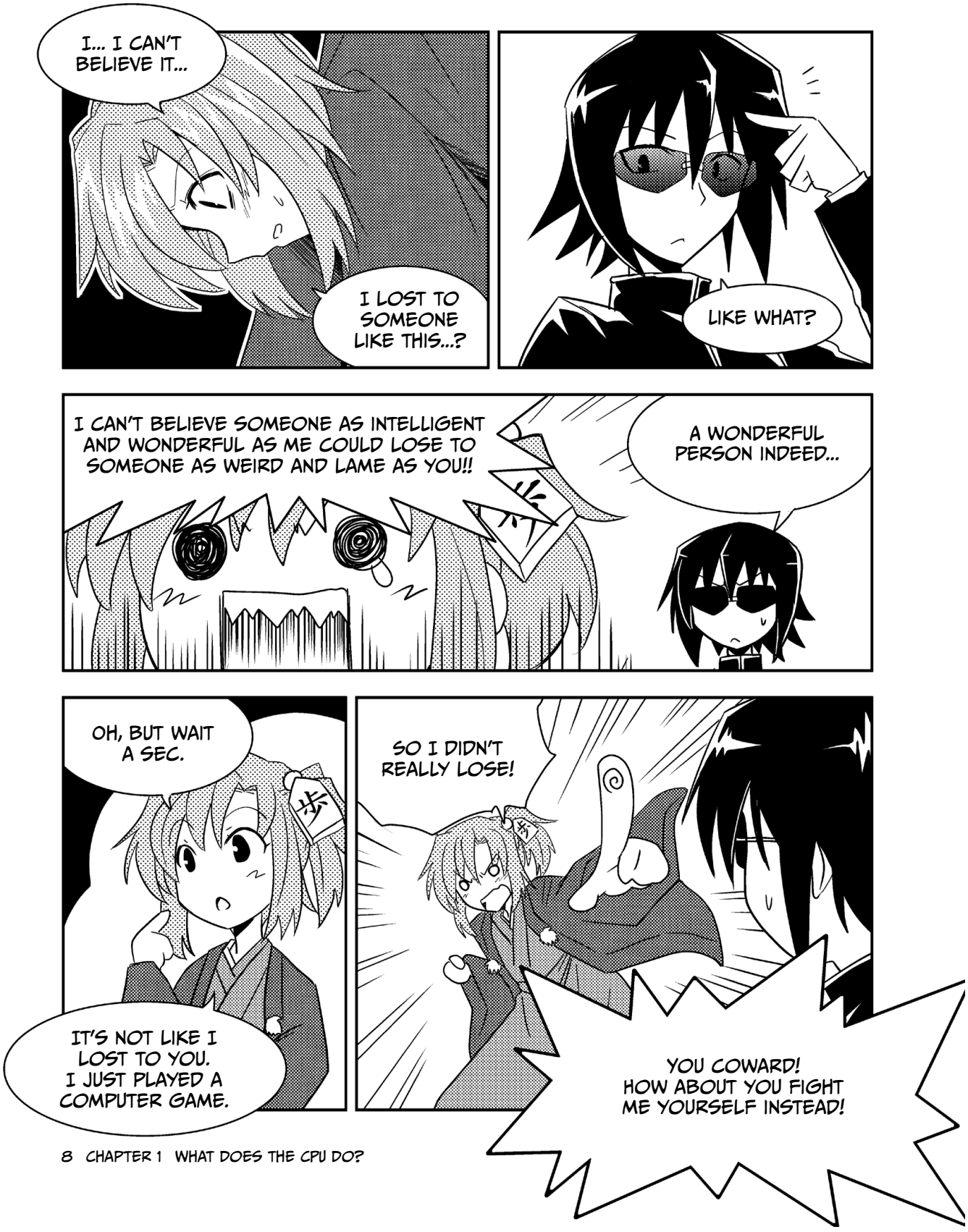

ER, YOU WANT ME, A REAL SHOGI PLAYER, TO PLAY YOUR VIDEO GAME?
HEH... IT'S NOT JUST A COMPUTER GAME.
I'M A BUSY PERSON, YOU KNOW...
THIS LAPTOP, THE SHOOTING STAR, IS RUNNING A PROGRAM OF MY OWN DESIGN.
CRASH
AND... IT'S STRONGER THAN YOU!
FUR
WELLL... I DON'T REALLY GET WHAT YOU'RE SAYING, BUT IT'S OBVIOUS YOU'RE LOOKING DOWN ON ME.
RUMBLE
CRACK
I JUST HAVE TO WIN, RIGHT? I HOPE YOU'RE READY TO BE DESTROYED...

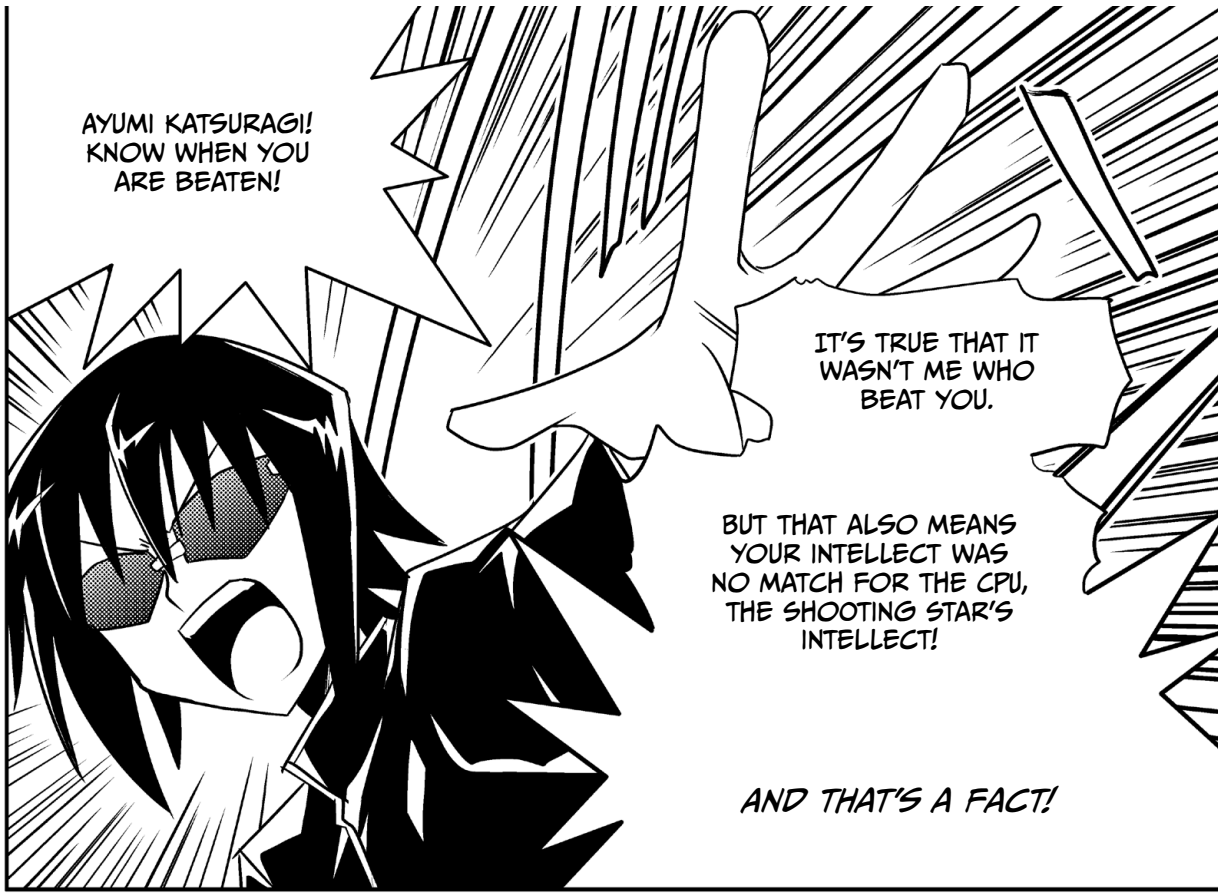
I LOST...?

HOW IS THAT POSSIBLE??



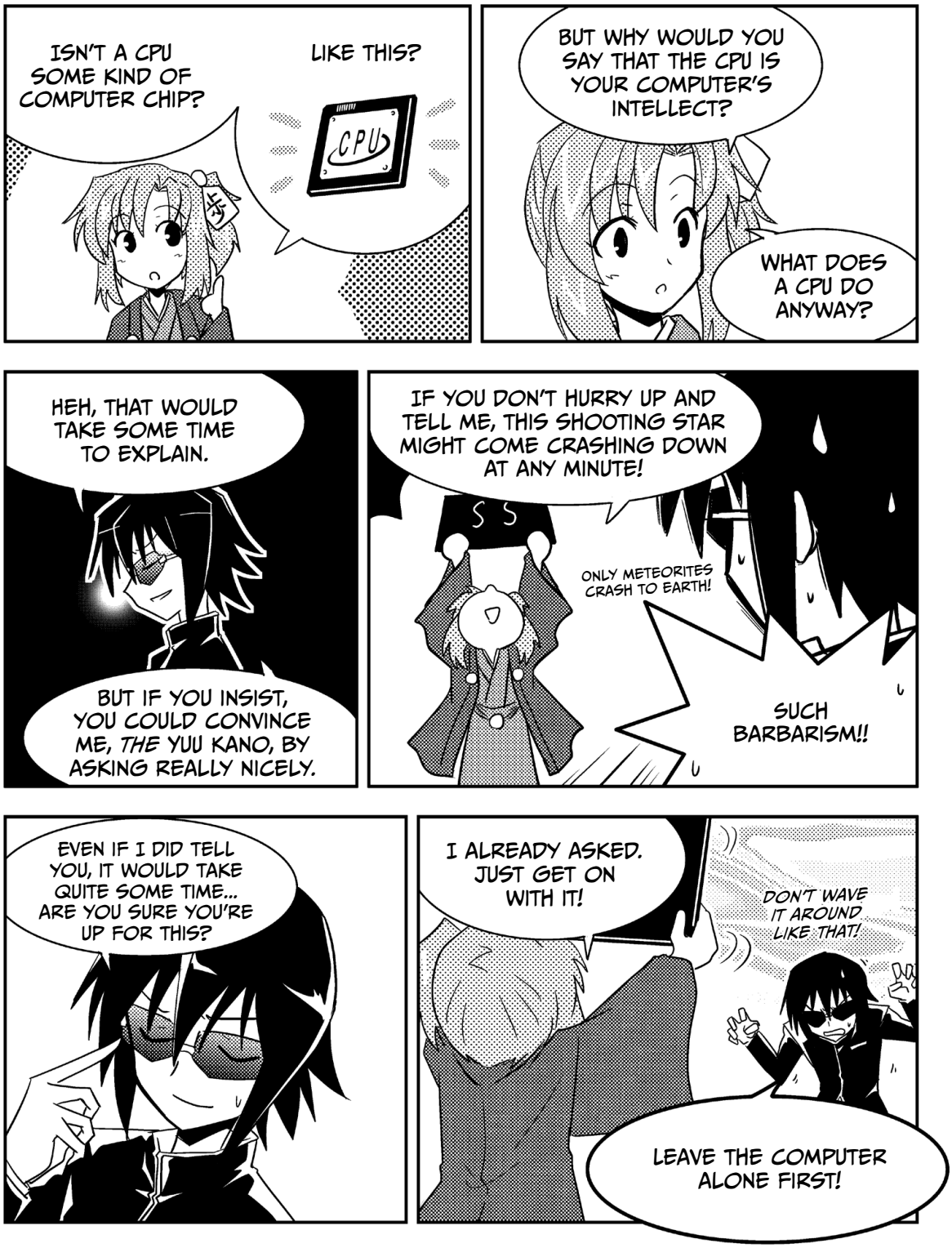



I... I CAN'T BELIEVE IT...
I LOST TO SOMEONE LIKE THIS...?
LIKE WHAT?
I CAN'T BELIEVE SOMEONE AS INTELLIGENT AND WONDERFUL AS ME COULD LOSE TO SOMEONE AS WEIRD AND LAME AS YOU!!
A WONDERFUL PERSON INDEED...
OH, BUT WAIT A SEC.
IT'S NOT LIKE I LOST TO YOU. I JUST PLAYED A COMPUTER GAME.
SO I DIDN'T REALLY LOSE!
YOU COWARD! HOW ABOUT YOU FIGHT ME YOURSELF INSTEAD!

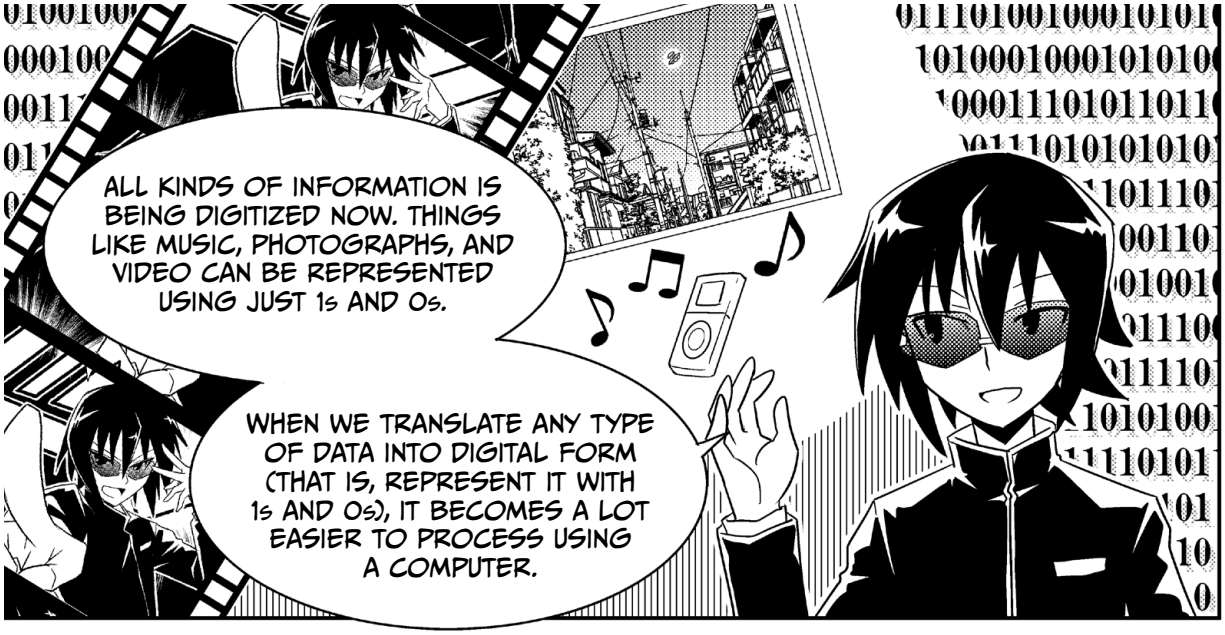

AYUMI KATSURAGI! KNOW WHEN YOU ARE BEATEN!
IT'S TRUE THAT IT WASN'T ME WHO BEAT YOU.
BUT THAT ALSO MEANS YOUR INTELLECT WAS NO MATCH FOR THE CPU, THE SHOOTING STAR'S INTELLECT!
AND THAT'S A FACT!

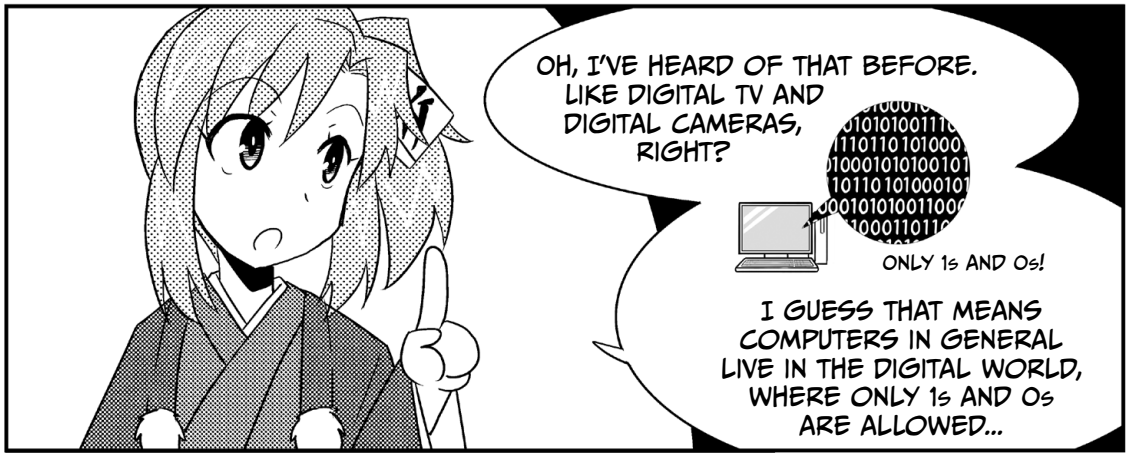

IT ALSO MEANS THAT YOURS TRULY, YUU KANO, THE GENIUS PROGRAMMER WHO BROUGHT THAT CPU TO LIFE, POSSESSES AN EQUAL—
NO! AN EVEN *MORE* IMPRESSIVE INTELLECT!


HUH?
CPU?

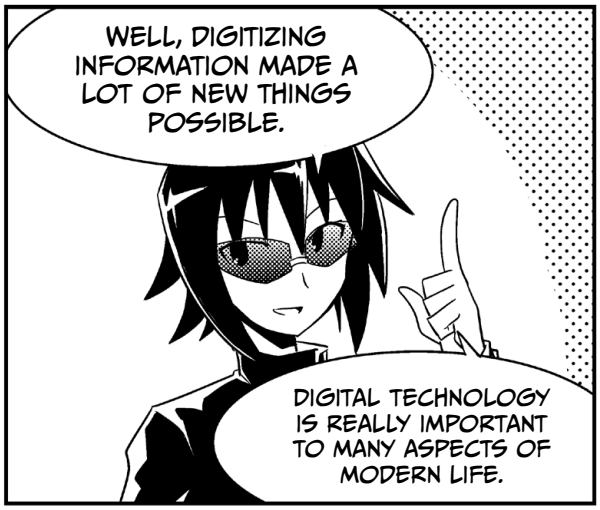

ISN'T A CPU SOME KIND OF COMPUTER CHIP?
LIKE THIS?
CPU
BUT WHY WOULD YOU SAY THAT THE CPU IS YOUR COMPUTER'S INTELLECT?
WHAT DOES A CPU DO ANYWAY?
HEH, THAT WOULD TAKE SOME TIME TO EXPLAIN.
BUT IF YOU INSIST, YOU COULD CONVINCE ME, *THE* YUU KANO, BY ASKING REALLY NICELY.
IF YOU DON'T HURRY UP AND TELL ME, THIS SHOOTING STAR MIGHT COME CRASHING DOWN AT ANY MINUTE!
ONLY METEORITES CRASH TO EARTH!
SUCH BARBARISM!!
EVEN IF I DID TELL YOU, IT WOULD TAKE QUITE SOME TIME... ARE YOU SURE YOU'RE UP FOR THIS?
I ALREADY ASKED. JUST GET ON WITH IT!
DON'T WAVE IT AROUND LIKE THAT!
LEAVE THE COMPUTER ALONE FIRST!

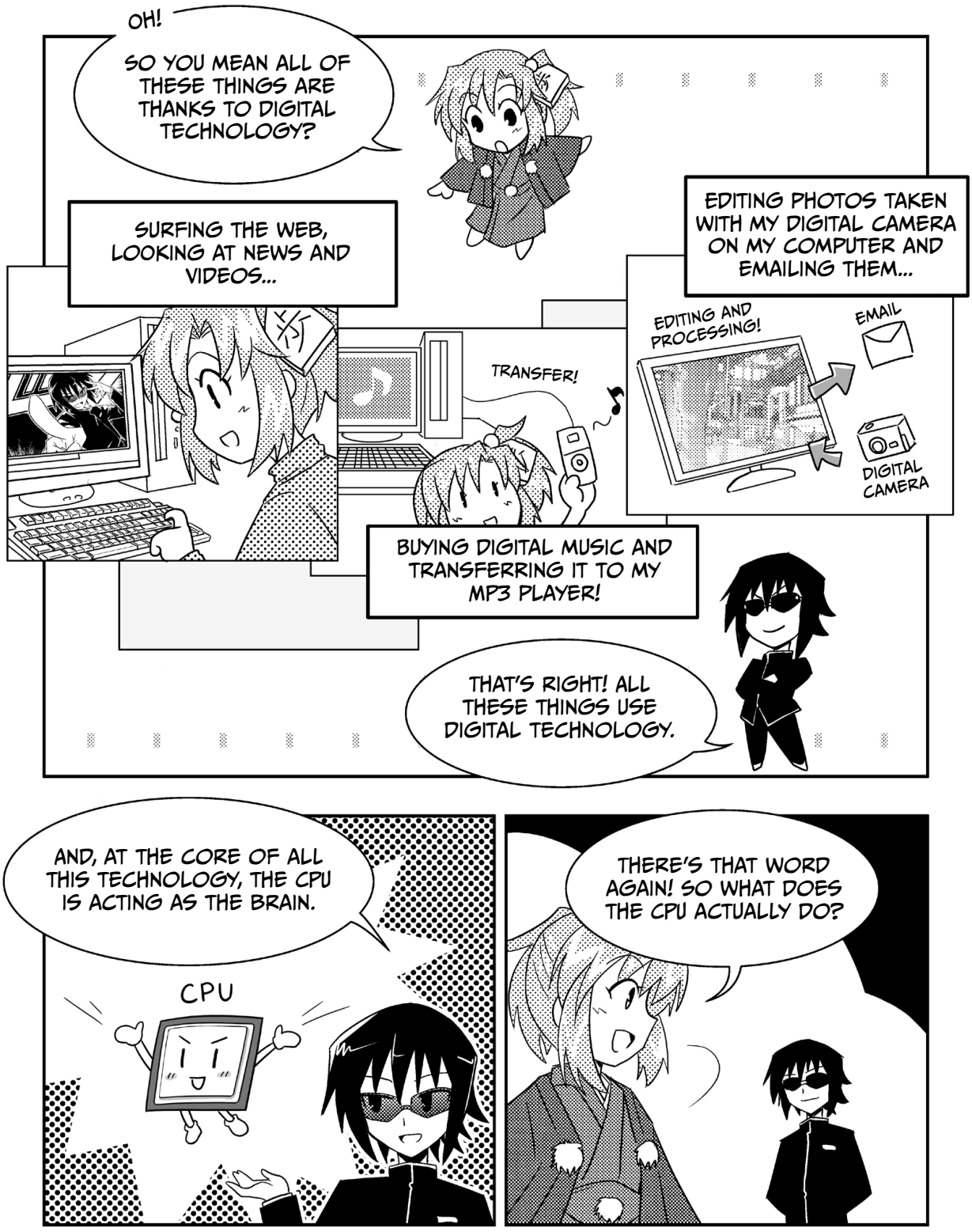
COMPUTERS CAN PROCESS ANY TYPE OF INFORMATION


ALL KINDS OF INFORMATION IS BEING DIGITIZED NOW. THINGS LIKE MUSIC, PHOTOGRAPHS, AND VIDEO CAN BE REPRESENTED USING JUST 1s AND 0s.
WHEN WE TRANSLATE ANY TYPE OF DATA INTO DIGITAL FORM (THAT IS, REPRESENT IT WITH 1s AND 0s), IT BECOMES A LOT EASIER TO PROCESS USING A COMPUTER.

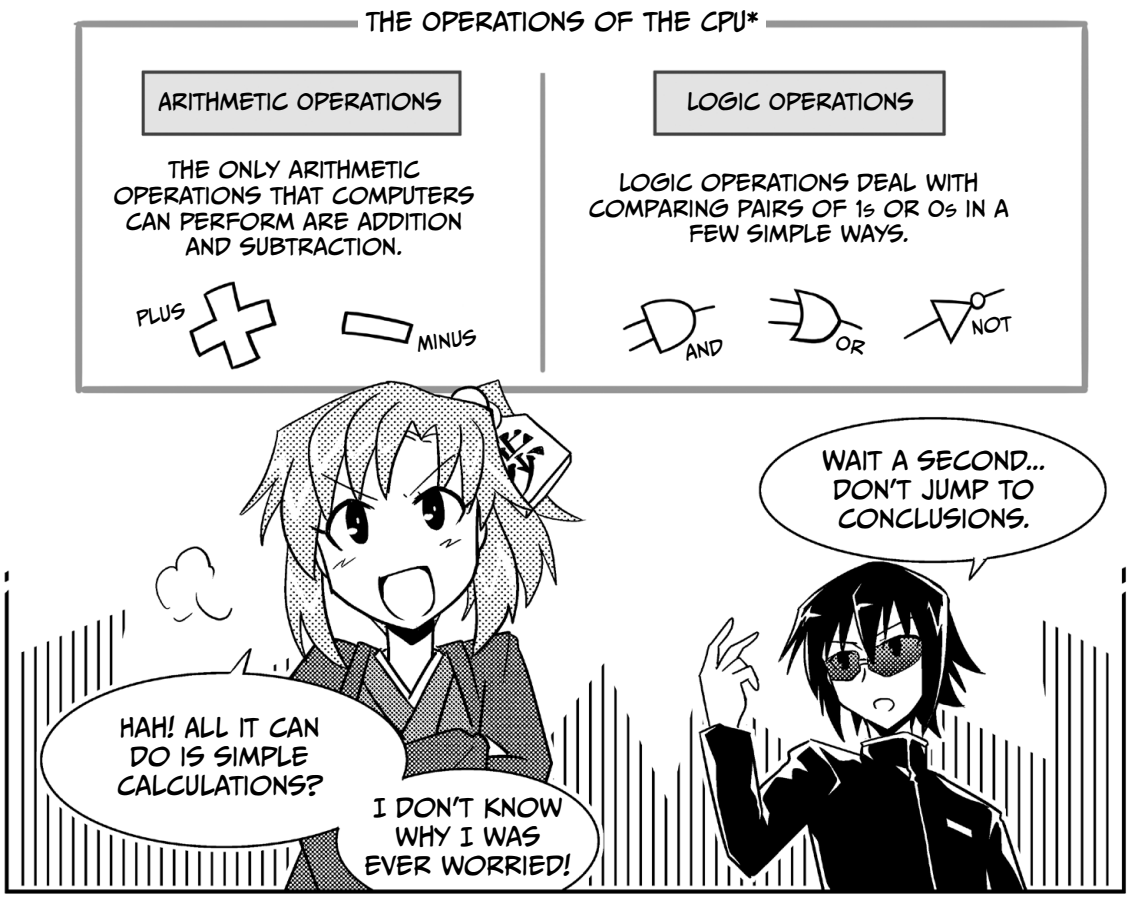

OH, I'VE HEARD OF THAT BEFORE. LIKE DIGITAL TV AND DIGITAL CAMERAS, RIGHT?
ONLY 1s AND 0s!
I GUESS THAT MEANS COMPUTERS IN GENERAL LIVE IN THE DIGITAL WORLD, WHERE ONLY 1s AND 0s ARE ALLOWED...


BUT WHAT'S SO GREAT ABOUT THAT?

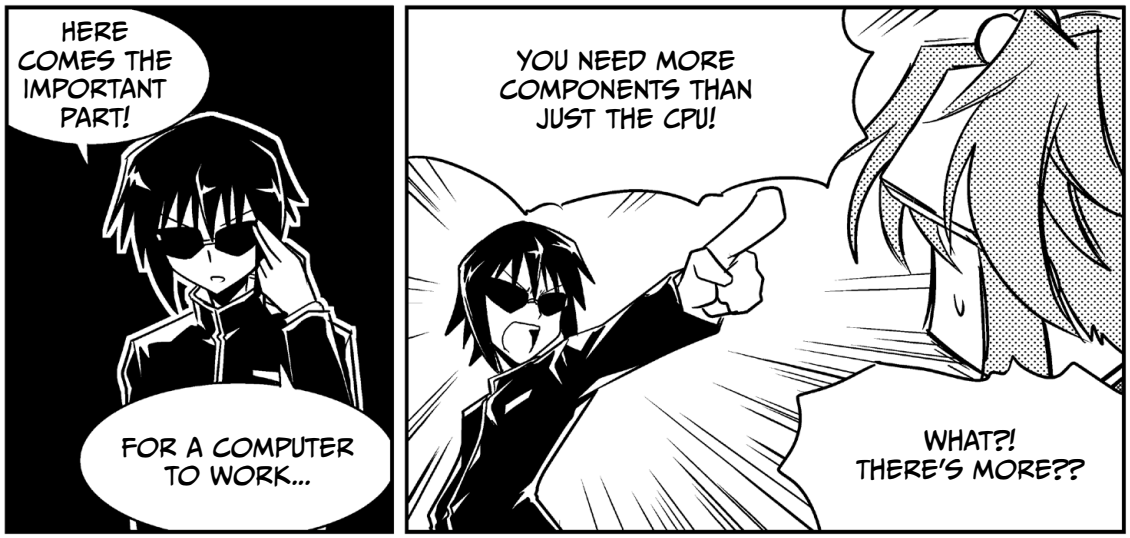

WELL, DIGITIZING INFORMATION MADE A LOT OF NEW THINGS POSSIBLE.
DIGITAL TECHNOLOGY IS REALLY IMPORTANT TO MANY ASPECTS OF MODERN LIFE.

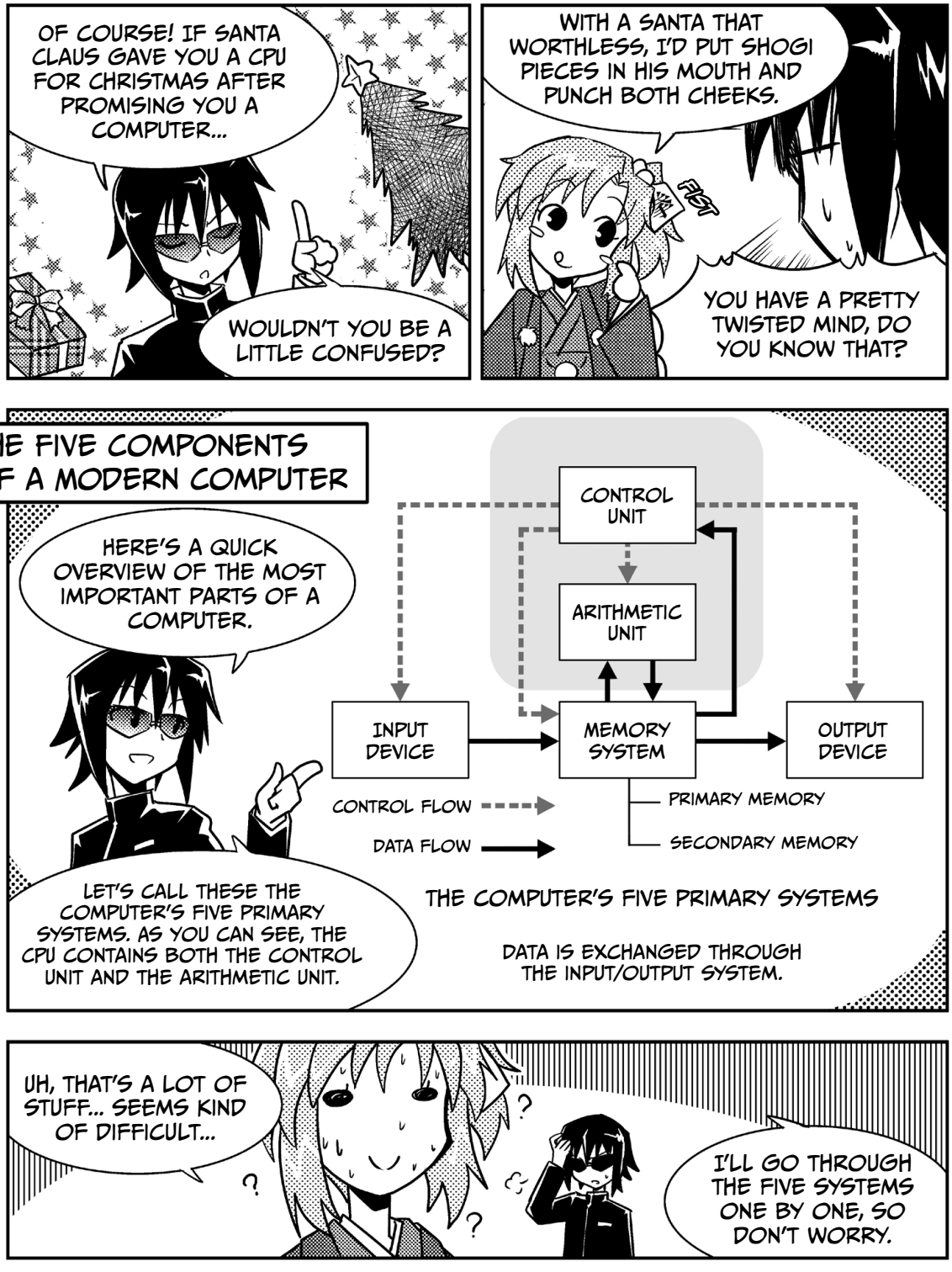

OH!
SO YOU MEAN ALL OF THESE THINGS ARE THANKS TO DIGITAL TECHNOLOGY?
SURFING THE WEB, LOOKING AT NEWS AND VIDEOS...
EDITING PHOTOS TAKEN WITH MY DIGITAL CAMERA ON MY COMPUTER AND EMAILING THEM...
EDITING AND PROCESSING!
EMAIL
DIGITAL CAMERA
TRANSFER!
BUYING DIGITAL MUSIC AND TRANSFERRING IT TO MY MP3 PLAYER!
THAT'S RIGHT! ALL THESE THINGS USE DIGITAL TECHNOLOGY.
AND, AT THE CORE OF ALL THIS TECHNOLOGY, THE CPU IS ACTING AS THE BRAIN.
CPU
THERE'S THAT WORD AGAIN! SO WHAT DOES THE CPU ACTUALLY DO?

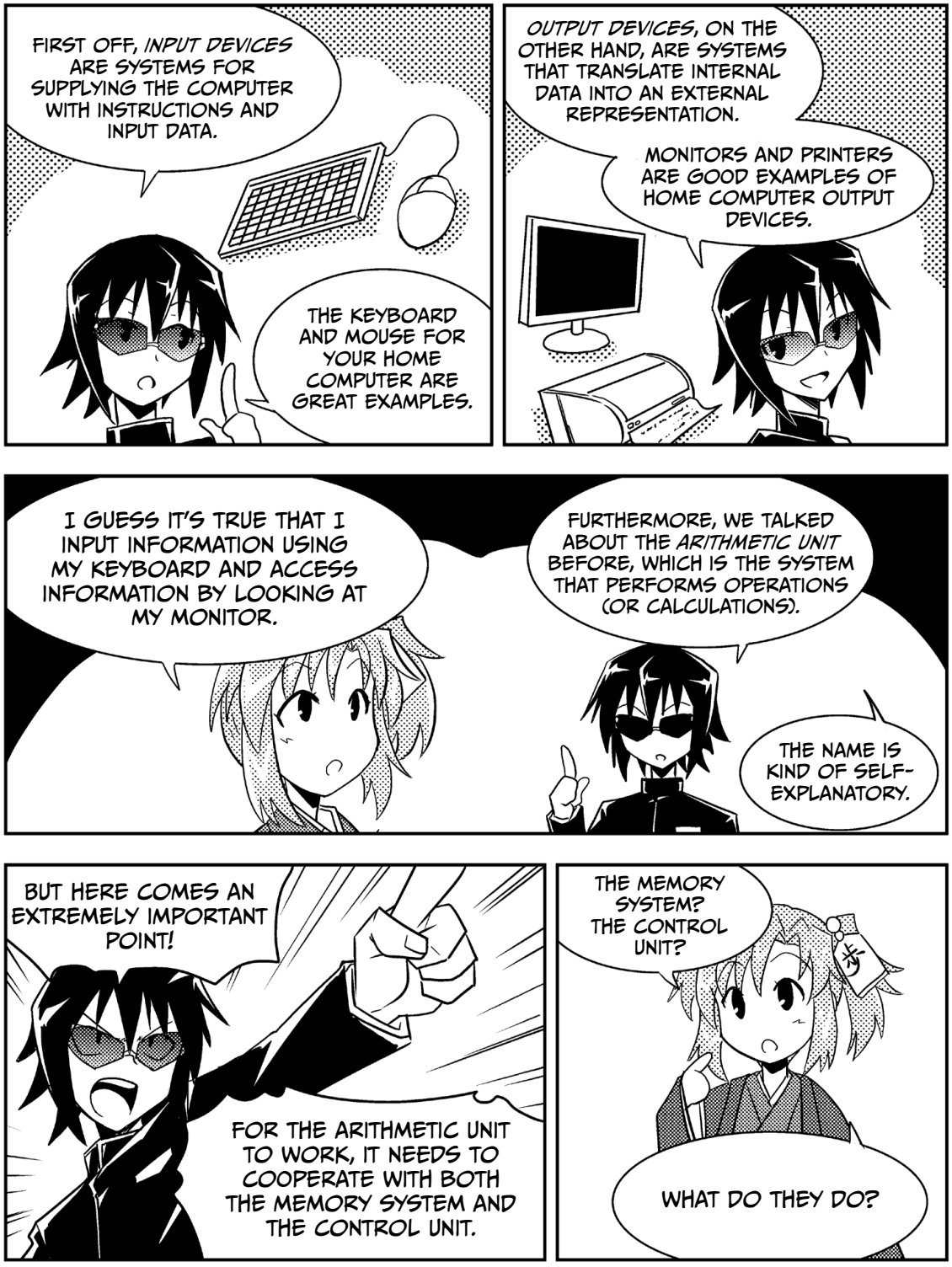

THE CPU IS THE CORE OF EACH COMPUTER
CPU IS SHORT FOR CENTRAL PROCESSING UNIT.
WHICH MEANS THAT THE CPU...
IS IN CHARGE OF THE COMPUTER'S OPERATIONS!
WHAAAAAAAAT! OPERATIONS?!!
WHAT ARE OPERATIONS?
OPERATIONS ARE COMPUTER CALCULATIONS,
CALCULATIONS USING ONLY 1s AND 0s.
IT'S ALSO WORTH MENTIONING THAT CPUs PERFORM TWO KINDS OF OPERATIONS.

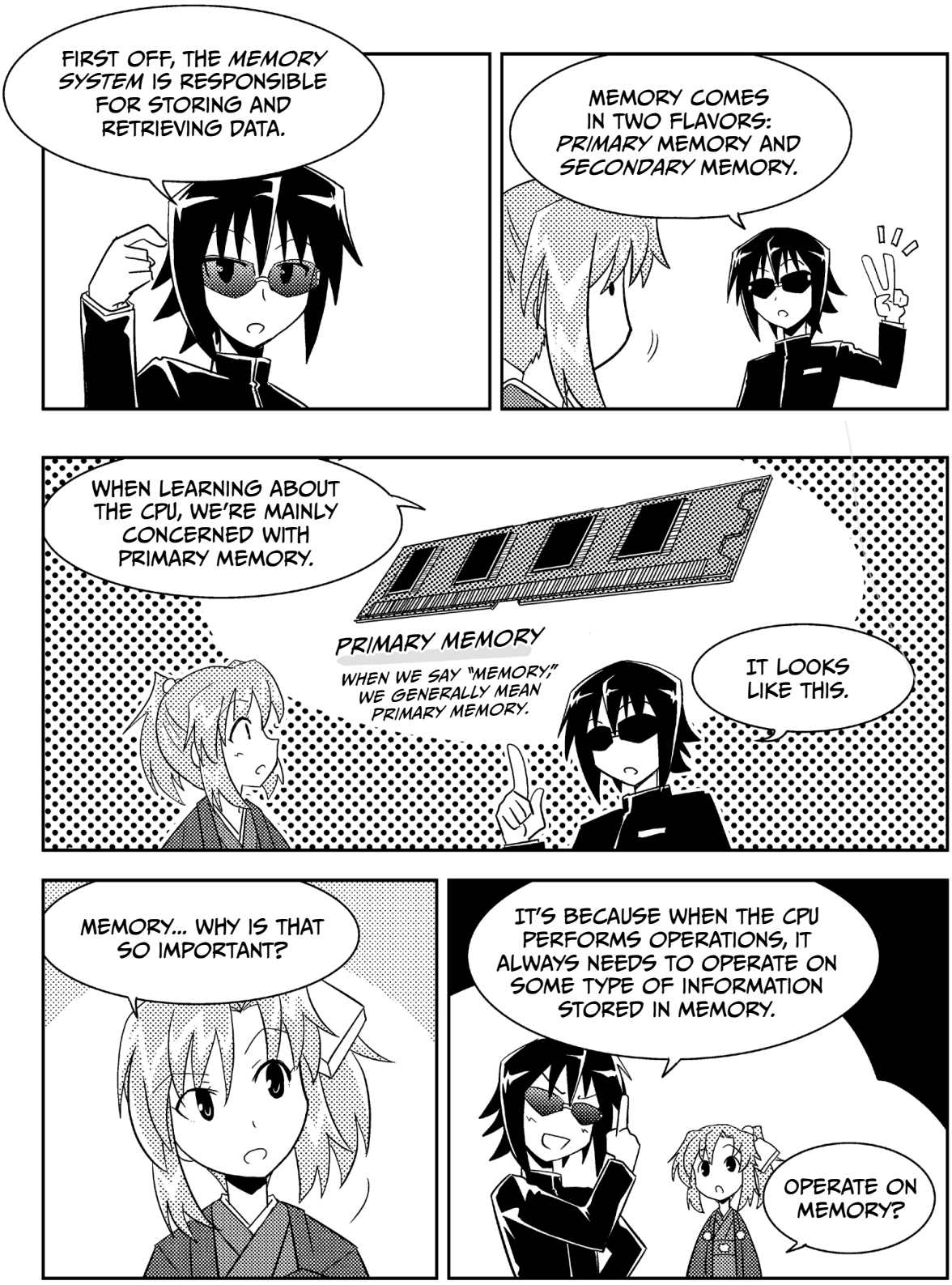


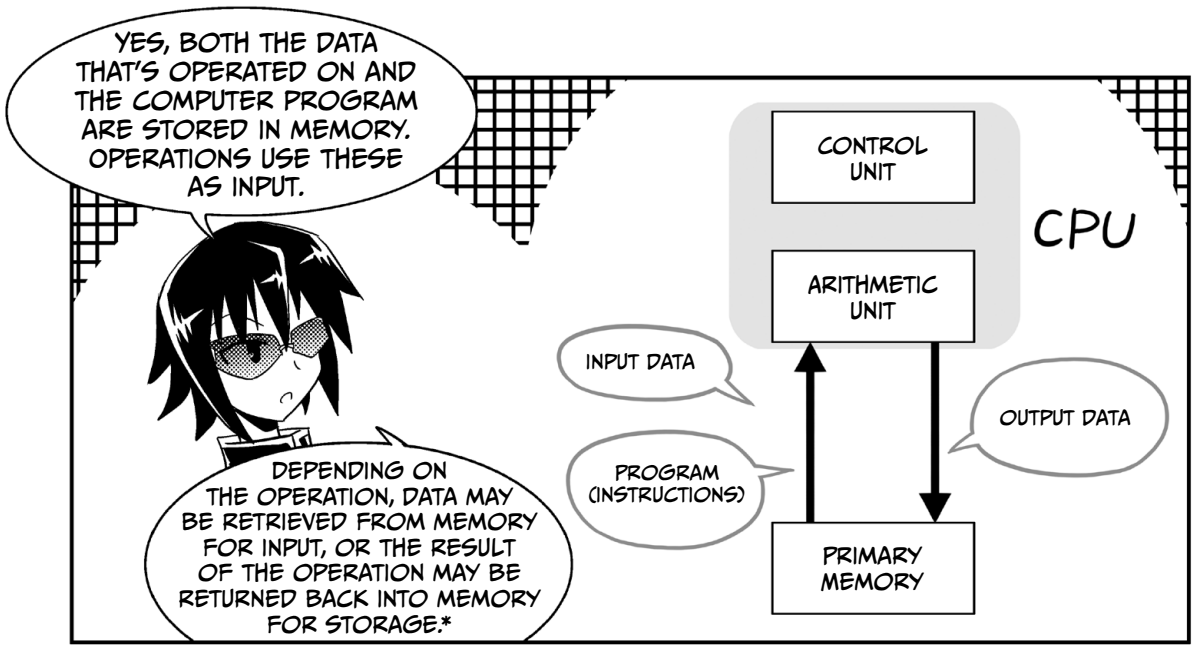
* IN ADDITION TO THESE OPERATIONAL UNITS, MODERN CPUs ALSO CONTAIN FPUs (FLOATING POINT UNITS) THAT CAN HANDLE MULTIPLICATION AND DIVISION. BUT THIS BOOK JUST STICKS TO THE BASICS.



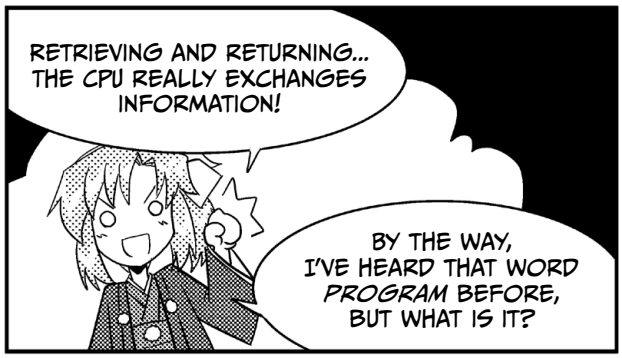

OF COURSE! IF SANTA CLAUS GAVE YOU A CPU FOR CHRISTMAS AFTER PROMISING YOU A COMPUTER...
WOULDN'T YOU BE A LITTLE CONFUSED?
WITH A SANTA THAT WORTHLESS, I'D PUT SHOGI PIECES IN HIS MOUTH AND PUNCH BOTH CHEEKS.
FIST
YOU HAVE A PRETTY TWISTED MIND, DO YOU KNOW THAT?
THE FIVE COMPONENTS OF A MODERN COMPUTER
HERE'S A QUICK OVERVIEW OF THE MOST IMPORTANT PARTS OF A COMPUTER.
CONTROL UNIT
ARITHMETIC UNIT
INPUT DEVICE
MEMORY SYSTEM
OUTPUT DEVICE
PRIMARY MEMORY
SECONDARY MEMORY
CONTROL FLOW
DATA FLOW
LET'S CALL THESE THE COMPUTER'S FIVE PRIMARY SYSTEMS. AS YOU CAN SEE, THE CPU CONTAINS BOTH THE CONTROL UNIT AND THE ARITHMETIC UNIT.
THE COMPUTER'S FIVE PRIMARY SYSTEMS
DATA IS EXCHANGED THROUGH THE INPUT/OUTPUT SYSTEM.
UH, THAT'S A LOT OF STUFF... SEEMS KIND OF DIFFICULT...
I'LL GO THROUGH THE FIVE SYSTEMS ONE BY ONE, SO DON'T WORRY.

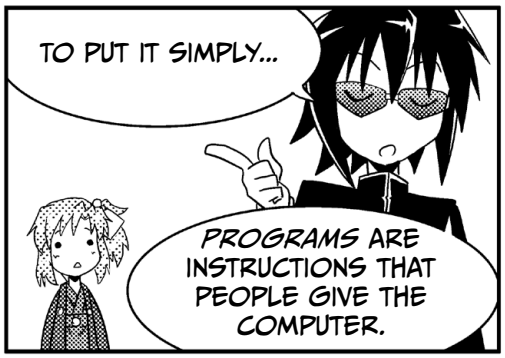

FIRST OFF, *INPUT DEVICES* ARE SYSTEMS FOR SUPPLYING THE COMPUTER WITH INSTRUCTIONS AND INPUT DATA.
THE KEYBOARD AND MOUSE FOR YOUR HOME COMPUTER ARE GREAT EXAMPLES.
OUTPUT DEVICES, ON THE OTHER HAND, ARE SYSTEMS THAT TRANSLATE INTERNAL DATA INTO AN EXTERNAL REPRESENTATION.
MONITORS AND PRINTERS ARE GOOD EXAMPLES OF HOME COMPUTER OUTPUT DEVICES.
I GUESS IT'S TRUE THAT I INPUT INFORMATION USING MY KEYBOARD AND ACCESS INFORMATION BY LOOKING AT MY MONITOR.
FURTHERMORE, WE TALKED ABOUT THE *ARITHMETIC UNIT* BEFORE, WHICH IS THE SYSTEM THAT PERFORMS OPERATIONS (OR CALCULATIONS).
THE NAME IS KIND OF SELF-EXPLANATORY.
BUT HERE COMES AN EXTREMELY IMPORTANT POINT!
FOR THE ARITHMETIC UNIT TO WORK, IT NEEDS TO COOPERATE WITH BOTH THE MEMORY SYSTEM AND THE CONTROL UNIT.
THE MEMORY SYSTEM? THE CONTROL UNIT?
WHAT DO THEY DO?

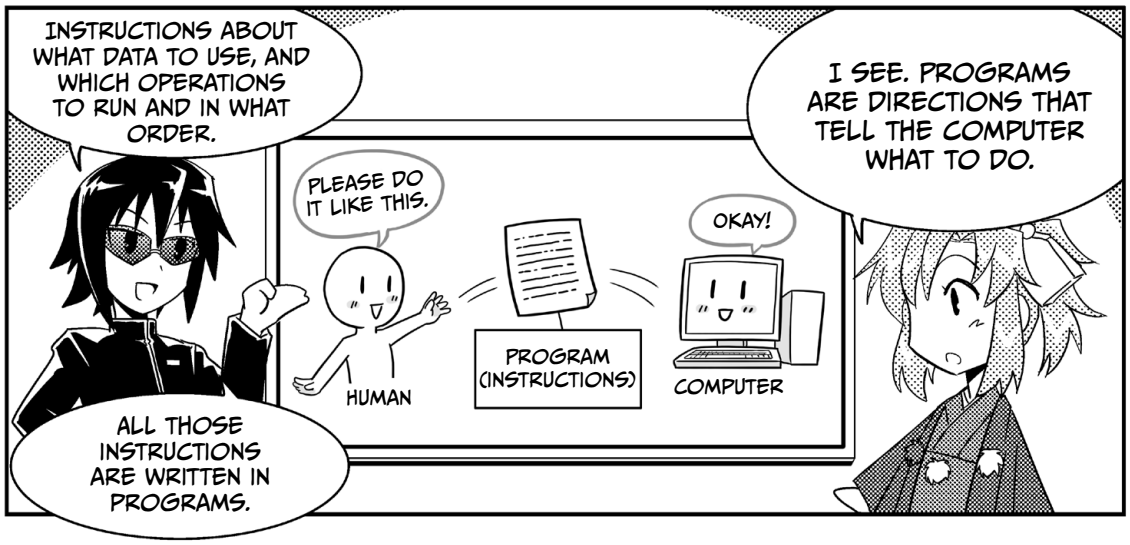

FIRST OFF, THE *MEMORY SYSTEM* IS RESPONSIBLE FOR STORING AND RETRIEVING DATA.
MEMORY COMES IN TWO FLAVORS: *PRIMARY* MEMORY AND *SECONDARY* MEMORY.
WHEN LEARNING ABOUT THE CPU, WE'RE MAINLY CONCERNED WITH PRIMARY MEMORY.
PRIMARY MEMORY
WHEN WE SAY "MEMORY," WE GENERALLY MEAN PRIMARY MEMORY.
IT LOOKS LIKE THIS.
MEMORY... WHY IS THAT SO IMPORTANT?
IT'S BECAUSE WHEN THE CPU PERFORMS OPERATIONS, IT ALWAYS NEEDS TO OPERATE ON SOME TYPE OF INFORMATION STORED IN MEMORY.
OPERATE ON MEMORY?

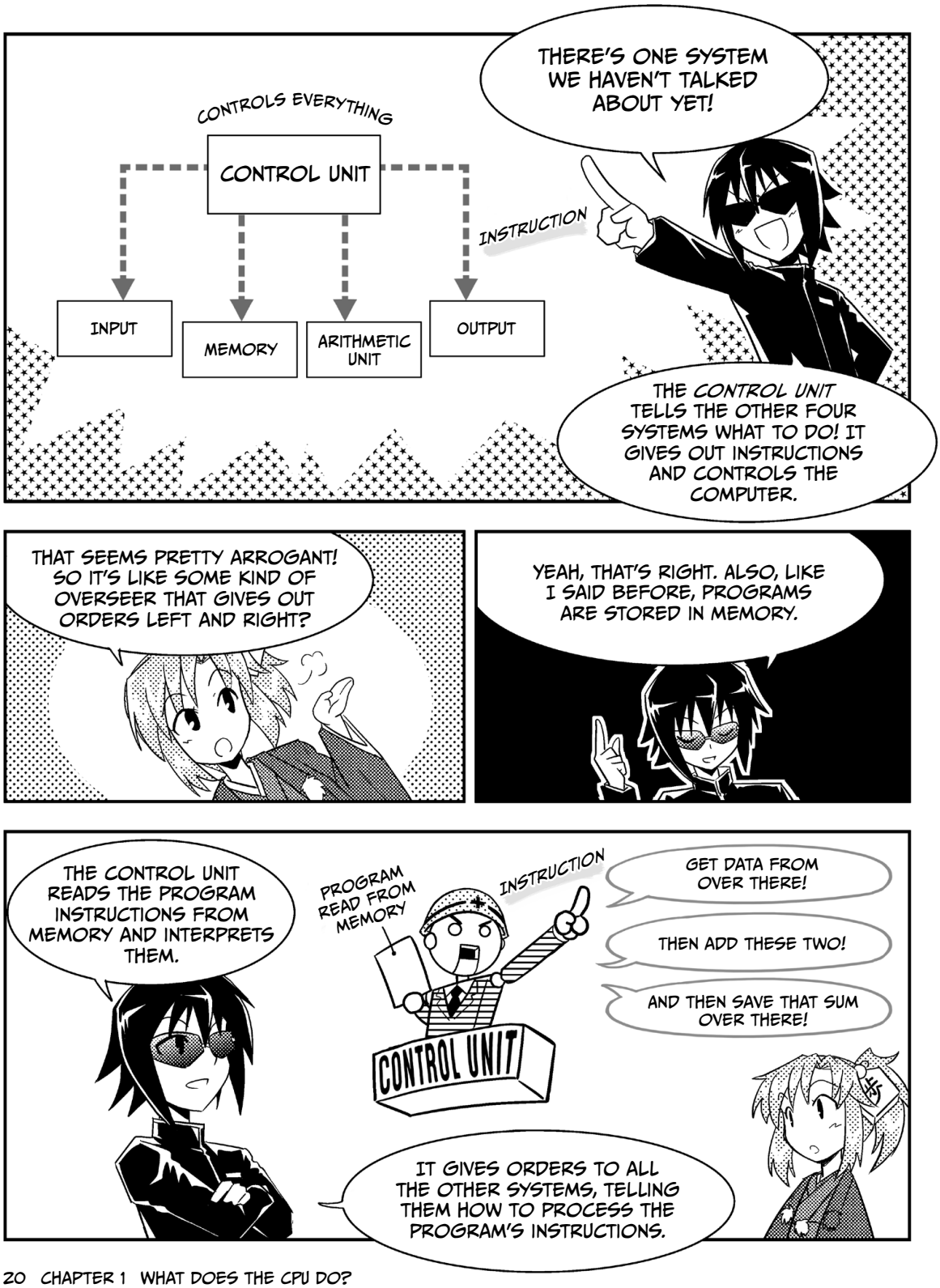


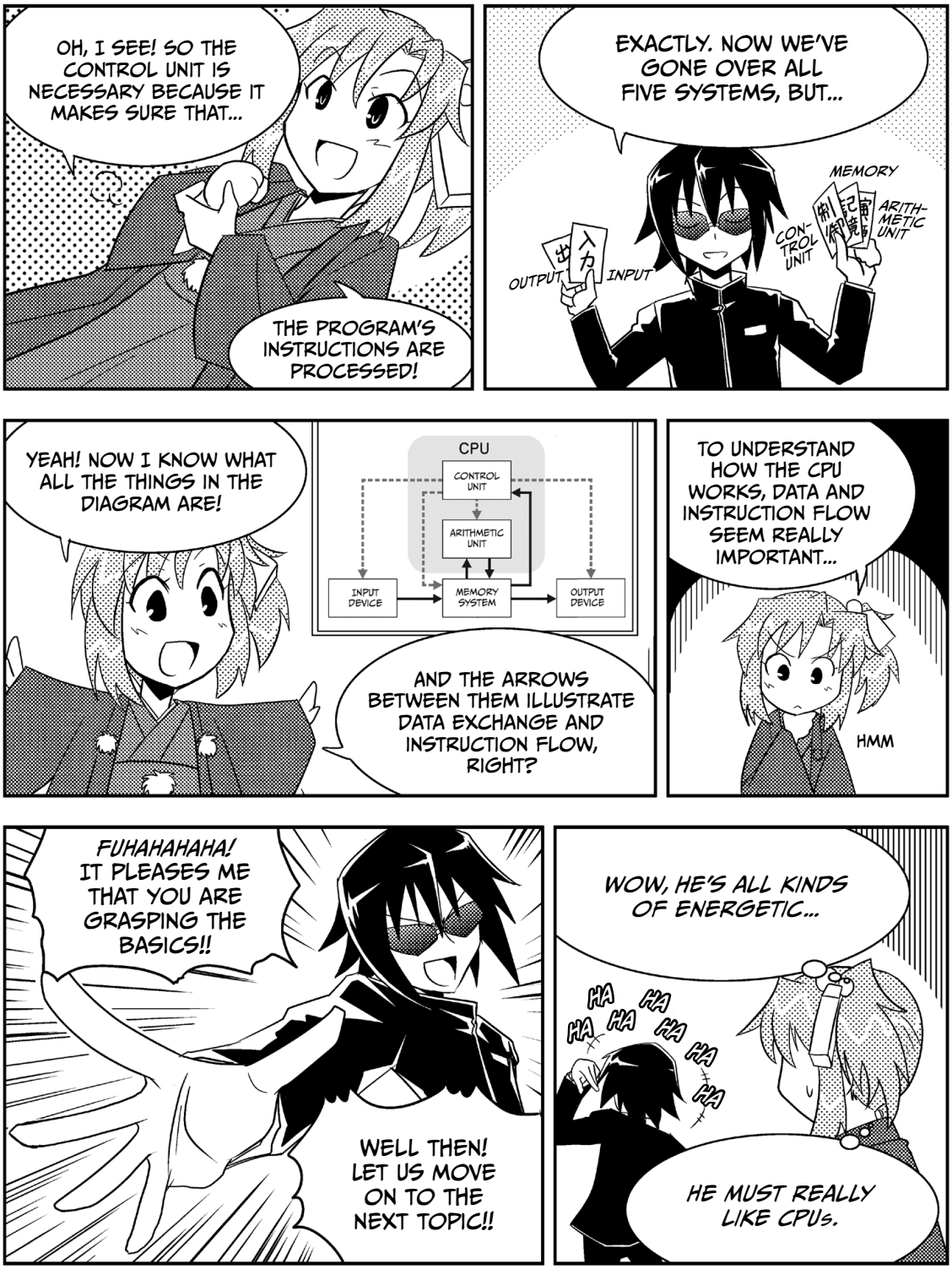
* THE CPU MAY USE EITHER REGISTERS OR CACHE MEMORY.



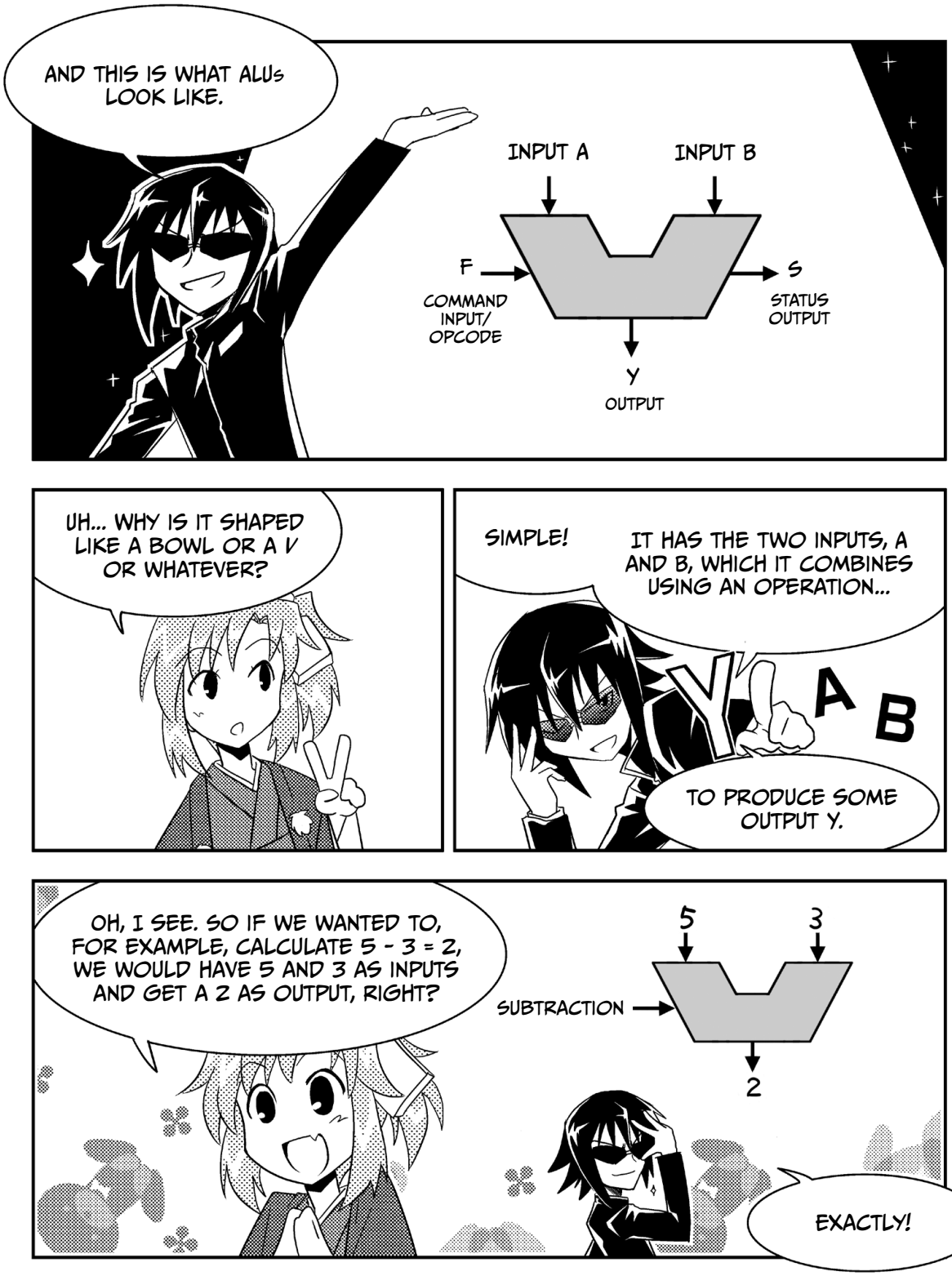


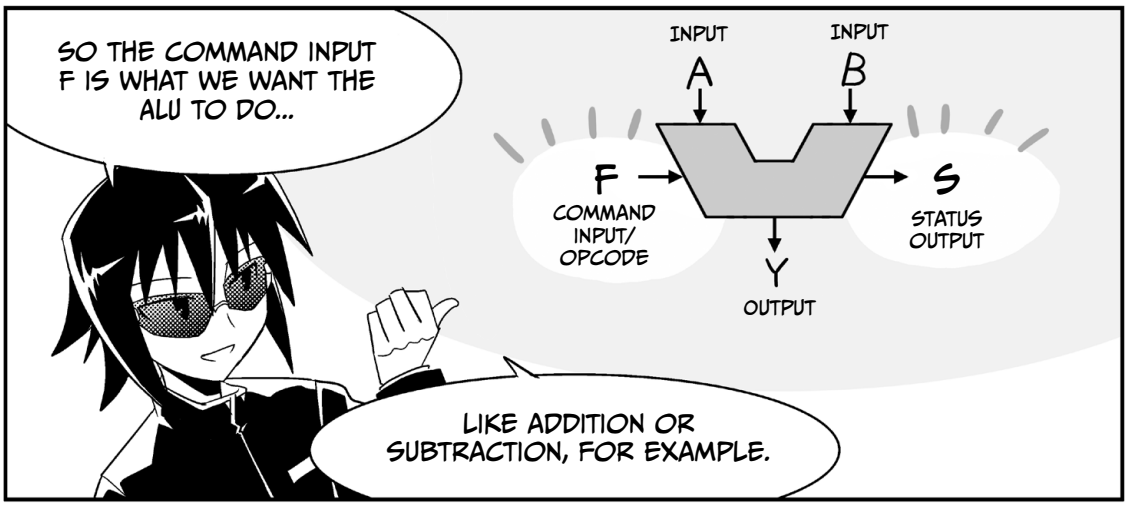


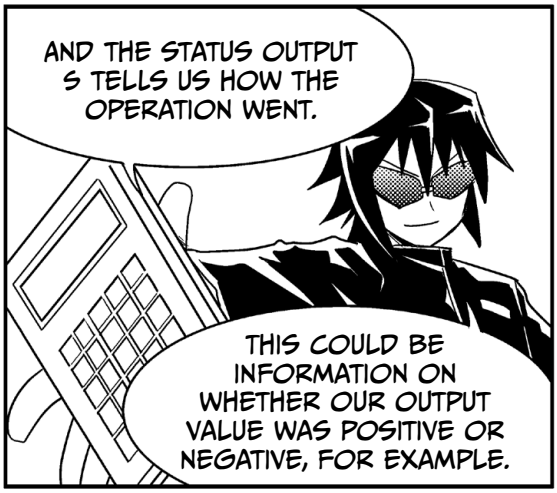

THERE'S ONE SYSTEM WE HAVEN'T TALKED ABOUT YET!
CONTROLS EVERYTHING
CONTROL UNIT
INSTRUCTION
INPUT
MEMORY
ARITHMETIC UNIT
OUTPUT
THE CONTROL UNIT TELLS THE OTHER FOUR SYSTEMS WHAT TO DO! IT GIVES OUT INSTRUCTIONS AND CONTROLS THE COMPUTER.
THAT SEEMS PRETTY ARROGANT! SO IT'S LIKE SOME KIND OF OVERSEER THAT GIVES OUT ORDERS LEFT AND RIGHT?
YEAH, THAT'S RIGHT. ALSO, LIKE I SAID BEFORE, PROGRAMS ARE STORED IN MEMORY.
THE CONTROL UNIT READS THE PROGRAM INSTRUCTIONS FROM MEMORY AND INTERPRETS THEM.
PROGRAM READ FROM MEMORY
INSTRUCTION
CONTROL UNIT
GET DATA FROM OVER THERE!
THEN ADD THESE TWO!
AND THEN SAVE THAT SUM OVER THERE!
IT GIVES ORDERS TO ALL THE OTHER SYSTEMS, TELLING THEM HOW TO PROCESS THE PROGRAM'S INSTRUCTIONS.

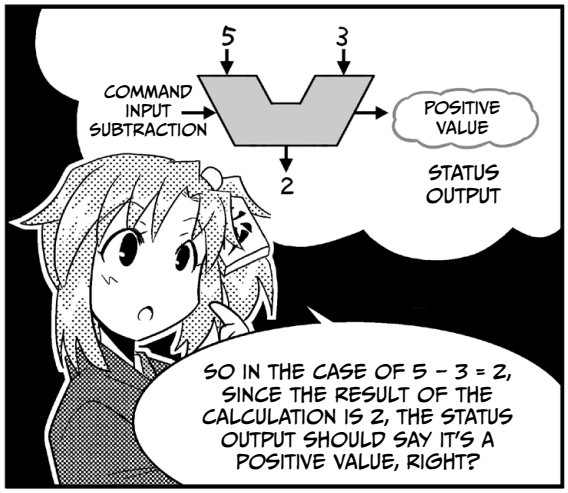

OH, I SEE! SO THE CONTROL UNIT IS NECESSARY BECAUSE IT MAKES SURE THAT...
THE PROGRAM'S INSTRUCTIONS ARE PROCESSED!
EXACTLY. NOW WE'VE GONE OVER ALL FIVE SYSTEMS, BUT...
MEMORY
ARITH-METIC UNIT
CON-TROL UNIT
OUTPUT
INPUT
YEAH! NOW I KNOW WHAT ALL THE THINGS IN THE DIAGRAM ARE!
CPU
CONTROL UNIT
ARITHMETIC UNIT
INPUT DEVICE
MEMORY SYSTEM
OUTPUT DEVICE
AND THE ARROWS BETWEEN THEM ILLUSTRATE DATA EXCHANGE AND INSTRUCTION FLOW, RIGHT?
TO UNDERSTAND HOW THE CPU WORKS, DATA AND INSTRUCTION FLOW SEEM REALLY IMPORTANT...
HMM
FUHAHAHAHA! IT PLEASES ME THAT YOU ARE GRASPING THE BASICS!!
WELL THEN! LET US MOVE ON TO THE NEXT TOPIC!!
WOW, HE'S ALL KINDS OF ENERGETIC...
HA HA HA HA HA HA HA HA
HE MUST REALLY LIKE CPUs.

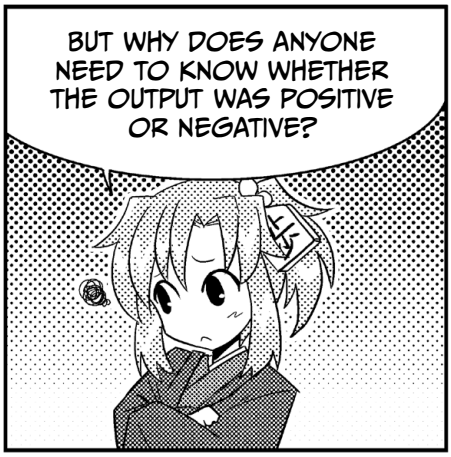

ALUs: THE CPU'S CORE
YOU'RE CATCHING ON PRETTY QUICKLY, IT SEEMS.
SO LET'S TALK A BIT ABOUT ALUs.
ALUs? NOT CPUs? WHAT'S THE DIFFERENCE?
WELL, ALUs ARE WHAT PERFORM OPERATIONS INSIDE THE CPU.
CONTROL UNIT
CPU
ARITHMETIC UNIT (ALU)
ALUs ARE THE ARITHMETIC UNIT'S PRINCIPAL COMPONENTS.
OH! THAT SEEMS LIKE IT'S SUPER IMPORTANT!
YES, ALU IS SHORT FOR ARITHMETIC LOGIC UNIT.
IT PERFORMS THE ARITHMETIC AND LOGIC OPERATIONS WE TALKED ABOUT BEFORE.

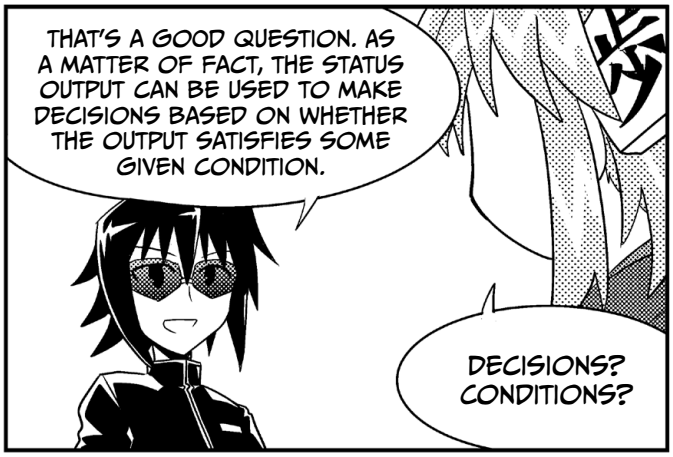

AND THIS IS WHAT ALUs LOOK LIKE.
INPUT A
INPUT B
F
COMMAND INPUT/ OPCODE
S
STATUS OUTPUT
Y
OUTPUT
UH... WHY IS IT SHAPED LIKE A BOWL OR A V OR WHATEVER?
SIMPLE!
IT HAS THE TWO INPUTS, A AND B, WHICH IT COMBINES USING AN OPERATION...
Y
A
B
TO PRODUCE SOME OUTPUT Y.
OH, I SEE. SO IF WE WANTED TO, FOR EXAMPLE, CALCULATE 5 - 3 = 2, WE WOULD HAVE 5 AND 3 AS INPUTS AND GET A 2 AS OUTPUT, RIGHT?
5
3
SUBTRACTION
2
EXACTLY!


SO THE COMMAND INPUT F IS WHAT WE WANT THE ALU TO DO...
INPUT
A
INPUT
B
F
COMMAND INPUT/ OPCODE
S
STATUS OUTPUT
Y
OUTPUT
LIKE ADDITION OR SUBTRACTION, FOR EXAMPLE.

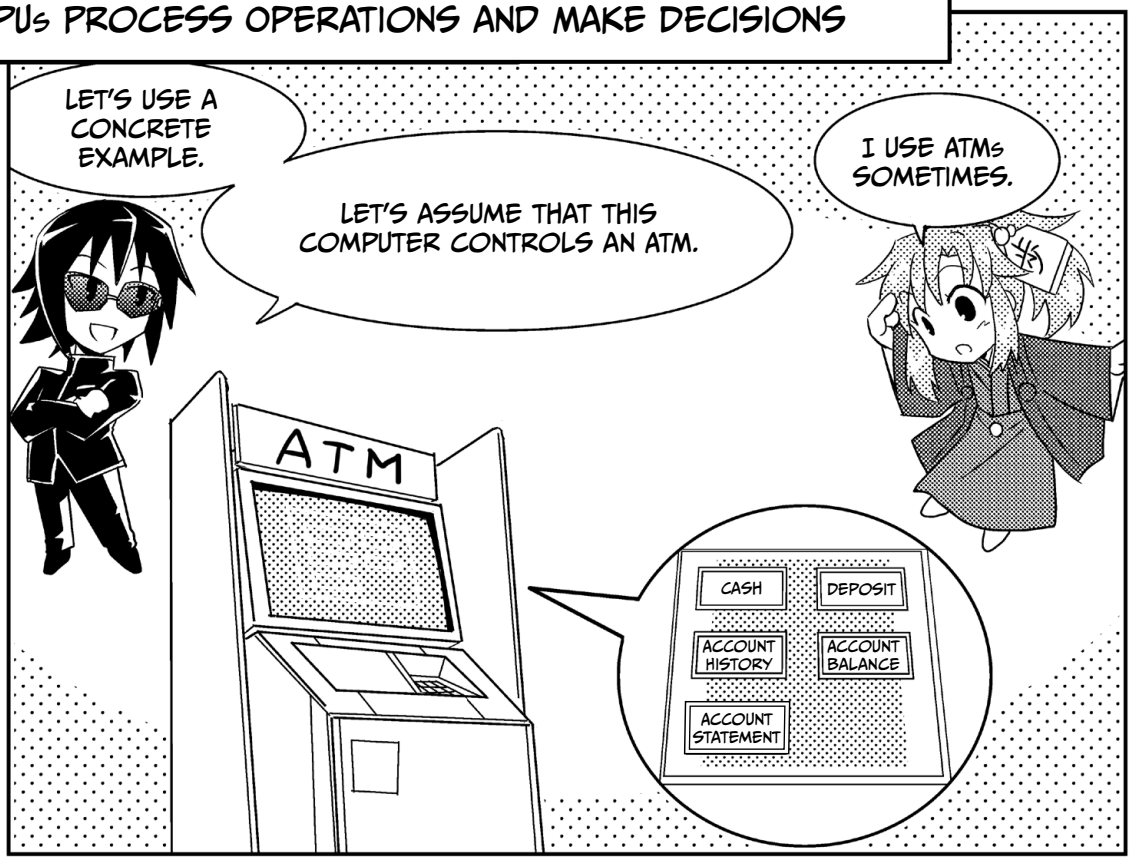

AND THE STATUS OUTPUT S TELLS US HOW THE OPERATION WENT.
THIS COULD BE INFORMATION ON WHETHER OUR OUTPUT VALUE WAS POSITIVE OR NEGATIVE, FOR EXAMPLE.

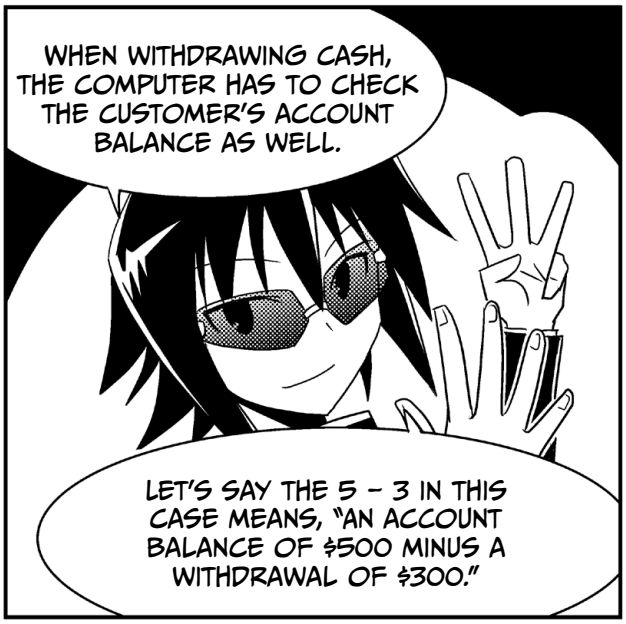

5
3
COMMAND INPUT SUBTRACTION
POSITIVE VALUE
2
STATUS OUTPUT
SO IN THE CASE OF 5 − 3 = 2, SINCE THE RESULT OF THE CALCULATION IS 2, THE STATUS OUTPUT SHOULD SAY IT'S A POSITIVE VALUE, RIGHT?

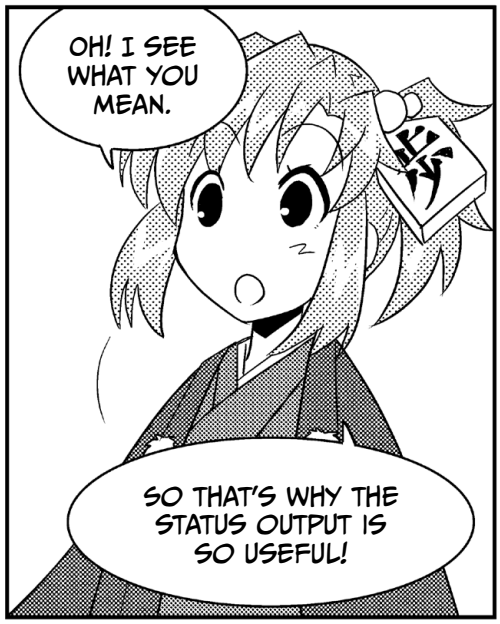

BUT WHY DOES ANYONE NEED TO KNOW WHETHER THE OUTPUT WAS POSITIVE OR NEGATIVE?


THAT'S A GOOD QUESTION. AS A MATTER OF FACT, THE STATUS OUTPUT CAN BE USED TO MAKE DECISIONS BASED ON WHETHER THE OUTPUT SATISFIES SOME GIVEN CONDITION.
DECISIONS? CONDITIONS?

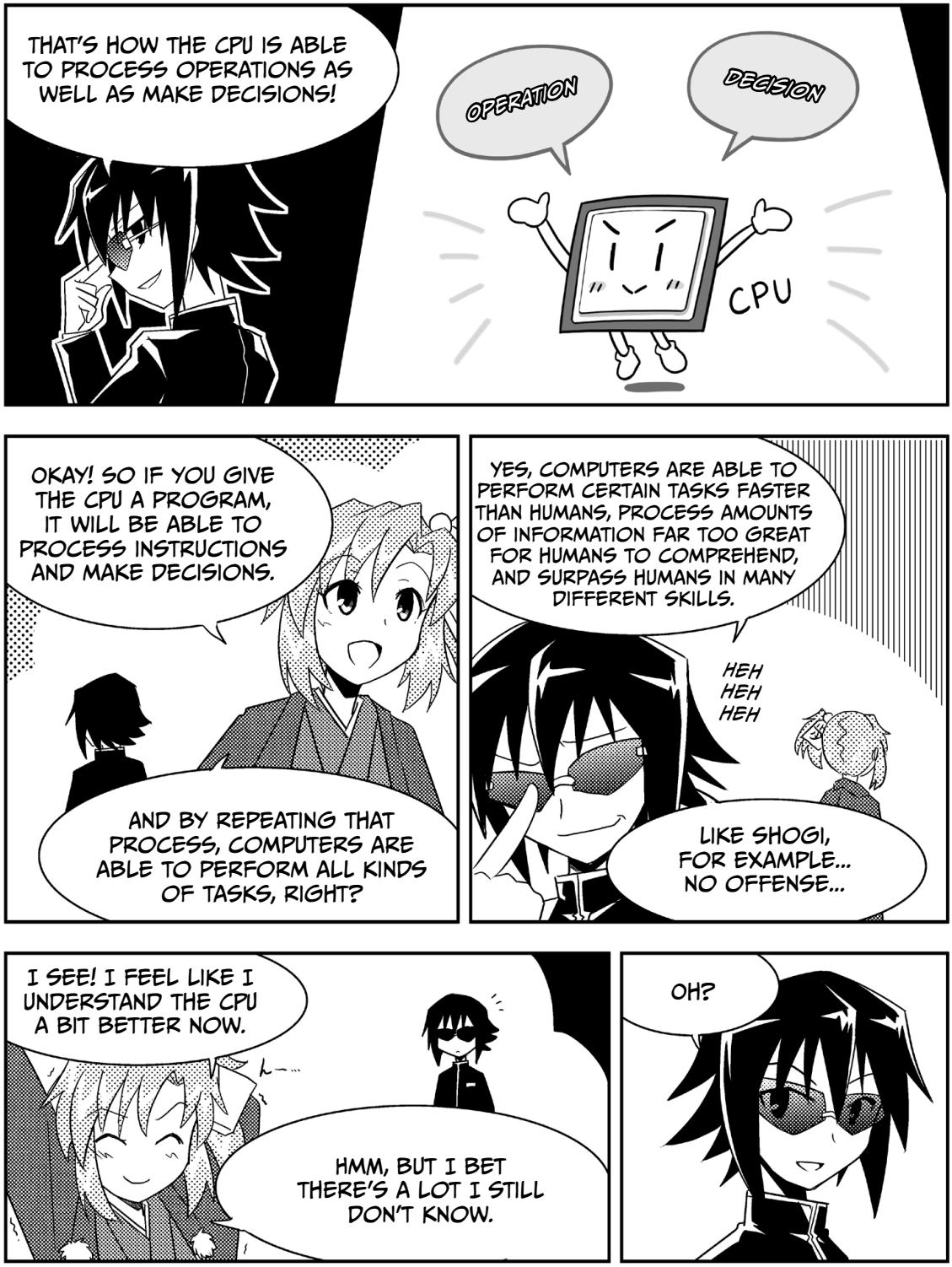
CPUs PROCESS OPERATIONS AND MAKE DECISIONS








BECAUSE WHAT HAPPENS AFTER THE SUBTRACTION IS DONE WILL BE COMPLETELY DIFFERENT DEPENDING ON WHETHER THE RESULT WAS POSITIVE OR NEGATIVE.
IF THE RESULT IS POSITIVE, THAT MEANS YOU HAVE ENOUGH MONEY IN YOUR ACCOUNT, SO THE ATM WILL GIVE THE MONEY TO YOU.
BUT IF THE RESULT IS NEGATIVE, THAT MEANS YOU TRIED TO WITHDRAW MORE MONEY THAN WHAT WAS AVAILABLE...
SO DEPRESSING...
AND YOU'LL GET A MESSAGE SAYING THAT YOU HAVE AN INSUFFICIENT BALANCE SO YOU CAN'T WITHDRAW THAT AMOUNT.
THE OPERATION RESULT WAS POSITIVE.
THE OPERATION RESULT WAS NEGATIVE.
PLEASE TAKE YOUR MONEY.
INSUFFICIENT BALANCE
INSUFFICIENT BALANCE
ACK...
THAT'S IT! IN OTHER WORDS, THE STATUS OUTPUT WILL TELL YOU IF YOU HAVE ENOUGH MONEY IN YOUR ACCOUNT TO MAKE A WITHDRAWAL.
THE CPU WILL MAKE A DECISION BASED ON WHETHER THE RESULT OF THE SUBTRACTION WAS POSITIVE OR NEGATIVE AND CHANGE ITS BEHAVIOR ACCORDINGLY.


THAT'S HOW THE CPU IS ABLE TO PROCESS OPERATIONS AS WELL AS MAKE DECISIONS!
OPERATION
DECISION
CPU
OKAY! SO IF YOU GIVE THE CPU A PROGRAM, IT WILL BE ABLE TO PROCESS INSTRUCTIONS AND MAKE DECISIONS.
AND BY REPEATING THAT PROCESS, COMPUTERS ARE ABLE TO PERFORM ALL KINDS OF TASKS, RIGHT?
YES, COMPUTERS ARE ABLE TO PERFORM CERTAIN TASKS FASTER THAN HUMANS, PROCESS AMOUNTS OF INFORMATION FAR TOO GREAT FOR HUMANS TO COMPREHEND, AND SURPASS HUMANS IN MANY DIFFERENT SKILLS.
HEH HEH HEH
LIKE SHOGI, FOR EXAMPLE... NO OFFENSE...
I SEE! I FEEL LIKE I UNDERSTAND THE CPU A BIT BETTER NOW.
HMM, BUT I BET THERE'S A LOT I STILL DON'T KNOW.
OH?

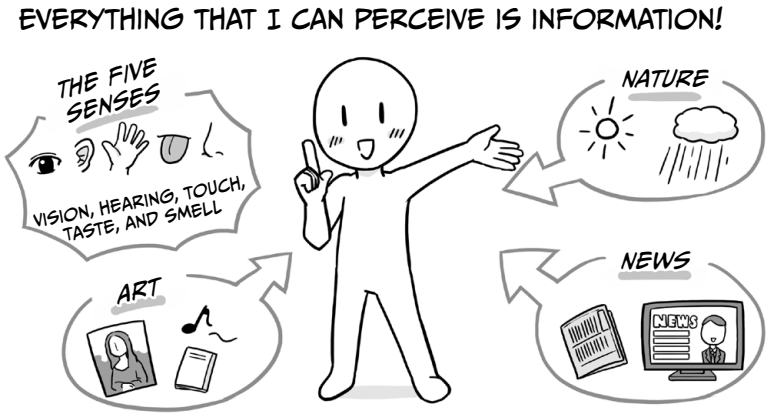

I GOTTA STUDY MORE!
OF COURSE, I WAS A LITTLE FRUSTRATED THAT I LOST AT SHOGI...
NO, I WAS SUPER FRUSTRATED! I'M PROBABLY SO MAD I WON'T BE ABLE TO SLEEP TONIGHT!!!
UGAAA——!
BUT I SEE THE CPU THAT BEAT ME AS MY RIVAL!
AND I WANT TO LEARN EVERYTHING I CAN ABOUT MY NEW RIVAL!
WELL, I HAVE TO ADMIT I FIND IT PRETTY FUN TO EXPLAIN COMPUTERS TO OTHER PEOPLE...
I... I MEAN, I SUPPOSE I FIND IT AMUSING TO EDUCATE THE IGNORANT MASSES FROM TIME TO TIME. *FUHAHAHAHAHA!*
OH, I SEE! I THINK I FINALLY GET WHY YOU'RE SO ENTHUSIASTIC ABOUT EXPLAINING THIS TO ME!
...?


YOU DON'T HAVE ANY FRIENDS, RIGHT? I'M RIGHT, AREN'T I?
YOU'RE JUST HAPPY TO FINALLY HAVE SOMEONE TO TALK TO, RIGHT? I UNDERSTAND THAT IT'S FUN TO HOLE UP AND WRITE PROGRAMS ALL BY YOUR LONESOME, BUT IT'S IMPORTANT TO TALK TO PEOPLE, TOO...
ARE YOU PITYING ME?!
IT'S OKAY—THIS IS A GREAT OPPORTUNITY FOR YOU! TEACH ME ABOUT CPUs!
OR I SHOULD SAY... IF YOU DON'T, YOU AREN'T GETTING THIS BACK...
HEHEHE...
DON'T TAKE IT HOSTAGE!!
I'LL TAKE THAT AS A YES! LET'S GET STUDYING!
HEY! DON'T PUT WORDS IN MY MOUTH!! I'VE GOT A LIFE TOO!

WHAT IS INFORMATION ANYWAY?

Information technology (IT) became an everyday phrase back in the 20th century. The term is frequently heard when people talk about the internet and other computer technology, but it's worth noting that this term predates the use of computers.

First off, what does the word *information* actually mean?To put it simply, information is everything in our environment that can be registered with any of our five senses.

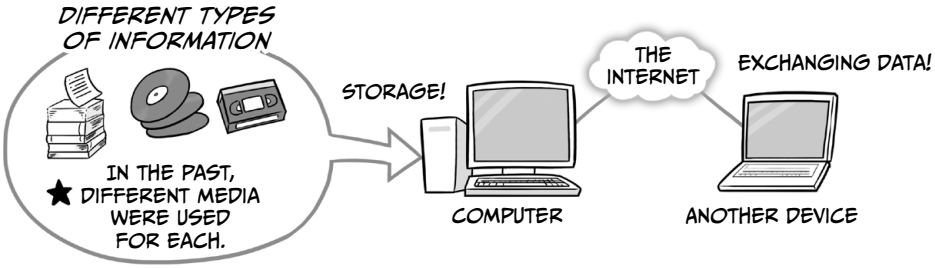


Everything that occurs in nature or in paintings, photographs, music, novels, news, radio, TV broadcasts, and so on is an example of information. Most of these things have been around for a lot longer than our society has had access to electricity. As information spreads throughout society, it affects our lives.

Every day, people and organizations value useful information while trying to filter out everything else. Information that is not important is called *noise*, and important information is called *signal*. Finding ways to maximize the *signal-to-noise ratio*—that is, the amount of signal in an output compared to the amount of noise—without accidentally losing necessary information is important.

One type of information that historically has been important both to people and organizations is information about food—what's safe or healthy to eat, how to find or grow it, and how far away it is or how much it costs to buy. Related information, such as climate and weather forecasts, is also vital. Obviously, information like this was valued long before the rise of the internet. For example, merchants like Bunzaemon Kinokuniya from Japan's Edo period specialized in products such as citrus and salmon and thrived because they valued this type of information. Indeed, the value of information has been respected for as long as people have needed to eat.

However, the digital age has affected many aspects of life. How has it affected our access to information? Well, thanks to the digitization of data, we are now able to process diverse data like text, audio, images, and video using the same methods. It can all be transmitted the same way (over the internet, for example) and stored in the same media (on hard drives, for example).



Computers that are connected to the same network can exchange digitized information. By using computers to match and analyze large sets of data instead of analyzing each instance or type of data individually, people can discover otherwise hidden trends or implications of the information.

Like the storage of data, information transmission has made incredible advances, thanks to important discoveries in electronics and electrical engineering. Commercial applications of this technology in devices such as telephones, radio, and television have played a role in accelerating this development. Today, almost all of Japan enjoys digital television, which uses digital transmission and compression technologies. CPUs play a central part in these applications by performing countless operations and coordinating the transfer of information.

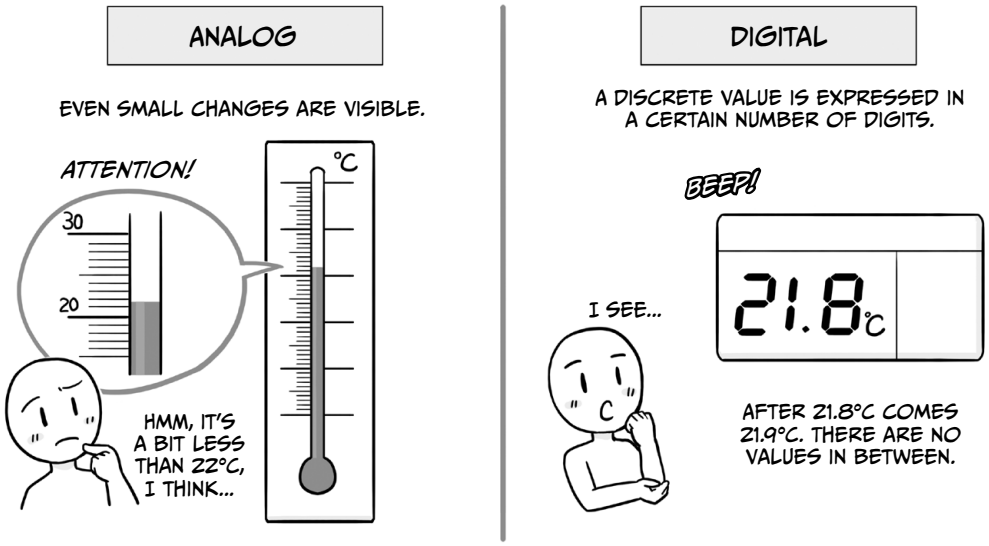
THE DIFFERENCE BETWEEN ANALOG AND DIGITAL INFORMATION

We have been talking about digitizing data into 1s and 0s so that information can be processed by a CPU. But before they are digitized, text, audio, video, and so on exist as analog data.

What is the difference between these two types of data? An example that illustrates the difference is thermometers. Analog thermometers contain a liquid that expands as it heats up, such as mercury or alcohol, in a gradated capillary tube that is marked with lines indicating the temperature. To determine the temperature, we look at the level of the liquid in the tube and compare it to the markings on the tube. We say that the analog thermometer has a *continuous* output because the temperature reading can fall anywhere between the marks on the tube.

Digital thermometers use a sensor to convert temperature into voltage* and then estimate the corresponding temperature. Because the temperature is represented numerically, the temperature changes in steps (that is, the values "jump"). For instance, if the initial temperature reading is 21.8 degrees Celsius and then the temperature increases, the next possible reading is 21.9 degrees Celsius. Because 0.1 is the smallest quantity that can be shown by this thermometer, changes in temperature can only be represented in steps of 0.1 and the value could never be between 21.8 and 21.9 degrees. Thus, digital output is said to be *discrete*.

* *Voltage* is a way of measuring electric currents and is expressed in volts.



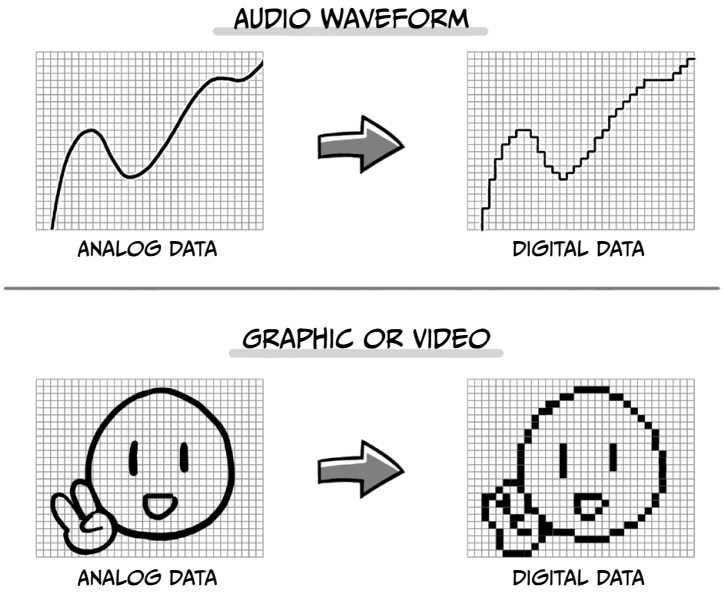
The word *digital* comes from the act of counting off numbers using our fingers—or digits. This tends to lead people to believe that digital computers can only work with data comprised of integers (whole numbers), which is not necessarily true.

In the digital world, everything is expressed in 1s and 0s. Indeed, they are not even what the CPU works with. Note that these are not actually numbers in this context. Instead, a 1 and a 0 are merely symbols. The CPU consists of transistors that transmit or inhibit electrical signals and consequently output either low or high voltages. It is these voltages that we represent as 1 or 0. A high voltage is represented with a 1, since the transistor's state is "on," and a low voltage, or an "off" transistor, is represented with a 0. In text, you could illustrate this by using the symbols ● and ○. The 1s and 0s are called *primitives*, meaning they are basic data types. Computers can work with decimal numbers as long as the value has a finite number of digits. Values such as these are also digital. The important thing to remember is that for any digital number, you can never add or remove a quantity smaller than the smallest possible value expressible.

Let's compare some analog data and its digitized version to better understand how they are alike and how they differ by looking at the figure on the next page. The first pair of images shows audio data, and the second pair shows image data.

As you can see, every time we translate analog data into digital data, some information is lost. But as you've undoubtedly experienced, most modern digitization processes are so good that humans can't tell the difference between the original and the digital copy, even when they are presented side by side.

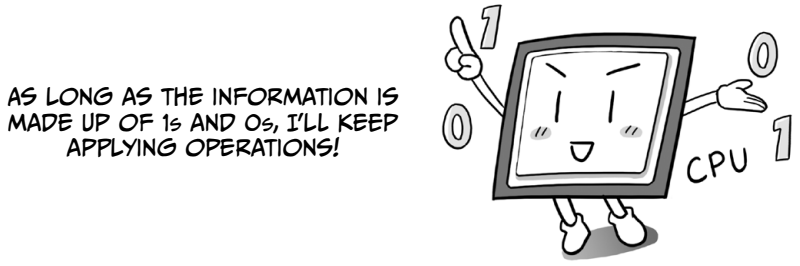
To store and transmit digital data of a quality such that our senses can't detect any loss of information, we use special compression techniques. These techniques always involve trade-offs among how much space is used, how much information is lost during compression, and how much processing time is needed to compress and decompress the data.



When color information is translated into digital form, it is split into three base component colors, most often red, green, and blue (known as RGB). These colors are combined to create a composite color on a screen. Each component color can be represented by a number, with larger numbers indicating there's more of that color.

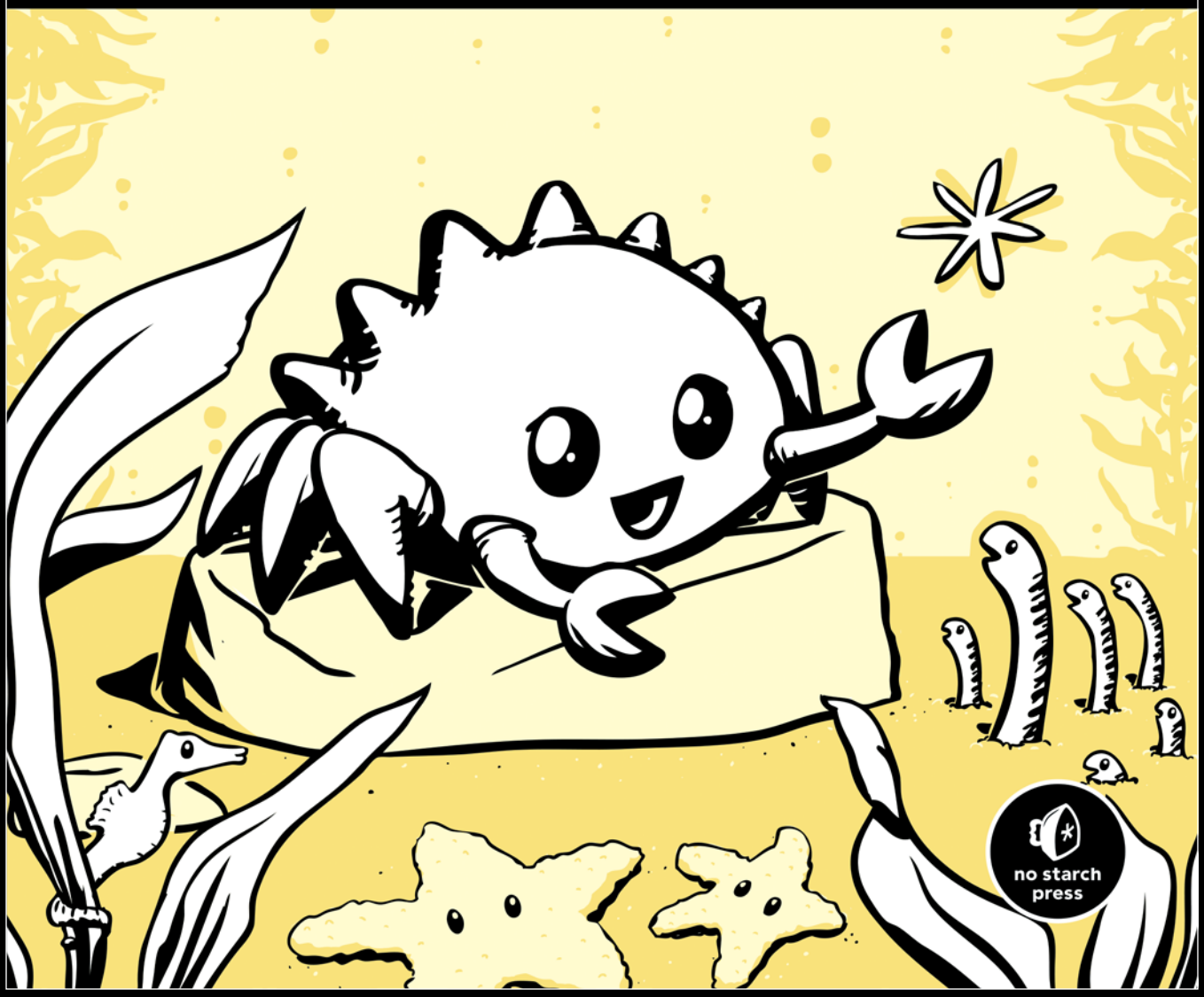
When heavily compressing audio or video data, we often use *lossy* techniques that change and simplify the data in such a way that we usually do not notice a difference. While this approach saves a lot of space, as the name implies, reconstructing the original data perfectly is impossible since vital information is missing. Other techniques—most notably all text compression techniques—use *lossless* compression, which guarantees that the original data can be completely reconstructed.

In any case, with the appropriate arithmetic and logic operations, as long as the data is digital, a CPU can use any compression technique on any form of information. Although digitizing data can involve the loss of some information, a major advantage of digital data over analog data is that it allows us to control noise when transmitting the data.



THE RUST PROGRAMMING LANGUAGE

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



2

GUESSING GAME

Let's jump into Rust by working through a hands-on project together! This chapter introduces you to a few common Rust concepts by showing you how to use them in a real program. You'll learn about `let`, `match`, methods, associated functions, using external crates, and more! The following chapters will explore these ideas in more detail. In this chapter, you'll practice the fundamentals.

We'll implement a classic beginner programming problem: a guessing game. Here's how it works: the program will generate a random integer between 1 and 100. It will then prompt the player to enter a guess. After entering a guess, it will indicate whether the guess is too low or too high. If the guess is correct, the game will print congratulations and exit.

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*.

```
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 job of `read_line` is to take whatever the user types into standard input and place that into a string, so it takes that string as an argument. The string argument needs to be mutable so the method can change the string's content by adding the user input.

The `&` indicates that this argument is a *reference*, which gives you a way to let multiple parts of your code access one piece of data without needing to copy that data into memory multiple times. References are a complex feature, and one of Rust's major advantages is how safe and easy it is to use references. You don't need to know a lot of those details to finish this program: Chapter 4 will explain references more thoroughly. For now, all you need to know is that like variables, references are immutable by default. Hence, you need to write `&mut guess` rather than `&guess` to make it mutable.

We're not quite done with this line of code. Although it's a single line of text, it's only the first part of the single logical line of code. The second part is this method:

```
.expect("Failed to read line");
```

When you call a method with the `.foo()` syntax, it's often wise to introduce a newline and other whitespace to help break up long lines. We could have written this code as:

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

However, one long line is difficult to read, so it's best to divide it: two lines for two method calls. Now let's discuss what this line does.

Handling Potential Failure with the Result Type

As mentioned earlier, `read_line` puts what the user types into the string we're passing it, but it also returns a value—in this case, an `io::Result`. Rust has a number of types named `Result` in its standard library: a generic `Result` as well as specific versions for submodules, such as `io::Result`.

The `Result` types are *enumerations*, often referred to as *enums*. An enumeration is a type that can have a fixed set of values, and those values are called the enum's *variants*. Chapter 6 will cover enums in more detail.

For `Result`, the variants are `Ok` or `Err`. `Ok` indicates the operation was successful, and inside the `Ok` variant is the successfully generated value. `Err` means the operation failed, and `Err` contains information about how or why the operation failed.

The purpose of these `Result` types is to encode error handling information. Values of the `Result` type, like any type, have methods defined on them. An instance of `io::Result` has an `expect` method that you can call. If this instance of `io::Result` is an `Err` value, `expect` will cause the program to crash and display the message that you passed as an argument to `expect`. If the `read_line` method returns an `Err`, it would likely be the result of an error coming from the underlying operating system. If this instance of `io::Result`

is an `Ok` value, `expect` will take the return value that `Ok` is holding and return just that value to you so you can use it. In this case, that value is the number of characters the user entered into standard input.

If you don't call `expect`, the program will compile, but you'll get a warning:

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
src/main.rs:10:5: 10:39 warning: unused result which must be used,
#[warn(unused_must_use)] on by default
src/main.rs:10     io::stdin().read_line(&mut guess);
                   ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Rust warns that you haven't used the `Result` value returned from `read_line`, indicating that the program hasn't handled a possible error.

The right way to suppress the warning is to actually write error handling, but since you just want to crash this program when a problem occurs, you can use `expect`. You'll learn about recovering from errors in Chapter 9.

Printing Values with println! Placeholders

Aside from the closing curly braces, there's only one more line to discuss in the code added so far, which is the following:

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

This line prints out the string we saved the user's input in. The set of {} is a placeholder: think of {} as little crab pincers that hold a value in place. You can print more than one value using {}: the first set of {} holds the first value listed after the format string, the second set holds the second value, and so on. Printing out multiple values in one call to `println!` would look like this:

```
let x = 5;
let y = 10;

println!("x = {} and y = {}", x, y);
```

This code would print out `x = 5 and y = 10`.

Testing the First Part

Let's test the first part of the guessing game. You can run it using **`cargo run`**:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
     Running `target/debug/guessing_game`
Guess the number!
Please input your guess.
6
You guessed: 6
```

At this point, the first part of the game is done: we're getting input from the keyboard and then printing it.

Generating a Secret Number

Next, we need to generate a secret number that the user will try to guess. The secret number should be different every time so the game is fun to play more than once. Let's use a random number between 1 and 100 so the game isn't too difficult. Rust doesn't yet include random number functionality in its standard library. However, the Rust team does provide a `rand` crate at *https://crates.io/crates/rand*.

Using a Crate to Get More Functionality

Remember that a *crate* is a package of Rust code. The project we've been building is a *binary crate*, which is an executable. The `rand` crate is a *library crate*, which contains code intended to be used in other programs.

Cargo's use of external crates is where it really shines. Before we can write code that uses `rand`, we need to modify the *Cargo.toml* file to include the `rand` crate as a dependency. Open that file now and add the following line to the bottom beneath the `[dependencies]` section header that Cargo created for you:

Filename: Cargo.toml

```
[dependencies]

rand = "0.3.14"
```

In the *Cargo.toml* file, everything that follows a header is part of a section that continues until another section starts. The `[dependencies]` section is where you tell Cargo which external crates your project depends on and which versions of those crates you require. In this case, we'll specify the `rand` crate with the semantic version specifier `0.3.14`. Cargo understands Semantic Versioning (sometimes called *SemVer*), which is a standard for writing version numbers. The number `0.3.14` is actually shorthand for `^0.3.14`, which means "any version that has a public API compatible with version 0.3.14."

Now, without changing any of the code, let's build the project, as shown in Listing 2-2:

```
$ cargo build
    Updating registry `https://github.com/rust-lang/crates.io-index`
 Downloading rand v0.3.14
 Downloading libc v0.2.14
   Compiling libc v0.2.14
   Compiling rand v0.3.14
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
```

Listing 2-2: The output from running `cargo build` after adding the `rand` crate as a dependency

You may see different version numbers (but they will all be compatible with the code, thanks to SemVer!), and the lines may be in a different order.

Now that we have an external dependency, Cargo fetches the latest versions of everything from the *registry*, which is a copy of data from https://crates.io. Crates.io is where people in the Rust ecosystem post their open source Rust projects for others to use.

After updating the registry, Cargo checks the `[dependencies]` section and downloads any you don't have yet. In this case, although we only listed `rand` as a dependency, Cargo also grabbed a copy of `libc`, because `rand` depends on `libc` to work. After downloading them, Rust compiles them and then compiles the project with the dependencies available.

If you immediately run `cargo build` again without making any changes, you won't get any output. Cargo knows it has already downloaded and compiled the dependencies, and you haven't changed anything about them in your *Cargo.toml* file. Cargo also knows that you haven't changed anything about your code, so it doesn't recompile that either. With nothing to do, it simply exits. If you open up the *src/main.rs* file, make a trivial change, and then save it and build again, you'll only see one line of output:

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
```

This line shows Cargo only updates the build with your tiny change to the *src/main.rs* file. Your dependencies haven't changed, so Cargo knows it can reuse what it has already downloaded and compiled for those. It just rebuilds your part of the code.

The Cargo.lock File Ensures Reproducible Builds

Cargo has a mechanism that ensures you can rebuild the same artifact every time you or anyone else builds your code: Cargo will use only the versions of the dependencies you specified until you indicate otherwise. For example, what happens if next week version `v0.3.15` of the `rand` crate comes out and contains an important bug fix but also contains a regression that will break your code?

The answer to this problem is the *Cargo.lock* file, which was created the first time you ran `cargo build` and is now in your *guessing_game* directory. When you build a project for the first time, Cargo figures out all the versions of the dependencies that fit the criteria and then writes them to the *Cargo.lock* file. When you build your project in the future, Cargo will see that the *Cargo.lock* file exists and use the versions specified there rather than doing all the work of figuring out versions again. This lets you have a reproducible build automatically. In other words, your project will remain at `0.3.14` until you explicitly upgrade, thanks to the *Cargo.lock* file.

Updating a Crate to Get a New Version

When you *do* want to update a crate, Cargo provides another command, `update`, which will:

1. Ignore the *Cargo.lock* file and figure out all the latest versions that fit your specifications in *Cargo.toml*.
2. If that works, Cargo will write those versions to the *Cargo.lock* file.

But by default, Cargo will only look for versions larger than `0.3.0` and smaller than `0.4.0`. If the `rand` crate has released two new versions, `0.3.15` and `0.4.0`, you would see the following if you ran `cargo update`:

```
$ cargo update
    Updating registry `https://github.com/rust-lang/crates.io-index`
    Updating rand v0.3.14 -> v0.3.15
```

At this point, you would also notice a change in your *Cargo.lock* file noting that the version of the `rand` crate you are now using is `0.3.15`.

If you wanted to use `rand` version `0.4.0` or any version in the `0.4.x` series, you'd have to update the *Cargo.toml* file to look like this instead:

```
[dependencies]

rand = "0.4.0"
```

The next time you run `cargo build`, Cargo will update the registry of crates available and reevaluate your `rand` requirements according to the new version you specified.

There's a lot more to say about Cargo and its ecosystem that we'll discuss in Chapter XX, but for now, that's all you need to know. Cargo makes it very easy to reuse libraries, so Rustaceans are able to write smaller projects that are assembled from a number of packages.

Generating a Random Number

Let's start *using* `rand`. The next step is to update *src/main.rs*, as shown in Listing 2-3:

Filename: src/main.rs

```
❶ extern crate rand;

  use std::io;
❷ use rand::Rng;

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

```
❸     let secret_number = rand::thread_rng().gen_range(1, 101);

      println!("The secret number is: {}", secret_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 2-3: Code changes needed in order to generate a random number

First, we add a line to the top ❶ that lets Rust know we'll be using that external dependency. This also does the equivalent of calling `use rand`, so now we can call anything in the `rand` crate by prefixing it with `rand::`.

Next, we add another `use` line: `use rand::Rng` ❷. `Rng` is a trait that defines methods that random number generators implement, and this trait must be in scope for us to use those methods. Chapter 10 will cover traits in detail.

Also, we're adding two more lines in the middle ❸. The `rand::thread_rng` function will give us the particular random number generator that we're going to use: one that is local to the current thread of execution and seeded by the operating system. Next, we call the `gen_range` method on the random number generator. This method is defined by the `Rng` trait that we brought into scope with the `use rand::Rng` statement. The `gen_range` method takes two numbers as arguments and generates a random number between them. It's inclusive on the lower bound but exclusive on the upper bound, so we need to specify `1` and `101` to request a number between 1 and 100.

Knowing which traits to import and which functions and methods to use from a crate isn't something that you'll just *know*. Instructions for using a crate are in each crate's documentation. Another neat feature of Cargo is that you can run the `cargo doc --open` command, which will build documentation provided by all of your dependencies locally and open it in your browser. If you're interested in other functionality in the `rand` crate, for example, run `cargo doc --open` and click **rand** in the sidebar on the left.

The second line that we added to the code prints the secret number. This is useful while we're developing the program to be able to test it, but we'll delete it from the final version. It's not much of a game if the program prints the answer as soon as it starts!

Try running the program a few times:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
     Running `target/debug/guessing_game`
Guess the number!
The secret number is: 7
Please input your guess.
```

```
4
You guessed: 4
$ cargo run
     Running `target/debug/guessing_game`
Guess the number!
The secret number is: 83
Please input your guess.
5
You guessed: 5
```

You should get different random numbers, and they should all be numbers between 1 and 100. Great job!

Comparing the Guess to the Secret Number

Now that we have user input and a random number, we can compare them. That step is shown in Listing 2-4:

```
extern crate rand;

use std::io;
❶ use std::cmp::Ordering;
use rand::Rng;

fn main() {

---snip---

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

    match❷ guess.cmp(&secret_number)❸ {
        Ordering::Less    => println!("Too small!"),
        Ordering::Greater => println!("Too big!"),
        Ordering::Equal   => println!("You win!"),
    }
}
```

Listing 2-4: Handling the possible return values of comparing two numbers

The first new bit here is another `use` ❶, bringing a type called `std::cmp::Ordering` into scope from the standard library. `Ordering` is another enum, like `Result`, but the variants for `Ordering` are `Less`, `Greater`, and `Equal`. These are the three outcomes that are possible when you compare two values.

Then we add five new lines at the bottom that use the `Ordering` type. The `cmp` method ❸ compares two values and can be called on anything that can be compared. It takes a reference to whatever you want to compare with: here it's comparing the `guess` to the `secret_number`. `cmp` returns a variant of the `Ordering` enum we imported with the `use` statement. We use

a `match` expression ❷ to decide what to do next based on which variant of `Ordering` was returned from the call to `cmp` with the values in `guess` and `secret_number`.

A `match` expression is made up of *arms*. An arm consists of a *pattern* and the code that should be run if the value given to the beginning of the `match` expression fits that arm's pattern. Rust takes the value given to `match` and looks through each arm's pattern in turn. The `match` construct and patterns are powerful features in Rust that let you express a variety of situations your code might encounter and help ensure that you handle them all. These features will be covered in detail in Chapter 6 and Chapter XX, respectively.

Let's walk through an example of what would happen with the `match` expression used here. Say that the user has guessed 50, and the randomly generated secret number this time is 38. When the code compares 50 to 38, the `cmp` method will return `Ordering::Greater`, because 50 is greater than 38. `Ordering::Greater` is the value that the `match` expression gets. It looks at the first arm's pattern, `Ordering::Less`, and sees that the value `Ordering::Greater` does not match `Ordering::Less`. So it ignores the code in that arm and moves to the next arm. The next arm's pattern, `Ordering::Greater`, *does* match `Ordering::Greater`! The associated code in that arm will execute and print `Too big!` to the screen. The `match` expression ends because it has no need to look at the last arm in this particular scenario.

However, the code in Listing 2-4 won't compile yet. Let's try it:

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
error[E0308]: mismatched types
  --> src/main.rs:23:21
   |
23 |     match guess.cmp(&secret_number) {
   |                     ^^^^^^^^^^^^^^^ expected struct `std::string::String`,
found integral variable
   |
   = note: expected type `&std::string::String`
   = note:    found type `&{integer}`

error: aborting due to previous error
Could not compile `guessing_game`.
```

The core of the error states that there are *mismatched types*. Rust has a strong, static type system. However, it also has type inference. When we wrote `let guess = String::new()`, Rust was able to infer that `guess` should be a `String` and didn't make us write the type. The `secret_number`, on the other hand, is a number type. A few number types can have a value between 1 and 100: `i32`, a 32-bit number; `u32`, an unsigned 32-bit number; `i64`, a 64-bit number; as well as others. Rust defaults to an `i32`, which is the type of `secret_number` unless you add type information elsewhere that would cause Rust to infer a different numerical type. The reason for the error is that Rust will not compare a string and a number type.

Ultimately, we want to convert the `String` the program reads as input into a real number type so we can compare it to the guess numerically. We can do that by adding the following two lines to the `main` function body:

```
---snip---

    let mut guess = String::new();

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

    let guess: u32 = guess.trim().parse()
        .expect("Please type a number!");

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

    match guess.cmp(&secret_number) {
        Ordering::Less    => println!("Too small!"),
        Ordering::Greater => println!("Too big!"),
        Ordering::Equal   => println!("You win!"),
    }
}
```

We create a variable named `guess`. But wait, doesn't the program already have a variable named `guess`? It does, but Rust allows us to *shadow* the previous value of `guess` with a new one. This feature is often used in similar situations in which you want to convert a value from one type to another type. Shadowing lets us reuse the `guess` variable name rather than forcing us to create two unique variables, like `guess_str` and `guess` for example. (Chapter 3 covers shadowing in more detail.)

We bind `guess` to the expression `guess.trim().parse()`. The `guess` in the expression refers to the original `guess` that was a `String` with the input in it. The `trim` method on a `String` instance will eliminate any whitespace at the beginning and end. `u32` can only contain numerical characters, but the user must press the ENTER key to satisfy `read_line`. When the user presses ENTER, a newline character is added to the string. For example, if the user types `5` and presses ENTER, `guess` looks like this: `5\n`. The `\n` represents "newline," the return key. The `trim` method eliminates `\n`, resulting in just `5`.

The `parse` method on strings parses a string into some kind of number. Because this method can parse a variety of number types, we need to tell Rust the exact number type we want by using `let guess: u32`. The colon (`:`) after `guess` tells Rust we'll annotate the variable's type. Rust has a few built-in number types; the `u32` seen here is an unsigned, 32-bit integer. It's a good default choice for a small positive number. You'll learn about other number types in Chapter 3. Additionally, the `u32` annotation in this example program and the comparison with `secret_number` means that Rust will infer that `secret_number` should be a `u32` as well. So now the comparison will be between two values of the same type!

The call to `parse` could easily cause an error. If, for example, the string contained `A👍%`, there would be no way to convert that to a number. Because it might fail, the `parse` method returns a `Result` type, much like the `read_line` method does as discussed earlier in "Handling Potential Failure with the Result Type" on page XX. We'll treat this `Result` the same way by using the `expect` method again. If `parse` returns an `Err Result` variant because it couldn't create a number from the string, the `expect` call will crash the game and print the message we give it. If `parse` can successfully convert the string to a number, it will return the `Ok` variant of `Result`, and `expect` will return the number that we want from the `Ok` value.

Let's run the program now!

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
     Running `target/guessing_game`
Guess the number!
The secret number is: 58
Please input your guess.
  76
You guessed: 76
Too big!
```

Nice! Even though spaces were added before the guess, the program still figured out that the user guessed 76. Run the program a few times to verify the different behavior with different kinds of input: guess the number correctly, guess a number that is too high, and guess a number that is too low.

We have most of the game working now, but the user can make only one guess. Let's change that by adding a loop!

Allowing Multiple Guesses with Looping

The `loop` keyword gives us an infinite loop. We'll add that now to give users more chances at guessing the number:

```
---snip---

    println!("The secret number is: {}", secret_number);

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

        ---snip---

        match guess.cmp(&secret_number) {
            Ordering::Less    => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal   => println!("You win!"),
        }
    }
}
```

As you can see, we've moved everything into a loop from the guess input prompt onward. Be sure to indent those lines another four spaces each, and run the program again. Notice that there is a new problem because the program is doing exactly what we told it to do: ask for another guess forever! It doesn't seem like the user can quit!

The user could always halt the program by using the keyboard shortcut CTRL-C. But there's another way to escape this insatiable monster, as mentioned in the `parse` discussion in "Comparing the Guess to the Secret Number" on page 11: if the user enters a non-number answer, the program will crash. The user can take advantage of that in order to quit, as shown here:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
     Running `target/guessing_game`
Guess the number!
The secret number is: 59
Please input your guess.
45
You guessed: 45
Too small!
Please input your guess.
60
You guessed: 60
Too big!
Please input your guess.
59
You guessed: 59
You win!
Please input your guess.
quit
thread 'main' panicked at 'Please type a number!: ParseIntError { kind:
InvalidDigit }', src/libcore/result.rs:785
note: Run with `RUST_BACKTRACE=1` for a backtrace.
error: Process didn't exit successfully: `target/debug/guess` (exit code: 101)
```

Typing `quit` actually quits the game, but so will any other non-number input. However, this is suboptimal to say the least. We want the game to automatically stop when the correct number is guessed.

Quitting After a Correct Guess

Let's program the game to quit when the user wins by adding a `break`:

```
---snip---

        match guess.cmp(&secret_number) {
            Ordering::Less    => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal   => {
                println!("You win!");
```

```
                break;
            }
        }
    }
}
```

By adding the `break` line after `You win!`, the program will exit the loop when the user guesses the secret number correctly. Exiting the loop also means exiting the program, because the loop is the last part of `main`.

Handling Invalid Input

To further refine the game's behavior, rather than crashing the program when the user inputs a non-number, let's make the game ignore a non-number so the user can continue guessing. We can do that by altering the line where `guess` is converted from a `String` to a `u32`:

```
---snip---

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

let guess: u32 = match guess.trim().parse() {
    Ok(num) => num,
    Err(_) => continue,
};

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

---snip---
```

Switching from an `expect` call to a `match` expression is how you generally move from crash on error to actually handling the error. Remember that `parse` returns a `Result` type, and `Result` is an enum that has the variants `Ok` or `Err`. We're using a `match` expression here, like we did with the `Ordering` result of the `cmp` method.

If `parse` is able to successfully turn the string into a number, it will return an `Ok` value that contains the resulting number. That `Ok` value will match the first arm's pattern, and the `match` expression will just return the `num` value that `parse` produced and put inside the `Ok` value. That number will end up right where we want it in the new `guess` variable we're creating.

If `parse` is *not* able to turn the string into a number, it will return an `Err` value that contains more information about the error. The `Err` value does not match the `Ok(num)` pattern in the first `match` arm, but it does match the `Err(_)` pattern in the second arm. The `_` is a catchall value; in this example, we're saying we want to match all `Err` values, no matter what information they have inside them. So the program will execute the second arm's code, `continue`, which means to go to the next iteration of the `loop` and ask for another guess. So effectively, the program ignores all errors that `parse` might encounter!

Now everything in the program should work as expected. Let's try it by running `cargo run`:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
     Running `target/guessing_game`
Guess the number!
The secret number is: 61
Please input your guess.
10
You guessed: 10
Too small!
Please input your guess.
99
You guessed: 99
Too big!
Please input your guess.
foo
Please input your guess.
61
You guessed: 61
You win!
```

Awesome! With one tiny final tweak, we will finish the guessing game: recall that the program is still printing out the secret number. That worked well for testing, but it ruins the game. Let's delete the `println!` that outputs the secret number. Listing 2-5 shows the final code:

```
extern crate rand;

use std::io;
use std::cmp::Ordering;
use rand::Rng;

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

    let secret_number = rand::thread_rng().gen_range(1, 101);

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

        let mut guess = String::new();

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

        let guess: u32 = match guess.trim().parse() {
            Ok(num) => num,
            Err(_) => continue,
        };

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

```
        match guess.cmp(&secret_number) {
            Ordering::Less    => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal   => {
                println!(“You win!”);
                break;
            }
        }
    }
}
```

Listing 2-5: Complete code of the guessing game

Summary

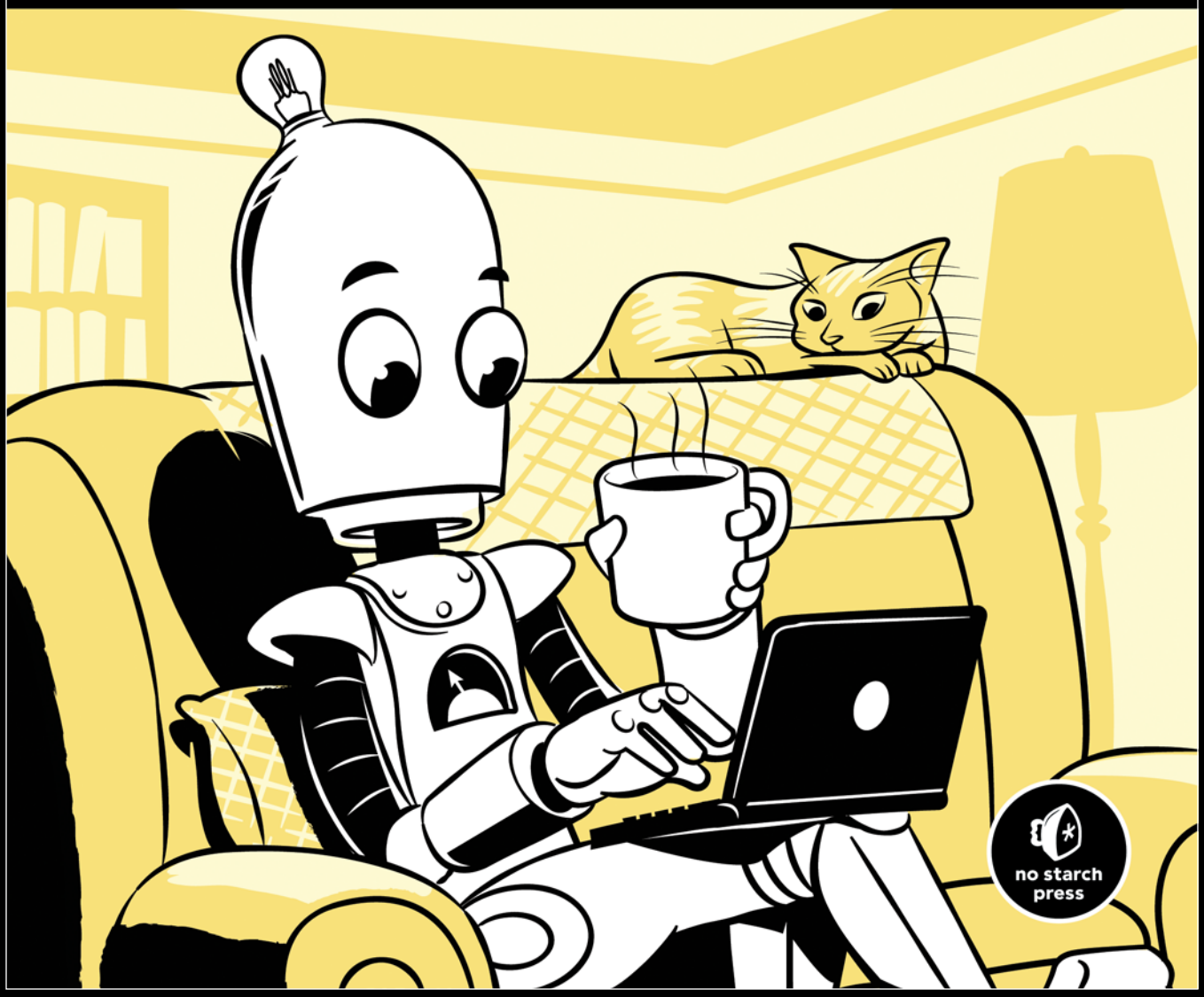
At this point, you've successfully built the guessing game! Congratulations!

This project was a hands-on way to introduce you to many new Rust concepts: `let`, `match`, methods, associated functions, using external crates, and more. In the next few chapters, you'll learn about these concepts in more detail. Chapter 3 covers concepts that most programming languages have, such as variables, data types, and functions, and shows how to use them in Rust. Chapter 4 explores ownership, which is a Rust feature that is most different from other languages. Chapter 5 discusses structs and method syntax, and Chapter 6 endeavors to explain enums.

LEARN JAVA THE EASY WAY

A *HANDS-ON* INTRODUCTION TO PROGRAMMING

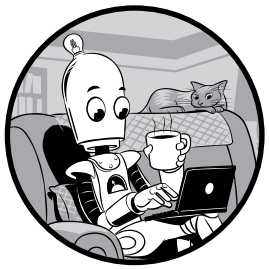
BRYSON PAYNE



no starch press

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.

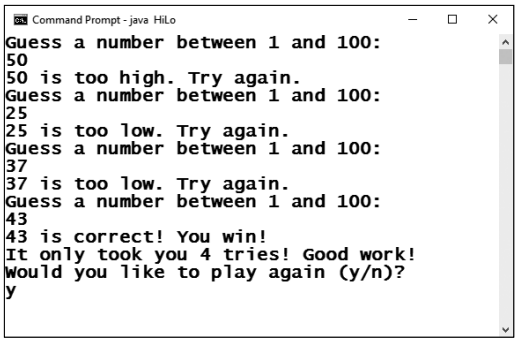


Figure 2-1: A text-based Hi-Lo guessing game

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 the guess is 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 on page 40, 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 New Java Project dialog should pop up, as shown in Figure 2-2.

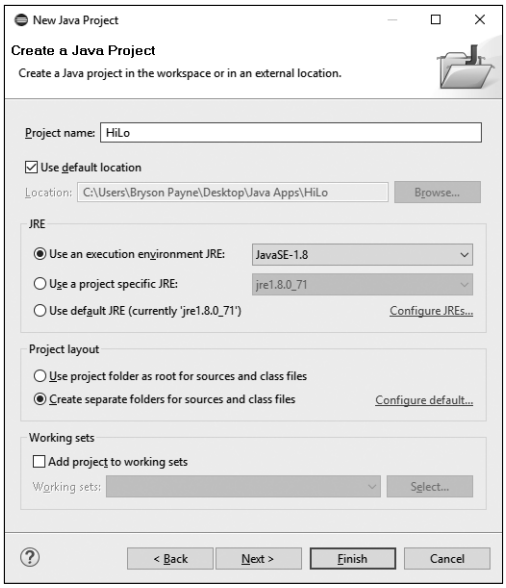


Figure 2-2: The New Java Project dialog for the Hi-Lo guessing game app

Type **`HiLo`** into the Project name field. 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. And we'll use camel case, as *Hi* and *Lo* are two words: *HiLo*. 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 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`.

To create the new `HiLo` class, first find your *HiLo* project folder under the Package Explorer pane on the left side of the Eclipse workspace. Expand the folder by clicking the small arrow to the left of it. You should see a subfolder called *src*, short for *source code*. All the text files containing your Java programs will go in this *src* folder.

Right-click the *src* folder and select **New ▸ Class**, as shown in Figure 2-3.

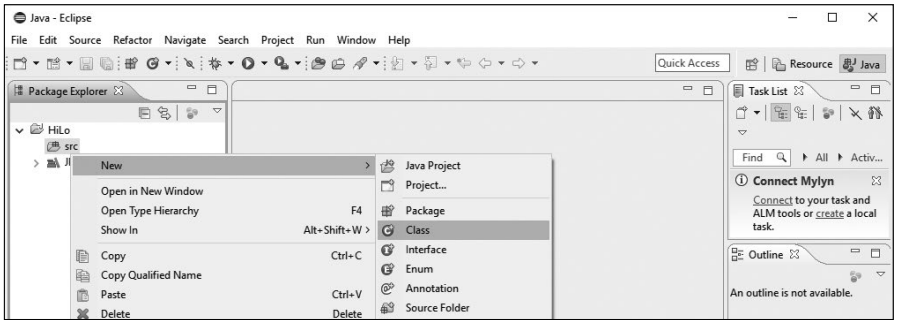


Figure 2-3: Creating a new class file for the Hi-Lo guessing game app

The New Java Class dialog will appear, as shown in Figure 2-4. Type **`HiLo`** into the Name field. Then, under *Which method stubs would you like to create?*, check the box for **public static void main(String[] args)**. This tells Eclipse that we're planning to write a `main()` program method, so Eclipse will include a *stub*, or skeleton, for the `main()` method that we can fill in with our own code. *Methods* are the functions in an object or class. The `main()` method is required any time you want to run an app as a stand-alone program.

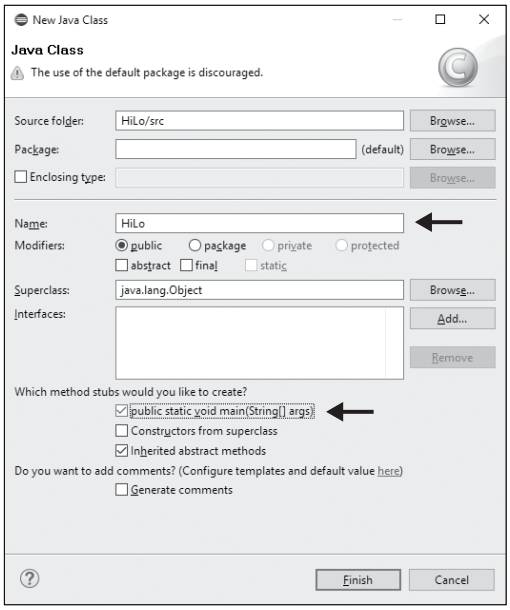


Figure 2-4: Name the new Java class HiLo *and select the checkbox to create a `main()` method.*

Click **Finish** in the New Java Class dialog, and you should see a new file named *HiLo.java* that contains the code shown in Listing 2-1. This Java file will be the outline of the Hi-Lo guessing game. We'll write the guessing game program by editing this file and adding code inside it.

```
❶ public class HiLo {
   ❷ public static void main(String[] args) {
         // TODO Auto-generated method stub

     }
}
```

Listing 2-1: The stub code for the `HiLo` guessing game class, generated by Eclipse

Eclipse creates this code all on its own. The class `HiLo` is `public` ❶, meaning we can run it from the command line or terminal.

Java groups statements with braces, { and }. The opening brace, {, begins a block of statements that will form the body of the `HiLo` class. The closing brace, }, ends the block of statements. Inside the class is the `main()` method ❷, which is the method that will run when the class is executed.

Inside the opening brace for the `main()` method is a comment line that starts with two forward slashes, `//`. Comments are for us (the humans) to read. They're ignored by the computer, so we can use them to help us remember what a section of code does or to leave notes for future use. You can delete the `TODO` comment in Listing 2-1.

Generating a Random Number

The first programming task for our game is to generate a random number. We'll use the `Math` class, which contains a method for generating a random floating-point (decimal) number between 0.0 and 1.0. Then, we'll convert that decimal value to an *integer* (a whole number) between 1 and 100. The `Math` class is a built-in class that contains many useful math functions like the ones you might find on a nice scientific calculator.

Inside the `main()` method, add the comment and line of code shown in Listing 2-2. (The new code is shown in black and the existing code is shown in gray.)

```
public class HiLo {
    public static void main(String[] args) {
        // Create a random number for the user to guess
        int theNumber = (int)(Math.random() * 100 + 1);
    }
}
```

Listing 2-2: The code to create a random number between 1 and 100

First, we need to create a variable to hold the random number the user will be trying to guess in the app. Since the app will ask the user to guess a whole number between 1 and 100, we'll use the `int` type, short for *integer*. We name our variable `theNumber`. The equal sign, `=`, assigns a value to our

new `theNumber` variable. We use the built-in `Math.random()` function to generate a random number between 0.0 and just under 1.0 (0.99999). Because `Math.random()` generates numbers only in that specific range, we need to multiply the random number we get by 100 to stretch the range from 0.0 to just under 100.0 (99.99999 or so). Then we add 1 to that value to ensure the number runs from 1.0 (0.0 + 1) to 100.99999.

The `(int)` part is called a *type cast*, or just *cast* for short. Casting changes the *type* of the number from a decimal number to an integer. In this case, everything after the decimal point is removed, resulting in a whole number between 1 and 100. Java then stores that number in the variable `theNumber`, the number the user is trying to guess in the game. Finally, we add a semicolon (`;`) to indicate the end of the instruction.

Now, you can add a `System.out.println()` statement to print the number you've generated:

```
        int theNumber = (int)(Math.random() * 100 + 1);
        System.out.println( theNumber );
    }
}
```

After we add this line of code, we can run the program to see it generate and print a random number. Click the green run button in the top menu bar to compile and run the program, as shown in Figure 2-5. You can also go to the **Run** menu and select **Run**.

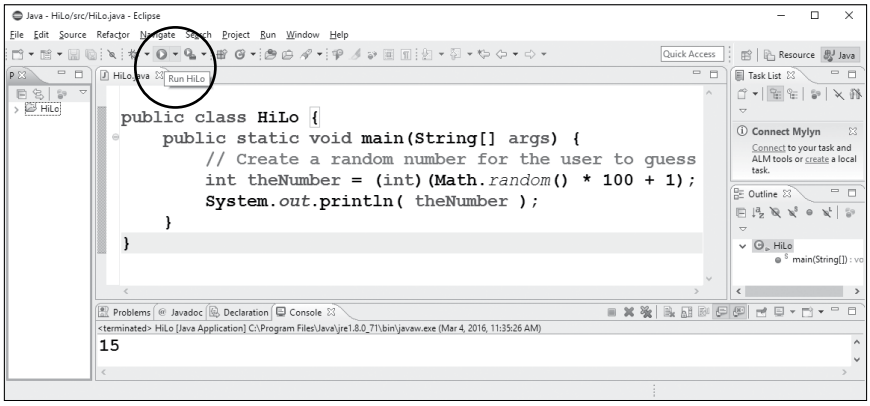


Figure 2-5: Printing a random number to the screen

Your random number will appear in the small console window at the bottom of the screen, as shown in Figure 2-5. If you run your program again, you'll see a different number between 1 and 100.

This would be a great time to play with the program a bit. Try generating a number between 1 and 10, or 1 and 1,000—even 1 to 1,000,000. Java will accept numbers all the way to a billion or so. Just remember to write your numbers without commas: 1,000 becomes `1000` in Java, and 1,000,000

is written `1000000`. You probably don't want to guess a number between 1 and 1,000,000 the first time you play the game, though, so remember to change this line back before you move ahead.

NOTE *Remember to save your code often. Eclipse will save for you automatically every time you run a program, but it's a good idea to save after every few lines of code. In fact, pressing CTRL-S (or ⌘-S) to save after each line of code isn't a bad habit to get into. I've never heard a coder say they wish they hadn't saved so often, but I've experienced losing unsaved code a few times myself, and it's not fun. Save often, and remember that you can use* ***Edit ▸ Undo*** *if you ever type something incorrectly or accidentally delete a section of code.*

Getting User Input from the Keyboard

Now let's add the code that allows the user to guess a number. To do this, we'll need to *import* some additional Java capabilities. Java comes with many libraries and packages that we can use in our own projects. Libraries and packages are sets of code that someone else has created. When we import them, we get new features that make creating our own programs even easier. We can access packages and libraries whenever we need them using the `import` statement.

For the guessing game program, we need to be able to accept keyboard input from the user. The `Scanner` class, contained in the `java.util` utilities package, provides several useful functions for working with keyboard input. Let's import the `Scanner` class into our program. Add the following statement at the top of the *HiLo.java* file, before the line `public class HiLo`:

```
import java.util.Scanner;

public class HiLo {
```

This line imports the `Scanner` class and all its functionality from the main Java utilities package. The `Scanner` class includes functions like `nextLine()` to accept a line of input from the keyboard and `nextInt()` to turn text input from the keyboard into an integer number that can be compared or used in calculations. To use the `Scanner` class for keyboard input, we have to tell it to use the keyboard as its source.

We want to do this before anything else in the program, so add this line of code inside the top of the `main()` method:

```
public class HiLo {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        // Create a random number for the user to guess
        int theNumber = (int)(Math.random() * 100 + 1);
```

This line creates a `Scanner` object called `scan` that pulls input from the computer's keyboard, `System.in`.

Although this new line of code sets up the `scan` object, it doesn't actually ask for input yet. To get the user to type in a guess, we'll need to prompt them by asking them to enter a number. Then, we'll take the number they enter from the keyboard and store it in a variable that we can compare against `theNumber`, the computer's original random number. Let's call the variable that will store the user's guess something easy to remember, like `guess`. Add the following line next:

```
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        // Create a random number for the user to guess
        int theNumber = (int)(Math.random() * 100 + 1);
        System.out.println( theNumber );
        int guess = 0;
```

This statement both *declares* a variable called `guess` of type `int` (an integer in Java), and it *initializes* the `guess` variable to a starting value of `0`. Some programming languages require a variable to be declared and then initialized in separate lines of code, but Java allows programmers to include both the declaration and initialization of variables in a single line. Java requires every variable to be declared with a specific *type*, or kind of information it should store. The user's guess will be a whole number, so we've declared `guess` as an `int`.

Next, we need to prompt the user to enter a guess. We can let the user know the program is ready for input by printing a line of text to the console window (or command line). We access this text-based screen as a part of our computer system through the `System` class, just like we did for keyboard input. But this time, we want to *output* information for the user to read. The object that lets us access the command line console for output is `System.out`. Similar to the `System.in` object that allows us to receive text input from the keyboard, `System.out` gives us the ability to output text to the screen. The specific function to print a line of text is the `println()` command:

```
        int guess = 0;
        System.out.println("Guess a number between 1 and 100:");
```

Here we are using *dot notation*, which lists a class or object, followed by a dot and then a method or an attribute of that class or object. Methods need to be called with dot notation to tell Java which object or class they belong to; for example, `Math.random()`. *Attributes* are the values stored in an object or class.

For example, `System` is a class representing your computer system. `System.out` is the command line screen object contained in the `System` class, because your computer monitor is part of your overall computer system. `System.out.println()` is a method to print a line of text using the `System.out` object. We'll get more practice using dot notation as we continue.

Now that the user knows what kind of input the program is expecting from them, it's time to check the keyboard for their guess. We'll use the `Scanner` object called `scan` that we created earlier. Scanners have a method

called `nextInt()` that looks for the next `int` value the user inputs from the keyboard. We'll store the user's guess in the variable `guess` that we created earlier:

```
        System.out.println("Guess a number between 1 and 100:");
        guess = scan.nextInt();
```

This statement will wait for the user to type something into the console window (hopefully a whole number between 1 and 100—we'll see how to make sure the user enters a valid number in Chapter 3) and press ENTER. The `nextInt()` method will take the string of text characters the user entered (`"50"`, for example), turn it into the correct numeric value (`50`), and then store that number in the variable `guess`. Take a moment to save the changes you've made so far.

Making the Program Print Output

We can also check to make sure our program is working so far by adding another `println()` statement:

```
        guess = scan.nextInt();
        System.out.println("You entered " + guess + ".");
```

This line uses the `System.out.println()` method again, but now we're combining text and numeric output. If the user guesses `50`, we want the output to read, `"You entered 50."` To make this happen, we form a `println()` statement that mixes text with the number stored in the variable `guess`.

Java allows us to concatenate strings of text using the `+` operator. We use double quotation marks to specify the text we want to output first (`"You entered "`). Note the space before the closing quotation marks—this tells th program that we want a space to appear in the printed output after the last word. Java ignores most spacing, but when a space is included inside the quotation marks of a string of text, it becomes part of that text.

We also want to print the number the user guessed. We've stored this value in the variable called `guess`, so we just have to use the `println()` statement to output that value. Fortunately, in Java, when you include a variable in a `println()` statement, Java prints the value contained in that variable. So, immediately after the text `"You entered "`, we add the concatenation operator (`+`) followed by the variable name `guess`. Finally, we want to end the sentence with a period, so we use another concatenation operator followed by the text we want, contained in double quotation marks, so it looks like `"."`.

Listing 2-3 puts together all of our lines of code so far.

```
import java.util.Scanner;

public class HiLo {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
```

```
        // Create a random number for the user to guess
        int theNumber = (int)(Math.random() * 100 + 1);
    ❶ // System.out.println( theNumber );
        int guess = 0;
        System.out.println("Guess a number between 1 and 100:");
        guess = scan.nextInt();
        System.out.println("You entered " + guess + ".");
    }
}
```

Listing 2-3: The code to this point generates a random number and allows the user to guess once.

At ❶, note that I turned `System.out.println( theNumber );` into a comment by adding a pair of forward slashes to the start of that line. This is called *commenting out*, and it's a useful technique for *debugging*—finding and fixing bugs or errors in programs. We used this `println()` statement earlier to show the value of the variable `theNumber` while we were writing and testing the program. Now, rather than deleting the line entirely, we can turn it into a comment so it's ignored by the computer. If we want to use that line again, we can just remove the `//` to include it in the program.

Now let's save our program and run it to see how it works so far. To run it, press the green run button or go to **Run ▸ Run**. Right now, the user can guess only once and the program doesn't check whether they guessed correctly. So next, we'll add some code so that the user can guess more than one time, and then we'll learn how to test each guess against `theNumber`.

Loops: Ask, Check, Repeat

To give the user more than one chance to guess the number, we need to learn how to build a loop! In the guessing game program, we need to ask the user for a guess until they guess correctly. *Loops* give us the ability to repeat a set of steps over and over. In this section, we'll build a loop for the steps that prompt the user for a guess and accept the keyboard input.

Loops are very powerful programming tools, and they're one of the reasons computers are so valuable in our daily lives and in the business world—computers are really good at repeating the same task predictably. And, if they're programmed correctly, they can do this all day, every day, without making mistakes. You or I might get tired of telling someone their guess is too high or too low, but the computer never does. It will also never forget the number or tell the player their guess is too low or too high when it's actually not.

Let's tap into the power of loops with a `while` loop. A `while` loop repeats a set of statements as long as some *condition* is true. A condition is just something we can test. For example, in this program, we want to know whether the user correctly guessed the secret number. If they didn't guess correctly, we want to keep giving them a chance to guess again until they get it right.

To write a `while` loop, we need to know what condition we want to test for before repeating the loop each time. In the guessing game, we want the

user to guess again as long as their guess isn't equal to the secret number theNumber. When the user's guess is equal to the secret number, the user wins and the game is over, so the loop should stop.

To create a while loop, we need to insert a while statement before the last three lines of code and then wrap the three lines for guessing inside a new pair of braces, as follows:

```
        int guess = 0;
        while (guess != theNumber) {
        System.out.println("Guess a number between 1 and 100:");
        guess = scan.nextInt();
        System.out.println("You entered " + guess + ".");
        }
    }
}
```

We use the keyword while to let Java know we're building a while loop, and then we put the appropriate condition inside parentheses. The part inside the parentheses, guess != theNumber, means that while the value stored in guess is not equal to (!=) the value stored in theNumber, the loop should repeat whatever statement or set of statements immediately follow this line of code. The != operator is a *comparison operator*—in this case, it compares guess and theNumber and evaluates whether they're different, or *not equal*. You'll learn about other comparison operators in the next section, but this is the one we need for the guessing while loop.

We need to tell Java what statements to repeat in the while loop, so I've added an opening brace, {, after the while statement. In the same way that braces group all the statements together in the main() method, these braces group statements together inside the while loop.

There are three statements that we want to include inside the loop. First we need the println() statement that prompts the user to guess a number. Then we need the statement that scans the keyboard and records the guess with the nextInt() method. Finally, we need the println() statement that tells the user what they entered. To turn this set of statements into a block of code that will be run repeatedly in the while statement, we write the while statement and condition first, then an opening brace, then all three statements, and finally, a closing brace. Don't forget the closing brace! Your program won't run if it's missing.

One good programming practice that will help you keep your code organized and readable is using tab spacing correctly. Highlight the three statements inside the braces for the while statement and then press the TAB key to indent them.

The result should look like the following code:

```
        int guess = 0;
        while (guess != theNumber) {
            System.out.println("Guess a number between 1 and 100:");
            guess = scan.nextInt();
            System.out.println("You entered " + guess + ".");
        }
```

```
    }
}
```

Correct indentation will help you remember to match up your opening and closing braces, and it will help you quickly see which statements are inside a loop or other block of code, as well as which statements are outside the loop. Indentation doesn't affect how your program runs, but if done well, it makes your program much easier to read and maintain.

Save your program now and run it to check that it works. The game is almost playable now, but we still need to tell the program to check if the user's guess is too high, too low, or just right. Time for (drum roll, please . . .) `if` statements!

if Statements: Testing for the Right Conditions

Now that the user is able to guess until they are correct, we need to check the guess to let them know whether they were too high or too low. The statement that allows us to do this is the `if` statement.

An `if` statement will select whether to run a block of statements once or not at all based on a condition, or a *conditional expression*.

We used a conditional expression before in the guessing loop: (`guess != theNumber`). To check whether a guess is too high or too low, we just need a few more comparison operators: less than (`<`), greater than (`>`), and equal to (`==`).

First, instead of just telling the user what their guess was, let's write some code to check whether their guess was too low. Replace the last line of the `while` statement with the following two-line `if` statement:

```
        while (guess != theNumber) {
            System.out.println("Guess a number between 1 and 100:");
            guess = scan.nextInt();
            if (guess < theNumber)
                System.out.println(guess + " is too low. Try again.");
        }
```

The `if` statement begins with the keyword `if`, followed by a conditional expression in parentheses. In this case, the condition is `guess < theNumber`, which means the value of the user's guess is less than the value of the random secret number. Notice there's no semicolon after the parentheses, because the `println()` statement that follows is actually part of the `if` statement. The whole statement tells the program that if the condition is true, it should print the user's guess and let them know they guessed too low. We use the concatenation operator (`+`) between the user's guess and the string of text telling them the guess was too low. Note the space after the first double quote and before `is`. This separates the user's guess from the word `is`.

If you run the program now and enter a low guess, like `1`, the `if` statement should tell the program to say your guess is too low. That's a good start, but what if we guess a number that's too high instead? In that case, we need an `else` statement.

The `else` statement gives the program a way to choose an alternative path, or set of steps, if the condition in the `if` statement is not true. We can test for guesses that are too high or too low with an `if-else` statement pair. Let's add an `else` statement right after the `if` statement:

```
❶ if (guess < theNumber)
      System.out.println(guess + " is too low. Try again.");
❷ else if (guess > theNumber)
      System.out.println(guess + " is too high. Try again.");
```

Notice that the code at ❷ looks similar to the code at ❶. Often when we're using `if-else` statements, we need to check for multiple conditions in a row, instead of just one. Here, we need to check for a guess that's too low, too high, or just right. In cases like this, we can chain `if-else` conditions together by placing the next `if` statement inside the `else` portion of the previous `if-else` statement. At ❷ we've begun the next `if` statement immediately after the `else` from the previous condition. If the guess is higher than the number, the program tells the user their guess is too high. Now that the program can tell the user if their guess is too high or too low, we just need to tell them if they guessed correctly and won!

If neither of the previous conditions is true—the user's guess is not too high and not too low—then they must have guessed the number. So we add one final `else` statement:

```
❶ if (guess < theNumber)
      System.out.println(guess + " is too low. Try again.");
❷ else if (guess > theNumber)
      System.out.println(guess + " is too high. Try again.");
❸ else
      System.out.println(guess + " is correct. You win!");
```

Notice that we don't need a conditional expression for this final `else` statement ❸. A correct guess is the only remaining option if the number is neither too high nor too low. In the case of a winning guess, we provide the statement to let the user know they've won. The full program up to this point is shown in Listing 2-4. Save your *HiLo.java* file and run the program to check that it works. It should prompt you to enter guesses until you guess the correct number.

```
import java.util.Scanner;

public class HiLo {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        // Create a random number for the user to guess
        int theNumber = (int)(Math.random() * 100 + 1);
```

```
        // System.out.println( theNumber );
        int guess = 0;
        while (guess != theNumber) {
            System.out.println("Guess a number between 1 and 100:");
            guess = scan.nextInt();
            if (guess < theNumber)
                System.out.println(guess + " is too low. Try again.");
            else if (guess > theNumber)
                System.out.println(guess + " is too high. Try again.");
            else
                System.out.println(guess + " is correct. You win!");
        }   // End of while loop for guessing
    }
}
```

Listing 2-4: The Hi-Lo guessing game is complete for a single full round of play.

The full program is now a completely playable guessing game! After the user wins, the program tells them that they guessed correctly and won, and then it ends, as shown in Figure 2-6.

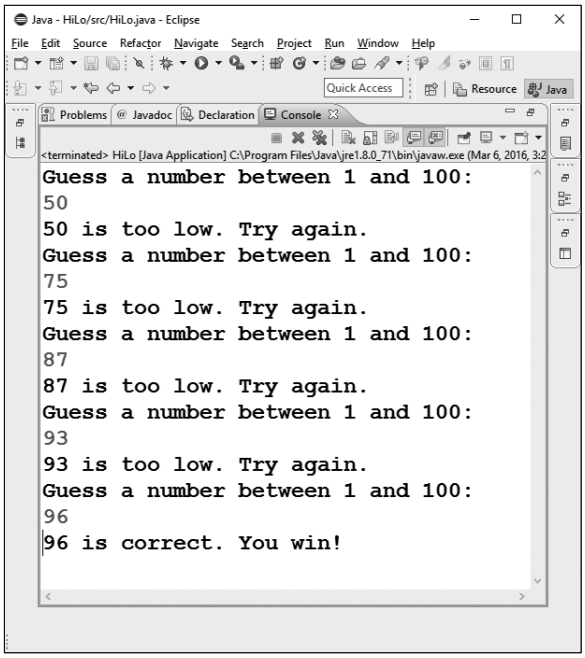


Figure 2-6: One full play-through of the Hi-Lo guessing game—the program ends when the user guesses the secret number.

Give yourself a hand! You've built a program in Java from scratch, and if this is your first program ever in Java, you deserve some kudos. Enjoy the game for a few rounds and see if you can guess the number in fewer tries each time. Test your program to make sure it works the way you want, and we'll add some improvements in the next section.

Adding a Play Again Loop

Right now, the only way to play the guessing game again is to rerun the program in Eclipse. Fortunately, we already know there's a way to make our program do something over and over again—we need another loop!

The guessing game program ends when the user guesses the right number because there's nothing after the `while` loop. The `while` loop ends when the condition (`guess != theNumber`) is no longer true. A user might want to play over and over once they get the hang of the game. For this play again loop, we'll learn a new keyword and a new kind of loop: the `do-while` loop.

Like the `while` loop, a `do-while` loop repeats a block of statements as long as a condition is true. Unlike the `while` loop, however, the block of code inside a `do-while` loop is guaranteed to run at least once. There are times when the condition at the top of a `while` loop may be false before the loop even starts, so the entire loop and all the lines of code inside it are ignored. Think of the condition of a `while` loop as being like a thermostat on a heater. If the temperature of the room is already warm enough and the condition for the heater to turn on isn't met, the heater may not turn on at all.

For our guessing game, or almost any game program in general, we choose a `do-while` loop (we sometimes call this the *game loop*), because the user probably wants to play the game at least once. We also usually want to ask the user if they would like to play again, and the user typically responds yes or no (or `y` or `n` in a text-based game like this one). The game will continue to play through the game loop as long as the user responds with a yes.

To check the user's response, we'll need a new type of variable: a `String`. Strings are objects that hold text within double quotation marks, like `"y"`, or `"yes"`, or `"My name is Bryson! I hope you like my game!"`. Earlier we used an integer variable, or `int` type, to hold the numbers the user was guessing, but now we need to hold text, so we'll use a `String` instead. We can add a `String` variable to the top of the program, right after the `Scanner` setup:

```
Scanner scan = new Scanner(System.in);
String playAgain = "";
```

Notice the `String` type begins with an uppercase `S`. This is because the `String` type is actually a class, complete with several useful functions for working with strings of text. I've named the variable `playAgain`, using camel case with a capital `A` to start the second word. Remember, no spaces are allowed in variable names. And, just like how we gave an initial value of `0` to the `guess` variable with `int guess = 0`, here we've given an initial value to the `playAgain` variable with `playAgain = ""`. The two double quotes, with no space between, indicate an empty string, or a `String` variable with no text in it. We'll assign a different text value to the variable later, when the user enters `y` or `n`.

Just as we did with the `while` loop, we'll need to figure out which statements should be repeated in the `do-while` loop. The `do-while` loop will be our main loop, so almost all of the statements in the program will go inside it. In fact, all the remaining statements after the `Scanner` and `String playAgain`

statements will be contained in the `do-while` loop. Those steps describe one full round of play, so for each round, the game repeats all of those steps again, from choosing a new random number to declaring a winning guess and asking the user to play again.

So, we can add the `do` keyword and an opening brace immediately after these two lines and before the code that creates the secret number:

```
        Scanner scan = new Scanner(System.in);
        String playAgain = "";
        do {
            // Create a random number for the user to guess
            int theNumber = (int)(Math.random() * 100 + 1);
            // System.out.println( theNumber );
            int guess = 0;
            while (guess != theNumber) {
```

Then, after the closing brace for the `while` loop for guessing and the brace following our last `else` statement, we'll ask the user if they would like to play again and get their response from the keyboard.

Then we need to close the `do-while` loop with a `while` condition to check whether the user replied with a yes:

```
        }   // End of while loop for guessing
         ❶ System.out.println("Would you like to play again (y/n)?");
         ❷ playAgain = scan.next();
     ❸ } while (playAgain.equalsIgnoreCase("y"));
  ❹ }
❺ }
```

The prompt asks the user `"Would you like to play again (y/n)?"` ❶, to which they can reply with a single letter, `y` for yes or `n` for no. At ❷, the `scan.next()` function scans the keyboard for input, but instead of looking for the next integer as `nextInt()` does, it looks for the next character or group of characters that the user types on the keyboard. Whatever the user types will get stored in the variable `playAgain`.

The line at ❸ closes the block of code that repeats the game with a brace, and it contains the `while` condition that determines whether the code will run again. Within the `while` condition, you can see an example of the `equals()` method of a `String` object. The `equals()` method tells you whether a string variable is exactly the same as another string of characters, and the `equalsIgnoreCase()` method tells you whether the strings are equal even if their capitalization is different. In our game, if the user wants to play again, they are asked to type `y`. However, if we just test for a lowercase `y`, we might miss an uppercase `Y` response. In this case, we want to be flexible by checking for the letter `y`, whether it is uppercase or lowercase, so we use the string method `equalsIgnoreCase()`.

The final `while` statement tells Java to continue to do the game loop while the string variable `playAgain` is either an uppercase or lowercase `y`. The final two closing braces at ❹ and ❺ are the ones that were already in the program. The one at ❹ closes the `main()` method, and the one at ❺ closes the entire `HiLo` class. I've included them just to show where lines ❶ through ❸ should be inserted.

The complete game to this point is shown in Listing 2-5.

```
import java.util.Scanner;
public class HiLo {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        String playAgain = "";
        do {
            // Create a random number for the user to guess
            int theNumber = (int)(Math.random() * 100 + 1);
            // System.out.println( theNumber );
            int guess = 0;
            while (guess != theNumber) {
                System.out.println("Guess a number between 1 and 100:");
                guess = scan.nextInt();
                if (guess < theNumber)
                    System.out.println(guess + " is too low. Try again.");
                else if (guess > theNumber)
                    System.out.println(guess + " is too high. Try again.");
                else
                    System.out.println(guess + " is correct. You win!");
            }   // End of while loop for guessing
            System.out.println("Would you like to play again (y/n)?");
            playAgain = scan.next();
        } while (playAgain.equalsIgnoreCase("y"));
    }
}
```

Listing 2-5: The Hi-Lo guessing game is ready to play over and over again.

Review your code to make sure you've added everything in the correct place, check your braces and semicolons, and take a moment to save your file. We'll test the game in the next section.

NOTE *Your indentation, which is the tab spacing at the beginning of each line, may not look exactly like the last code snippet because we've added braces in a couple of new places. Fortunately, adding new features, including loops and other blocks of code, is so common in Java that Eclipse has a menu option to clean up indentation automatically. First, select (highlight) all the text in your* HiLo.java *file on the screen. Then, go to* ***Source ▸ Correct Indentation****. Eclipse will correctly indent each line of code to show which statements are meant to be grouped together. As I mentioned before, the indentation doesn't matter to the computer (the program will run just fine even with* no *tabs or extra spaces), but good indentation and spacing help make the program easier to read.*

Testing the Game

Now that the play again loop is in place, the game should run perfectly. First, save your *HiLo.java* file and choose **Run ▸ Run** to test the program. After you guess the first random number correctly, the program should ask you if you'd like to play again. As long as you respond `y` (or `Y`) and press ENTER, the program should keep giving you new random numbers to guess. In the screenshot in Figure 2-7, notice that the game starts over when I respond `y` to the prompt to play again.

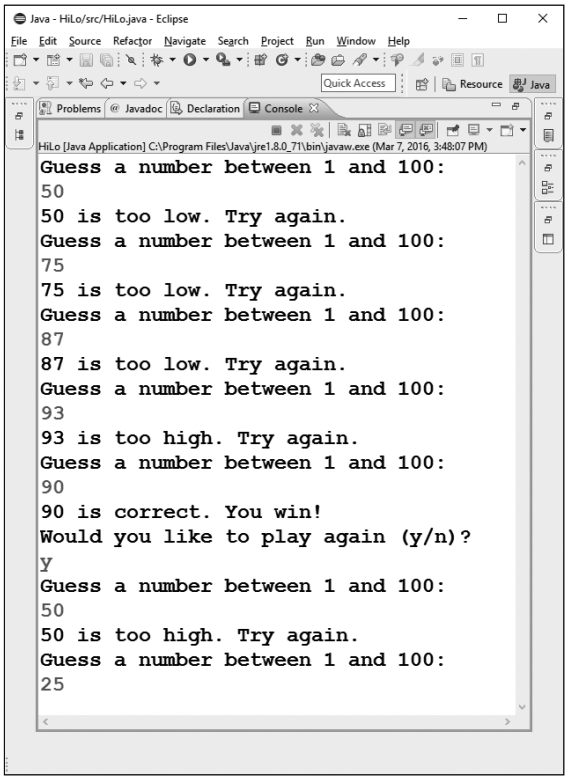


Figure 2-7: The guessing game is fully playable for multiple rounds as long as the user answers `y` or `Y`.

When the user finishes playing and responds to the play again question with `n`, or anything other than `y` or `Y`, the game will end. However, we might want to thank them for playing after they've finished the game. Add the following line after the final `while` statement, before the final two closing braces:

```
        } while (playAgain.equalsIgnoreCase("y"));
        System.out.println("Thank you for playing! Goodbye.");
    }
}
```

Finally, the last line we'll add to the guessing game app is to address a warning you may have noticed in Eclipse. This warning appears as a faint yellow line under the declaration of the `scan` object, as well as a yellow triangle with an exclamation point to the left of that line. Eclipse is bringing to our attention that we've opened a resource that we haven't closed. In programming, this can create what's known as a *resource leak*. This doesn't usually matter if we just open one `Scanner` object for keyboard input, but if we leave multiple `Scanner` objects open without closing them, the program could fill up memory, slowing or even crashing the user's system. We use the `close()` method of the `Scanner` class to tell our program to close the connection to the keyboard.

Add the following line after the `println()` statement thanking the user for playing, before the final two closing braces:

```
        System.out.println("Thank you for playing! Goodbye.");
        scan.close();
    }
}
```

You'll notice that the yellow warning disappears from the Eclipse editor window when we add this line. Eclipse helps with common programming errors like misspellings or missing punctuation, and it even warns us about problems that *could* occur like resource leaks and unused variables. As you build bigger, more complex applications in Java, these features of the IDE will become even more valuable. You can find more information on using Eclipse to debug your programs in the appendix.

The finished program, shown in Listing 2-6, is a fully playable guessing game, complete with the option to play again and guess a new random number every game.

```
import java.util.Scanner;

public class HiLo {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        String playAgain = "";
        do {
            // Create a random number for the user to guess
            int theNumber = (int)(Math.random() * 100 + 1);
            // System.out.println( theNumber );
            int guess = 0;
            while (guess != theNumber) {
                System.out.println("Guess a number between 1 and 100:");
                guess = scan.nextInt();
                if (guess < theNumber)
                    System.out.println(guess + " is too low. Try again.");
                else if (guess > theNumber)
                    System.out.println(guess + " is too high. Try again.");
                else
                    System.out.println(guess + " is correct. You win!");
            }   // End of while loop for guessing
```

```
            System.out.println("Would you like to play again (y/n)?");
            playAgain = scan.next();
        } while (playAgain.equalsIgnoreCase("y"));
        System.out.println("Thank you for playing! Goodbye.");
        scan.close();
    }
}
```

Listing 2-6: The finished text-based, command line guessing game

There are a few things worth noting about the finished Hi-Lo guessing game program. First, despite all the work in writing it, the code is relatively short—fewer than 30 lines long. Nevertheless, you could play it forever if you wanted to. Second, this program not only demonstrates conditions and looping, it also makes use of a loop inside another loop. This is called a *nested loop*, because the guessing loop is contained in, or nested inside, the play again loop. The indentation helps us see where the `do-while` loop begins and ends, and we can see the smaller `while` loop and its `if` statements tabbed over and nested inside the bigger `do-while` loop. Finally, we end the program neatly—both for the user, by thanking them for playing, and for the computer, by closing the scanner resource.

What You Learned

While building a simple, fun, playable game, we've picked up several valuable programming concepts along the way. That's the way I first learned how to code as a kid—I would find a fun game or graphical app, program it, then change it, take it apart, and try new things. Play and exploration are an important part of learning anything new, and I hope you'll take a little time to try new things with each program. The programming challenges at the end of each chapter will also give you an opportunity to try a few new things.

In building this guessing game, we've developed a wide range of skills in Java:

- Creating a new class, `HiLo`
- Importing an existing Java package, `java.util.Scanner`
- Using a `Scanner` object to accept keyboard input
- Declaring and initializing integer and string variables
- Generating a random number with `Math.random()` and casting it to an integer
- Using `while` and `do-while` loops to repeat a set of steps while a condition is true
- Printing text strings and variable values to the command line console
- Scanning integers and strings from the keyboard and storing them in variables
- Testing various conditional expressions in `if` and `if-else` statements

- Using `String` methods to compare string values with `equalsIgnoreCase()`
- Closing input resources like `Scanner` objects with the `close()` method
- Running a command line program from inside Eclipse

In addition to practical skills, you've also developed a working knowledge of several important programming concepts in Java:

Variables `theNumber` is an integer variable, or `int`, and so is `guess`. `playAgain` is a string variable, or `String`. We change the values of these variables as we play the game by entering new number guesses or answering `y` or `n`.

Methods Methods are what we call functions inside a class in Java. `Math.random()` is a method for generating random numbers between 0.0 and 1.0. The `scan.nextInt()` method accepts numeric input from the user. `System.out.println()` is a function for displaying text to the console or terminal window.

Conditionals The `if-else` statements allow us to test whether a condition, like `guess < theNumber`, is true and run a different block of code depending on the outcome of that test. We also use conditional expressions to determine whether to perform a loop again, as in the statement `while (guess != theNumber)`. This statement will loop as long as `guess` is not equal to `theNumber`. Remember the test for "is equal to" is the double equal sign: `==`.

Loops A `while` loop lets us repeat a block of code as long as a condition is true. We used a `while` loop in the guessing game to keep asking the user for another guess until they got the right number. A `do-while` loop always runs at least once, and we used one to ask the user if they wanted to play again.

Classes The whole HiLo app is a Java class, `public class HiLo`. A class is a template. Now that we've built a class template for the HiLo guessing game, we can reuse it to play the guessing game across many different computers. We also imported the `Scanner` and `Math` classes in this app to accept user input and generate random numbers. We'll write our own classes to do something new, and we'll take advantage of the classes already included in Java to do everyday tasks like input, math, and more.

Programming Challenges

Try these programming challenges to review and practice what you've learned and to expand your programming skills. If you get stuck, you can visit the book's website at *https://www.nostarch.com/learnjava/* to download sample solutions, or you can watch this lesson in the video course online at *http://www.udemy.com/java-the-easy-way/* for step-by-step solutions. Chapter 2 is free to preview, and you can use the coupon code BOOKHALFOFF to save 50 percent when you buy the full course.

#1: Expanding Your Range

For this first programming challenge, change the guessing game to use a bigger range of numbers. Instead of 1 to 100, try having the user guess between –100 and 100!

HINT *Multiply* `Math.random()` *by 200 and subtract 100 from the result.*

Remember to change *both* the programming statement that generates the random number *and* the prompt that tells the user the range they should guess between.

If you want an easier game, you can change the range from 1 to 10 and wow your friends when you can guess the secret number in just four tries. Try other ranges, like 1 to 1,000, or even 1 to 1,000,000, or use negative ranges if you want! (Remember that you can't use commas when writing the number in Java.) You'll not only get better at programming but also improve your math skills. Change the program however you'd like and have fun with it!

STRATEGIZE YOUR GUESSES

You may discover that the more you play the guessing game, the faster you'll be able to guess the secret number. You might even stumble onto the fact that you can guess the number fastest by guessing in the middle of a range with each new guess. This technique is called a *binary search*. Guessing a number in the middle of the possible range cuts the number of possibilities in half each time.

Here's how it works. For a number from 1 to 100, guess 50. If that's too low, you know the secret number must be between 51 and 100, so guess 75, right in the middle of this range. If that's too low, try a number halfway between 76 and 100, which would be 87. One reason the binary search is so valuable is that we can reduce the number of guesses to just seven tries (or fewer) to find a secret number between 1 and 100, every time. Try it out!

When you get the hang of guessing a number between 1 and 100 in seven tries or less, try guessing a number from 1 to 1,000 in just 10 tries. If you're really brave (and have a pencil nearby), try guessing a number from 1 to 1,000,000. Believe it or not, it should take you just 20 guesses.

#2: Counting Tries

We've already built a pretty cool guessing game app, but let's try adding one more feature to the game. Your challenge is to count and report how many tries it takes the user to guess the secret number. It could look something like the following:

```
62 is correct! You win!
It only took you 7 tries! Good work!
```

To accomplish this task, you'll need to create a new variable (you might add a line like `int numberOfTries = 0;`), and you'll have to add to the number of tries every time the guessing loop executes. You can do this by increasing the variable `numberOfTries` by one for each new loop using `numberOfTries = numberOfTries + 1`. Be sure to include text to let the user know the number of tries.

It may take a few tries to get all the code working in the right order at the right time, but it's worth the effort and will help you practice your new skills. In Chapter 3, we'll build this feature into a different version of the guessing game. In the meantime, I hope you'll come up with even more ideas for improving and changing the game. Playing with your programs, taking them apart, and rebuilding them can be the best way to learn.

#3: Playing MadLibs

For your final challenge in this chapter, let's write a completely new program. We've learned how to ask a user for input and store it in a variable. We've also learned how to print out both text and variable values to the screen. With those skills, you can build even more interesting and fun programs.

Have you ever played MadLibs? Let's try to use our new skills to build a program in that same style. MadLibs asks a player for various words or parts of speech, such as a color, a past-tense verb, or an adjective, and then inserts the words the player chose into a template, usually resulting in a funny story. For example, if a player gave a color of "pink," a past-tense verb of "burped," and an adjective of "silly" and then inserted them into the template "The ____ dragon ____ at the ____ knight," they would get the result "The *pink* dragon *burped* at the *silly* knight."

Now, the challenge is to write a new program, *MadLibs.java*, with a class called `MadLibs` and a `main()` method that prompts the user for several words. Those words should each be stored in a different `String` variable, like `color`, `pastTenseVerb`, `adjective`, and `noun`, which you initialize as empty strings. Then, after the user has entered their last word, the program should print a completed sentence or story by replacing the empty strings with the words the user provided, like this:

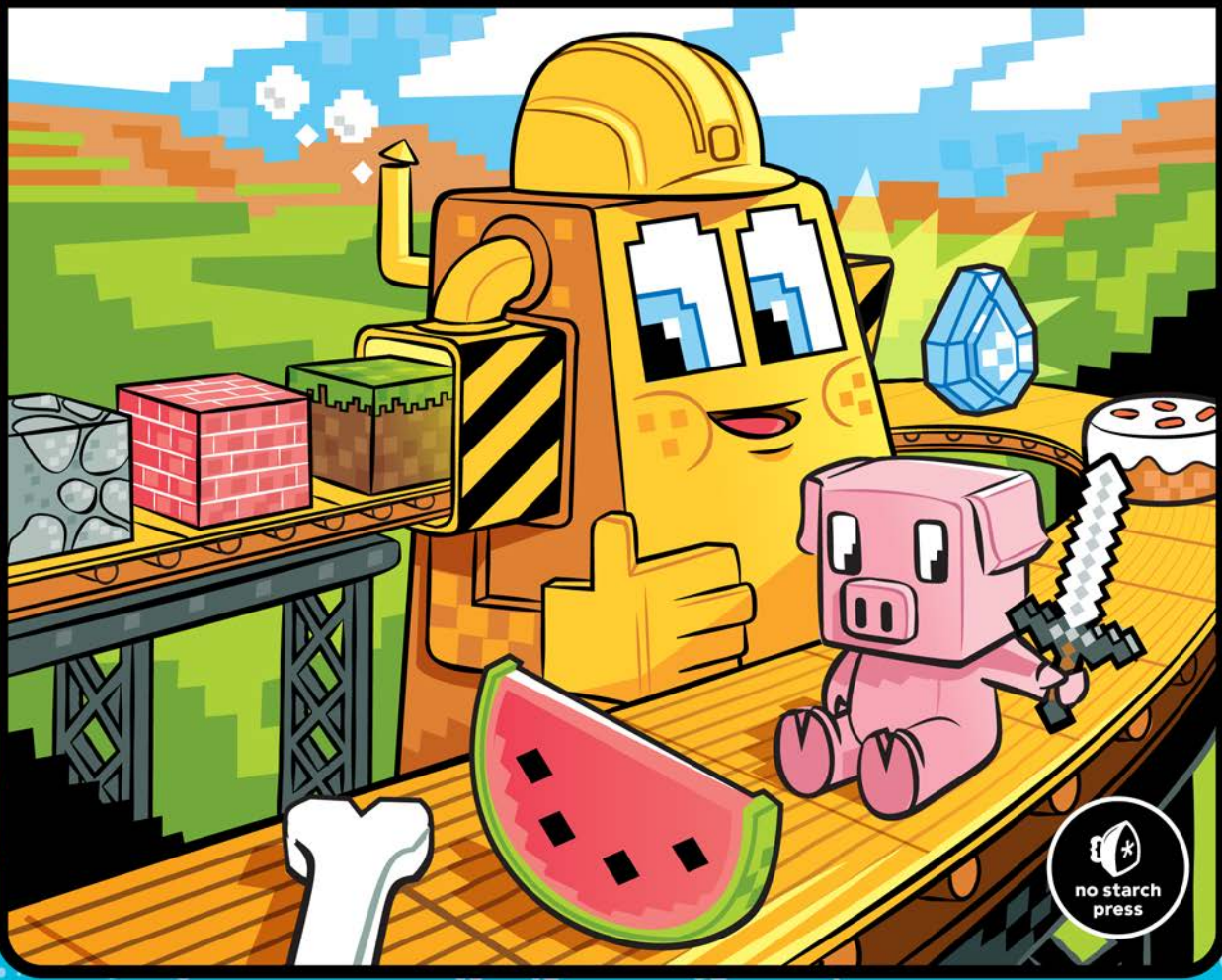
```
System.out.print("The " + color + " dragon " + pastTenseVerb + " at the " + adjective);
System.out.println(" knight, who rode in on a sturdy, giant " + noun + ".");
```

Note that the first statement is a `print()` statement instead of a `println()`. The `print()` statement continues printing at the end of the same line, allowing us to build a longer paragraph or story. The `println()` statement, however, always skips a line after printing, like when you press ENTER at the end of the line. You can write a longer MadLibs story by using different variable names like `noun1`, `noun2`, and `noun3`. Give it a try, and get ready to laugh at the funny stories you create! Try to personalize each program you create by adding new features and making it your own.

AUTOMATE THE MINECRAFT STUFF

MINE, FARM, AND BUILD WITH CODE

AL SWEIGART



no starch press

9

BUILDING A COBBLESTONE GENERATOR

As you mine, the most common blocks you'll find are stone, which become cobblestone when mined. But you can turn cobblestone back into stone by smelting it in a furnace. Then you can craft this stone into stone bricks for your buildings' construction materials.

Whew! That's a lot of work in dangerous dark mines just to get stone bricks. In this chapter, you'll learn how to create a cobblestone generator that will give you infinite cobblestone to work with; then you'll create a turtle program to automatically mine and smelt that cobblestone into stone. Safe inside your base, you'll have a production line for an endless amount of stone bricks to build with.

BLUEPRINTS FOR THE COBBLESTONE GENERATOR

Although you can obtain cobblestone by mining it in Minecraft, you can also create it by mixing a stream of water with a stream of lava, which forms a block of cobblestone. You can use this knowledge to build a cobblestone generator that performs the same process to create an unlimited number of cobblestone blocks. Turtles can mine this cobblestone forever because their tools don't wear out.

To create a cobblestone generator, follow the blueprint in Figure 9-1. A generator has three layers of blocks. You'll need one type of block to act as an enclosure to hold the water and lava streams. I used glass, but you can use any nonflammable blocks. You'll need to use three iron to craft a bucket, and then fill the bucket from a lava pool. Although you can find lava pools on the surface, they're more commonly found deep underground near bedrock. You can draw water from the rivers, ponds, and oceans on the surface.

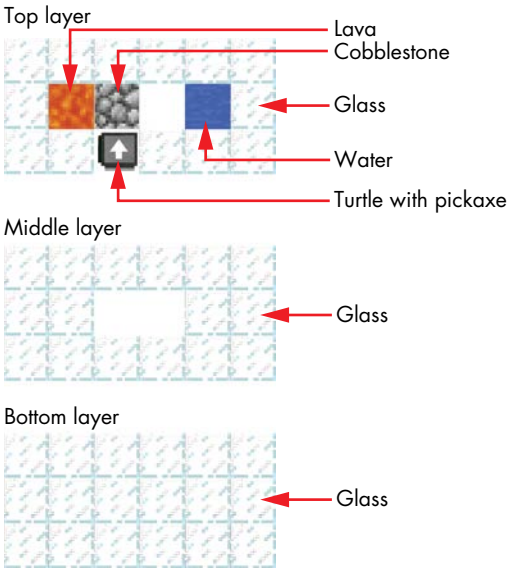


Figure 9-1: A bird's-eye view of the blueprints for a cobblestone generator

When placing the lava and water, *be sure to place the lava first*. Otherwise, the water stream will mix directly with the lava block (instead of its stream), turning it into obsidian. You don't need to place the cobblestone block in the blueprint: it will form automatically.

Figure 9-2 shows the completed cobblestone generator.

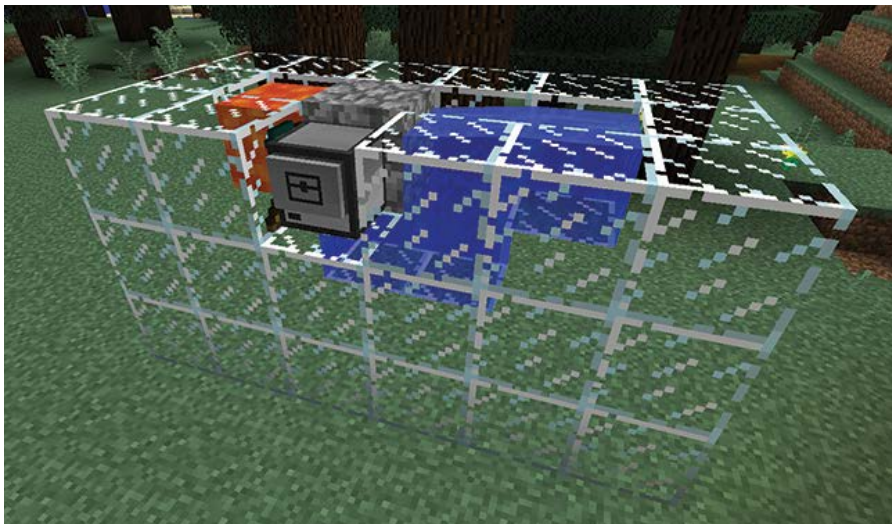


Figure 9-2: The completed cobblestone generator with a turtle in the empty slot on the top layer

Whenever the cobblestone block in the generator is mined, it opens space for the lava to flow into to create a new cobblestone block. You could mine the cobblestone an infinite number of times, but you would wear out a lot of pickaxes. By placing a turtle into the open spot on the top layer facing the cobblestone block, you can have the turtle mine the cobblestone blocks forever. Because the turtle doesn't even need to move, it doesn't use any fuel either!

SETTING UP FURNACES FOR SMELTING THE COBBLESTONE

Even though we now have an infinite supply of cobblestone, we still need to smelt the cobblestone into stone to use it. To do this, we'll create the `cobminer` program, which will make a turtle mine the cobblestone from the generator and then deposit the mined cobblestone into furnaces to smelt into stone.

Before you create the `cobminer` program, you'll need to do some setup. First, you need to extend the cobblestone generator with five furnaces by adding them to the middle layer behind the turtle, as shown in Figure 9-3.

The turtle running the new cobblestone miner program will mine until it has a full stack of 64 cobblestones. Then it will move backward over the furnaces, dropping the cobblestone into them. The furnaces will smelt the cobblestone into stone. If all the furnaces are full, the turtle will wait five minutes before trying to drop cobblestone into them again. This entire process will repeat forever.

Figure 9-3: Five furnaces added to the middle layer of the cobblestone generator (left) and the furnaces in the game (right)

In Chapter 10, we'll create a `brickcrafter` program to run a second turtle. The `brickcrafter` program will pick up the smelted stone from the furnaces and use it to craft stone bricks. The turtle will store these stone brick blocks in a nearby chest for the player.

WRITING THE COBMINER PROGRAM

To write the `cobminer` program, run **`edit cobminer`** from the command shell and enter the following code:

cobminer

```
 1. --[[ Stone Brick Factory program by Al Sweigart
 2. Mines cobblestone from a generator, turtle 1 of 2 ]]
 3.
 4. os.loadAPI('hare')  -- load the hare library
 5. local numToDrop
 6. local NUM_FURNACES = 5
 7.
 8. print('Starting mining program...')
 9. while true do
10.   -- mine cobblestone
11.   if turtle.detect() then
12.     print('Cobblestone detected. Mining...')
13.     turtle.dig()  -- mine cobblestone
14.   else
15.     print('No cobblestone. Sleeping...')
16.     os.sleep(0.5)  -- half second pause
17.   end
18.
19.   -- check for a full stack of cobble
20.   hare.selectItem('minecraft:cobblestone')
21.   if turtle.getItemCount() == 64 then
22.     -- check turtle's fuel
23.     if turtle.getFuelLevel() < (2 * NUM_FURNACES) then
24.       error('Turtle needs more fuel!')
25.     end
```

```
26.
27.     -- put cobble in furnaces
28.     print('Dropping off cobblestone...')
29.     for furnacesToFill = NUM_FURNACES, 1, -1 do
30.       turtle.back()  -- move over furnace
31.       numToDrop = math.floor(turtle.getItemCount() / furnacesToFill)
32.       turtle.dropDown(numToDrop)  -- put cobblestone in furnace
33.     end
34.
35.     -- move back to cobblestone generator
36.     for moves = 1, NUM_FURNACES do
37.       turtle.forward()
38.     end
39.
40.     if turtle.getItemCount() > 0 then
41.       print('All furnaces full. Sleeping...')
42.       os.sleep(300)  -- wait for 5 minutes
43.     end
44.   end
45. end
```

After you've entered these instructions, press CTRL, make sure **[Save]** is selected, and press ENTER. Then quit the editor by pressing CTRL, selecting **[Exit]**, and pressing ENTER. You can also download this program by running **`pastebin get YhvSiw7e cobminer`**. In addition, you'll need the `hare` module, which you can download by running **`pastebin get wwzvaKuW hare`**.

RUNNING THE COBMINER PROGRAM

After you've set up the cobblestone generator with five furnaces, position the turtle facing the cobblestone block and run **`cobminer`**. The turtle will begin to mine the cobblestone until it has 64 blocks; then it will drop them into the furnaces. Until you write the `brickcrafter` program in Chapter 10, you'll have to manually load fuel into the furnaces and remove the smelted stone blocks from them. To create fuel for the furnaces, smelt the wood blocks from the tree farming turtles in separate furnaces to produce charcoal for fueling the furnaces. Let's look at each part of the `cobminer` program.

SETTING UP YOUR PROGRAM AND MAKING A CONSTANT VARIABLE

The first couple of lines of the program contain the usual comment that describes what the program is.

```
1. --[[ Stone Brick Factory program by Al Sweigart
2. Mines cobblestone from a generator, turtle 1 of 2 ]]
3.
4. os.loadAPI('hare')  -- load the hare library
5. local numToDrop
```

Line 4 loads the `hare` module so the program can call `hare.selectItem()`. Line 5 declares a variable called `numToDrop`, which is used later in the program.

Line 6 declares the `NUM_FURNACES` variable, which contains an integer that represents the number of furnaces that are placed behind the turtle.

```
6. local NUM_FURNACES = 5
```

This program has `5` furnaces, but you can add more furnaces if you like. If you add more furnaces to your cobblestone generator, set the value of `NUM_FURNACES` to the new number of furnaces.

The `NUM_FURNACES` variable is uppercase because it's a *constant* variable, which means its value never changes. Uppercase names for constants are just a convention; constants are still regular variables. The capitalized name helps remind you that you shouldn't write code that changes the variable. It might seem odd to have a variable whose value never changes, but using constants will make your code easier to understand and convenient to make any future changes.

For example, you need to indicate the number of furnaces in your code, but if you use `5` instead of `NUM_FURNACES` in your code and later change the number of furnaces, you would have to update your code everywhere `5` is used. When you use a constant like `NUM_FURNACES`, you can update your code by just changing the assignment statement on line 6. Constants make your code clearer and easy to modify.

MINING THE COBBLESTONE FROM THE GENERATOR

Line 9 begins the program's main `while` loop, which contains the code to make the turtle mine cobblestone, move over the furnaces, and drop cobblestone into the furnaces.

```
 8. print('Starting mining program...')
 9. while true do
10.   -- mine cobblestone
11.   if turtle.detect() then
12.     print('Cobblestone detected. Mining...')
13.     turtle.dig()  -- mine cobblestone
```

The first part, mining cobblestone, begins on line 11. The `turtle.detect()` function returns `true` if a cobblestone block is in front of the turtle. In that case, the program displays `Cobblestone detected. Mining...` and the call `turtle.dig()` on line 13 mines the cobblestone.

However, if there is no cobblestone because it was previously mined (and the new cobblestone block hasn't formed yet), `turtle.detect()`returns `false`. When the `if` statement's condition is `false`, the block of code after the `else` statement on line 14 runs.

```
14.   else
15.     print('No cobblestone. Sleeping...')
16.     os.sleep(0.5)  -- half second pause
17.   end
```

This code displays `No cobblestone. Sleeping...` and calls `os.sleep(0.5)` to pause the program for half a second so a new cobblestone block has enough time to form. This new cobblestone block will be mined when the program loops around again. When the turtle is done mining cobblestone, it needs to smelt the cobblestone in the furnaces.

INTERACTING WITH FURNACES

Furnaces have three slots: a *fuel slot* where burnable items like coal power the furnace, an *input slot* for items to be smelted, and an *output slot* where the smelted items remain until the player takes them. The turtle's position next to the furnace determines whether the turtle is putting an item into the furnace as fuel, putting an item in as a block to smelt, or removing the final product. If the turtle is on the side of the furnace, it can drop and take items from the furnace's fuel slot. If a turtle is above a furnace, it can drop and take items from the furnace's input slot to smelt them. If a turtle is below a furnace, it can take smelted items from the furnace's output slot. Figure 9-4 shows these positions.

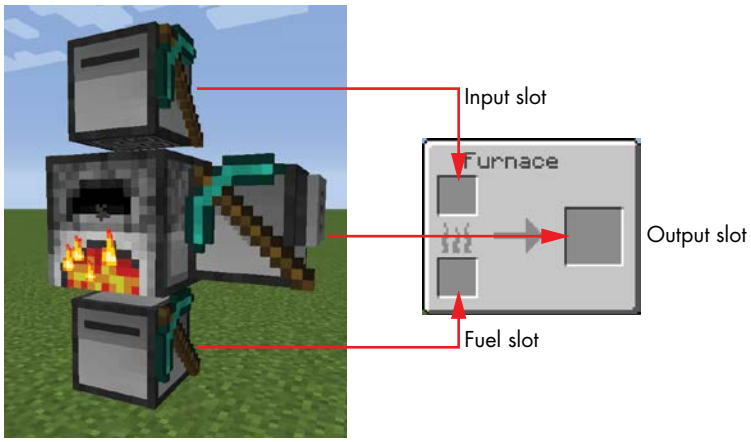


Figure 9-4: The turtle's position indicates which furnace slot it interacts with.

Next, the turtle will drop the cobblestone blocks into the furnaces.

MAKING CODE READABLE WITH CONSTANTS

After checking for a cobblestone block to mine, the program then checks whether the turtle has collected 64 blocks of cobblestone, which is the

maximum an inventory slot can hold. If so, the turtle is ready to drop them into the furnaces behind it after checking that it has enough fuel to travel across the furnaces and back to the cobblestone generator.

```
19.   -- check for a full stack of cobble
20.   hare.selectItem('minecraft:cobblestone')
21.   if turtle.getItemCount() == 64 then
22.     -- check turtle's fuel
23.     if turtle.getFuelLevel() < (2 * NUM_FURNACES) then
24.       error('Turtle needs more fuel!')
25.     end
```

The `hare.selectItem()` function on line 20 finds the inventory slot with cobblestone and selects the slot. Line 21 calls `turtle.getItemCount()` to check the number of cobblestone in this slot. If the total is `64` cobblestone, the program calls `turtle.getFuelLevel()` to check the turtle's fuel level.

Line 23 checks whether the turtle's fuel level is less than `2 * NUM_FURNACES`. The reason is that the turtle needs enough fuel to move over each furnace and then move back across each furnace to return to its starting position, as shown in Figure 9-5.

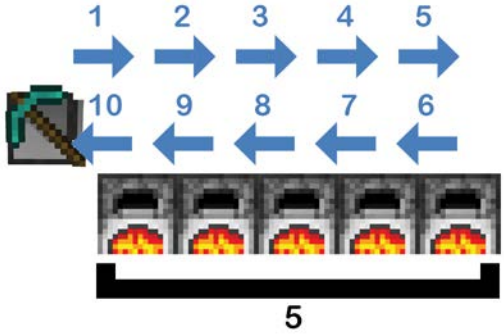


Figure 9-5: The amount of fuel needed to move over the furnaces and back is twice the number of furnaces.

If the turtle doesn't have enough fuel, line 24 calls `error()` and displays `Turtle needs more fuel!`. Then line 25 terminates the program.

DROPPING THE COBBLESTONE INTO THE FURNACES

When the turtle has 64 cobblestone and enough fuel to travel across the furnaces and back, the turtle can move backward and drop off the cobblestone, like in Figure 9-6.

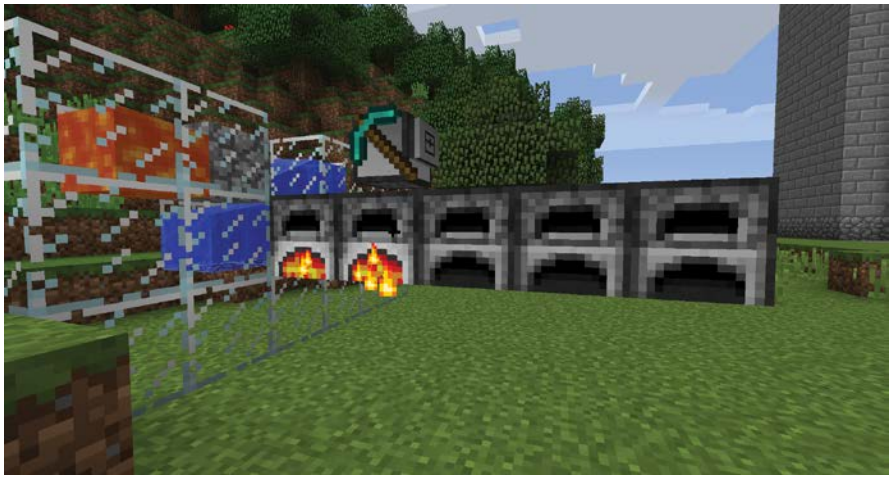


Figure 9-6: The turtle dropping cobblestone blocks into the furnaces.

The `for` loop on line 30 is slightly different from `for` loops we've used before. A `for` loop can count up, as in `for i = 1, 10 do`, but it can also count in different increments when you include a third number, a *step argument*, to the statement. Instead of adding `1` on each iteration, the `for` loop will add the number indicated in the step argument. If you use a negative number, as in `for i = 10, 1, -1 do`, you can make the loop count down.

```
27.     -- put cobble in furnaces
28.     print('Dropping off cobblestone...')
29.     for furnacesToFill = NUM_FURNACES, 1, -1 do
30.       turtle.back()  -- move over furnace
31.       numToDrop = math.floor(turtle.getItemCount() / furnacesToFill)
32.       turtle.dropDown(numToDrop)  -- put cobblestone in furnace
33.     end
```

The `for` loop on line 29 tells the turtle to move backward `NUM_FURNACES` (or `5`) times. However, it begins counting at `5` down to `1` instead of `1` up to `5`, so we can use the `for` loop variable, `furnacesToFill`, on line 31 to calculate how many cobblestone to drop into each furnace. This calculation uses the `math.floor()` function. Let's look at how this function works.

ROUNDING NUMBERS WITH THE MATH.FLOOR() AND MATH.CEIL() FUNCTIONS

The `math.floor()` function rounds down the number it's passed and returns it, whereas the `math.ceil()` function ("ceil" as in "ceiling") rounds up the number it's passed and returns it. Enter the following into the Lua shell to see how these functions work.

```
lua> math.floor(4.2)
4
lua> math.floor(4.9)
4
lua> math.floor(10.5)
10
lua> math.floor(12.0)
12
lua> math.ceil(4.2)
5
lua> math.ceil(4.9)
5
lua> math.ceil(10.5)
11
❶ lua> math.ceil(12.0)
12
```

Passing a value to `math.floor()` results in the number without its decimal point, whereas passing a number to `math.ceil()` rounds up the number to the next number. When you pass `math.ceil()` a number with a decimal value of `0`, it doesn't round up but instead rounds to the closest integer, as you can see at ❶. The rounding by these functions helps us evenly distribute the turtle's cobblestone into the furnaces.

CALCULATING THE COBBLESTONE TO DISTRIBUTE IN EACH FURNACE

In Minecraft, each furnace can hold up to 64 items to smelt in its input slot. It's more efficient to have all five furnaces smelting instead of putting all the cobblestone in the first furnace but leaving the remaining four empty. For efficiency, we want all the furnaces smelting at the same time instead of just one. To calculate how many cobblestone to drop into each furnace, we'll divide the number of cobblestone in the current slot by `NUM_FURNACES`. Because this division operation might not result in a whole number, such as 64 / 5 = 12.8, we'll pass this number to `math.floor()`, which in this case rounds down the number to 12. Then we'll use the solution to drop that number of cobblestone into each furnace so all the furnaces can smelt at the same time.

But this calculation has a couple of problems. For example, if you have 64 cobblestone and 5 furnaces, the turtle will drop 12 cobblestone in each furnace and be left with 4 blocks. Turtles can mine cobblestone quicker than furnaces can smelt them. And each furnace can hold 64 items at most in its input slot. For each furnace that is full and can't accept any more cobblestone, the turtle would be left with the 12 blocks it couldn't drop. In this case, if even one furnace is full, the turtle would be left with 16 blocks of cobblestone! To address this issue, we'll make a different calculation. Let's look at lines 29 to 33 again:

```
29.     for furnacesToFill = NUM_FURNACES, 1, -1 do
30.       turtle.back()  -- move over furnace
```

```
31.       numToDrop = math.floor(turtle.getItemCount() / furnacesToFill)
32.       turtle.dropDown(numToDrop)  -- put cobblestone in furnace
33.     end
```

Using `numToDrop`, line 31 calculates the number of cobblestone to drop in each furnace with `numToDrop = math.floor(turtle.getItemCount() / furnacesToFill)`. Instead of calculating the number of cobblestone to drop once and storing that number in the `numToDrop` variable, the value of `numToDrop` is recalculated on every iteration. Table 9-1 shows how `numToDrop` is calculated on each iteration of the `for` loop when all the furnaces are empty.

Table 9-1: `numToDrop` Values When All Furnaces Are Empty

Iteration	math.floor(turtle.getItemCount() / furnacesToFill)	numToDrop	Number of cobblestone dropped into furnace
First	`math.floor(64 / 5)`	`12`	12
Second	`math.floor(52 / 4)`	`13`	13
Third	`math.floor(39 / 3)`	`13`	13
Fourth	`math.floor(26 / 2)`	`13`	13
Fifth	`math.floor(13 / 1)`	`13`	13
			Total: 64

However, let's pretend the second furnace is full because the player dropped some of their own mined cobblestone into it. Now no cobblestone can be dropped into the second furnace. But because `numToDrop` is recalculated on each iteration of the `for` loop, the code automatically drops more cobblestone into the later furnaces. Table 9-2 shows how `numToDrop` is calculated on each iteration. Notice that on the second iteration, the number of cobblestone dropped in the furnace is 0 because the second furnace is full.

Table 9-2: `numToDrop` Values When the Second Furnace Is Full

Iteration	math.floor(turtle.getItemCount() / furnacesToFill)	numToDrop	Number of cobblestone dropped into furnace
First	`math.floor(64 / 5)`	`12`	12
Second	`math.floor(52 / 4)`	`13`	0
Third	`math.floor(52 / 3)`	`17`	17
Fourth	`math.floor(35 / 2)`	`17`	17
Fifth	`math.floor(18 / 1)`	`18`	18
			Total: 64

Lines 29 to 33 show that a bit of clever code can make the furnaces work at maximum efficiency. When the `for` loop has finished, the turtle will be over the last furnace and needs to move back to the cobblestone block.

MOVING THE COBBLESTONE MINER BACK INTO POSITION

Lines 36 to 38 keep moving the turtle forward until it is in front of the cobblestone block.

```
35.     -- move back to cobblestone generator
36.     for moves = 1, NUM_FURNACES do
37.       turtle.forward()
38.     end
```

At this point, the turtle checks the current slot. Remember, the turtle can mine cobblestone quicker than furnaces can smelt them. It won't be long before all of the furnaces are completely full but the turtle has 64 cobblestone blocks to drop in them. If any cobblestone is still in it, then all the furnaces were full and the turtle was unable to put this cobblestone in them.

```
40.     if turtle.getItemCount() > 0 then
41.       print('All furnaces full. Sleeping...')
42.       os.sleep(300)  -- wait for 5 minutes
43.     end
44.   end
45. end
```

The `turtle.getItemCount()` returns the number of items in the currently selected slot. If this number is greater than `0` (meaning the turtle still has some cobblestone), line 42 pauses the program for 300 seconds, or 5 minutes, to give the furnaces more time to smelt the previous cobblestone.

Line 43 ends the `if` statement block on line 40, line 44 ends the `if` statement block on line 21, and line 45 ends the `while` loop on line 9. Finally, the execution loops back to line 9 so the turtle continues mining cobblestone and filling the furnaces until it runs out of fuel.

As with the tree farming program in the Chapter 8, you can scale your cobblestone production by building multiple cobblestone generators. You can also add more furnaces behind the turtle and change the `NUM_FURNACES` constant. (Five or six furnaces are plenty to smelt cobblestone; otherwise, your turtle won't be able to mine fast enough to keep up with the furnaces!)

SUMMARY

The cobblestone generator can create an infinite amount of cobblestone blocks for the turtle to mine. The `cobminer` program makes the turtle mine these cobblestone blocks and drop them into the furnaces behind the turtle.

In this chapter, you learned how to build a cobblestone generator that mixes lava and water streams to produce an endless amount of cobblestone blocks. You also learned about constants, which are variables that don't change their values and make your code more readable. In addition, you

learned about the step argument in `for` loops, which lets you create `for` loops that count down instead of up. Finally, you learned how the `math.floor()` and `math.ceil()` functions can round a number down and up, respectively.

In Chapter 10, we'll use the `cobminer` program to create a stone brick factory, which is a two-turtle operation. You'll need to program another turtle to take smelted stone out of the furnaces and craft them into stone brick using the `brickcrafter` program, which we'll write.

CRACKING CODES WITH PYTHON

AN INTRODUCTION TO *BUILDING* AND *BREAKING* *CIPHERS*

AL SWEIGART

5

THE CAESAR CIPHER

"BIG BROTHER IS WATCHING YOU."
—George Orwell, Nineteen Eighty-Four

In Chapter 1, we used a cipher wheel and a chart of letters and numbers to implement the Caesar cipher. In this chapter, we'll implement the Caesar cipher in a computer program.

The reverse cipher we made in Chapter 4 always encrypts the same way. But the Caesar cipher uses keys, which encrypt the message differently depending on which key is used. The keys for the Caesar cipher are the integers from 0 to 25. Even if a cryptanalyst knows the Caesar cipher was used, that alone doesn't give them enough information to break the cipher. They must also know the key.

TOPICS COVERED IN THIS CHAPTER

- The `import` statement
- Constants
- `for` loops
- `if`, `else`, and `elif` statements
- The `in` and `not in` operators
- The `find()` string method

Source Code for the Caesar Cipher Program

Enter the following code into the file editor and save it as *caesarCipher.py*. Then download the *pyperclip.py* module from *https://www.nostarch.com/crackingcodes/* and place it in the same directory (that is, the same folder) as the file *caesarCipher.py*. This module will be imported by *caesarCipher.py*; we'll discuss this in more detail in "Importing Modules and Setting Up Variables" on page 56.

When you're finished setting up the files, press F5 to run the program. If you run into any errors or problems with your code, you can compare it to the code in the book using the online diff tool at *https://www.nostarch.com/crackingcodes/*.

caesarCipher.py

```
 1. # Caesar Cipher
 2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
 3.
 4. import pyperclip
 5.
 6. # The string to be encrypted/decrypted:
 7. message = 'This is my secret message.'
 8.
 9. # The encryption/decryption key:
10. key = 13
11.
12. # Whether the program encrypts or decrypts:
13. mode = 'encrypt' # Set to either 'encrypt' or 'decrypt'.
14.
15. # Every possible symbol that can be encrypted:
16. SYMBOLS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz12345
      67890 !?.'
17.
18. # Store the encrypted/decrypted form of the message:
19. translated = ''
20.
```

```
21. for symbol in message:
22.     # Note: Only symbols in the SYMBOLS string can be
          encrypted/decrypted.
23.     if symbol in SYMBOLS:
24.         symbolIndex = SYMBOLS.find(symbol)
25.
26.         # Perform encryption/decryption:
27.         if mode == 'encrypt':
28.             translatedIndex = symbolIndex + key
29.         elif mode == 'decrypt':
30.             translatedIndex = symbolIndex - key
31.
32.         # Handle wraparound, if needed:
33.         if translatedIndex >= len(SYMBOLS):
34.             translatedIndex = translatedIndex - len(SYMBOLS)
35.         elif translatedIndex < 0:
36.             translatedIndex = translatedIndex + len(SYMBOLS)
37.
38.         translated = translated + SYMBOLS[translatedIndex]
39.     else:
40.         # Append the symbol without encrypting/decrypting:
41.         translated = translated + symbol
42.
43. # Output the translated string:
44. print(translated)
45. pyperclip.copy(translated)
```

Sample Run of the Caesar Cipher Program

When you run the *caesarCipher.py* program, the output looks like this:

```
guv6Jv6Jz!J6rp5r7Jzr66ntrM
```

The output is the string `'This is my secret message.'` encrypted with the Caesar cipher using a key of `13`. The Caesar cipher program you just ran automatically copies this encrypted string to the clipboard so you can paste it in an email or text file. As a result, you can easily send the encrypted output from the program to another person.

You might see the following error message when you run the program:

```
Traceback (most recent call last):
  File "C:\caesarCipher.py", line 4, in <module>
    import pyperclip
ImportError: No module named pyperclip
```

If so, you probably haven't downloaded the *pyperclip.py* module into the right folder. If you confirm that *pyperclip.py* is in the folder with *caesarCipher .py* but still can't get the module to work, just comment out the code on lines 4 and 45 (which have the text `pyperclip` in them) from the *caesarCipher.py* program by placing a `#` in front of them. This makes Python ignore the code

that depends on the *pyperclip.py* module and should allow the program to run successfully. Note that if you comment out that code, the encrypted or decrypted text won't be copied to the clipboard at the end of the program. You can also comment out the `pyperclip` code from the programs in future chapters, which will remove the copy-to-clipboard functionality from those programs, too.

To decrypt the message, just paste the output text as the new value stored in the `message` variable on line 7. Then change the assignment statement on line 13 to store the string `'decrypt'` in the variable `mode`:

```
 6. # The string to be encrypted/decrypted:
 7. message = 'guv6Jv6Jz!J6rp5r7Jzr66ntrM'
 8.
 9. # The encryption/decryption key:
10. key = 13
11.
12. # Whether the program encrypts or decrypts:
13. mode = 'decrypt' # Set to either 'encrypt' or 'decrypt'.
```

When you run the program now, the output looks like this:

```
This is my secret message.
```

Importing Modules and Setting Up Variables

Although Python includes many built-in functions, some functions exist in separate programs called modules. *Modules* are Python programs that contain additional functions that your program can use. We import modules with the appropriately named `import` statement, which consists of the `import` keyword followed by the module name.

Line 4 contains an `import` statement:

```
 1. # Caesar Cipher
 2. # https://www.nostarch.com/crackingcodes/ (BSD Licensed)
 3.
 4. import pyperclip
```

In this case, we're importing a module named `pyperclip` so we can call the `pyperclip.copy()` function later in this program. The `pyperclip.copy()` function will automatically copy strings to your computer's clipboard so you can conveniently paste them into other programs.

The next few lines in *caesarCipher.py* set three variables:

```
 6. # The string to be encrypted/decrypted:
 7. message = 'This is my secret message.'
 8.
 9. # The encryption/decryption key:
10. key = 13
```

```
11.
12. # Whether the program encrypts or decrypts:
13. mode = 'encrypt' # Set to either 'encrypt' or 'decrypt'.
```

The `message` variable stores the string to be encrypted or decrypted, and the `key` variable stores the integer of the encryption key. The `mode` variable stores either the string `'encrypt'`, which makes code later in the program encrypt the string in `message`, or `'decrypt'`, which makes the program decrypt rather than encrypt.

Constants and Variables

Constants are variables whose values shouldn't be changed when the program runs. For example, the Caesar cipher program needs a string that contains every possible character that can be encrypted with this Caesar cipher. Because that string shouldn't change, we store it in the constant variable named `SYMBOLS` in line 16:

```
15. # Every possible symbol that can be encrypted:
16. SYMBOLS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz12345
      67890 !?.'
```

Symbol is a common term used in cryptography for a single character that a cipher can encrypt or decrypt. A *symbol set* is every possible symbol a cipher is set up to encrypt or decrypt. Because we'll use the symbol set many times in this program, and because we don't want to type the full string value each time it appears in the program (we might make typos, which would cause errors), we use a constant variable to store the symbol set. We enter the code for the string value once and place it in the `SYMBOLS` constant.

Note that `SYMBOLS` is in all uppercase letters, which is the naming convention for constants. Although we *could* change `SYMBOLS` just like any other variable, the all uppercase name reminds the programmer not to write code that does so.

As with all conventions, we don't *have* to follow this one. But doing so makes it easier for other programmers to understand how these variables are used. (It can even help you when you're looking at your own code later.)

On line 19, the program stores a blank string in a variable named `translated` that will later store the encrypted or decrypted message:

```
18. # Store the encrypted/decrypted form of the message:
19. translated = ''
```

Just as in the reverse cipher in Chapter 5, by the end of the program, the `translated` variable will contain the completely encrypted (or decrypted) message. But for now it starts as a blank string.

The for Loop Statement

At line 21, we use a type of loop called a `for` loop:

```
21. for symbol in message:
```

Recall that a `while` loop will loop as long as a certain condition is `True`. The `for` loop has a slightly different purpose and doesn't have a condition like the `while` loop. Instead, it loops over a string or a group of values. Figure 5-1 shows the six parts of a `for` loop.

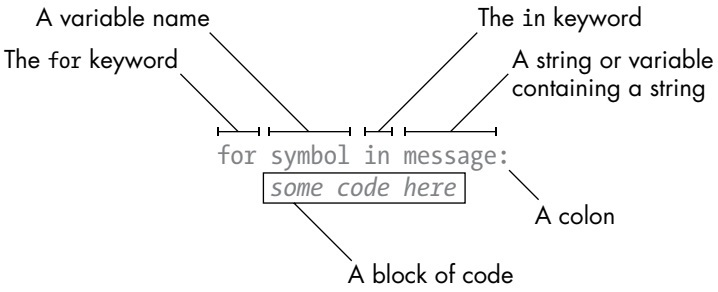


Figure 5-1: The six parts of a for loop statement

Each time the program execution goes through the loop (that is, on each iteration through the loop) the variable in the `for` statement (which in line 21 is `symbol`) takes on the value of the next character in the variable containing a string (which in this case is `message`). The `for` statement is similar to an assignment statement because the variable is created and assigned a value except the `for` statement cycles through different values to assign the variable.

An Example for Loop

For example, type the following into the interactive shell. Note that after you type the first line, the `>>>` prompt will disappear (represented in our code as `...`) because the shell is expecting a block of code after the `for` statement's colon. In the interactive shell, the block will end when you enter a blank line:

```
>>> for letter in 'Howdy':
...     print('The letter is ' + letter)
...
The letter is H
The letter is o
The letter is w
The letter is d
The letter is y
```

This code loops over each character in the string 'Howdy'. When it does, the variable letter takes on the value of each character in 'Howdy' one at a time in order. To see this in action, we've written code in the loop that prints the value of letter for each iteration.

A while Loop Equivalent of a for Loop

The for loop is very similar to the while loop, but when you only need to iterate over characters in a string, using a for loop is more efficient. You could make a while loop act like a for loop by writing a bit more code:

```
❶ >>> i = 0
❷ >>> while i < len('Howdy'):
❸ ...     letter = 'Howdy'[i]
❹ ...     print('The letter is ' + letter)
❺ ...     i = i + 1
...
The letter is H
The letter is o
The letter is w
The letter is d
The letter is y
```

Notice that this while loop works the same as the for loop but is not as short and simple as the for loop. First, we set a new variable i to 0 before the while statement ❶. This statement has a condition that will evaluate to True as long as the variable i is less than the length of the string 'Howdy' ❷. Because i is an integer and only keeps track of the current position in the string, we need to declare a separate letter variable to hold the character in the string at the i position ❸. Then we can print the current value of letter to get the same output as the for loop ❹. When the code is finished executing, we need to increment i by adding 1 to it to move to the next position ❺.

To understand lines 23 and 24 in *caesarCipher.py*, you need to learn about the if, elif, and else statements, the in and not in operators, and the find() string method. We'll look at these in the following sections.

The if Statement

Line 23 in the Caesar cipher has another kind of Python instruction—the if statement:

```
23.     if symbol in SYMBOLS:
```

You can read an if statement as, "If this condition is True, execute the code in the following block. Otherwise, if it is False, skip the block." An if statement is formatted using the keyword if followed by a condition, followed by a colon (:). The code to execute is indented in a block just as with loops.

An Example if Statement

Let's try an example of an `if` statement. Open a new file editor window, enter the following code, and save it as *checkPw.py*:

checkPw.py

```
print('Enter your password.')
❶ typedPassword = input()
❷ if typedPassword == 'swordfish':
❸     print('Access Granted')
❹ print('Done')
```

When you run this program, it displays the text `Enter your password.` and lets the user type in a password. The password is then stored in the variable `typedPassword` ❶. Next, the `if` statement checks whether the password is equal to the string `'swordfish'` ❷. If it is, the execution moves inside the block following the `if` statement to display the text `Access Granted` to the user ❸; otherwise, if `typedPassword` isn't equal to `'swordfish'`, the execution skips the `if` statement's block. Either way, the execution continues on to the code after the `if` block to display `Done` ❹.

The else Statement

Often, we want to test a condition and execute one block of code if the condition is `True` and another block of code if it's `False`. We can use an `else` statement after an `if` statement's block, and the `else` statement's block of code will be executed if the `if` statement's condition is `False`. For an `else` statement, you just write the keyword `else` and a colon (`:`). It doesn't need a condition because it will be run if the `if` statement's condition isn't true. You can read the code as, "If this condition is `True`, execute this block, or else, if it is `False`, execute this other block."

Modify the *checkPw.py* program to look like the following (the new lines are in bold):

checkPw.py

```
print('Enter your password.')
typedPassword = input()
❶ if typedPassword == 'swordfish':
      print('Access Granted')
else:
❷     print('Access Denied')
❸ print('Done')
```

This version of the program works almost the same as the previous version. The text `Access Granted` will still display if the `if` statement's condition is `True` ❶. But now if the user types something other than `swordfish`, the `if` statement's condition will be `False`, causing the execution to enter the `else` statement's block and display `Access Denied` ❷. Either way, the execution will still continue and display `Done` ❸.

The elif Statement

Another statement, called the `elif` statement, can also be paired with `if`. Like an `if` statement, it has a condition. Like an `else` statement, it follows an `if` (or another `elif`) statement and executes if the previous `if` (or `elif`) statement's condition is `False`. You can read `if`, `elif`, and `else` statements as, "If this condition is `True`, run this block. Or else, check if this next condition is `True`. Or else, just run this last block." Any number of `elif` statements can follow an `if` statement. Modify the *checkPw.py* program again to make it look like the following:

checkPw.py

```
print('Enter your password.')
typedPassword = input()
❶ if typedPassword == 'swordfish':
❷     print('Access Granted')
❸ elif typedPassword == 'mary':
      print('Hint: the password is a fish.')
❹ elif typedPassword == '12345':
      print('That is a really obvious password.')
else:
      print('Access Denied')
print('Done')
```

This code contains four blocks for the `if`, `elif`, and `else` statements. If the user enters `12345`, then `typedPassword == 'swordfish'` evaluates to `False` ❶, so the first block with `print('Access Granted')` ❷ is skipped. The execution next checks the `typedPassword == 'mary'` condition, which also evaluates to `False` ❸, so the second block is also skipped. The `typedPassword == '12345'` condition is `True` ❹, so the execution enters the block following this `elif` statement to run the code `print('That is a really obvious password.')` and skips any remaining `elif` and `else` statements. *Notice that one and only one of these blocks will be executed.*

You can have zero or more `elif` statements following an `if` statement. You can have zero or one but not multiple `else` statements, and the `else` statement always comes last because it only executes if none of the conditions evaluate to `True`. The first statement with a `True` condition has its block executed. The rest of the conditions (even if they're also `True`) aren't checked.

The in and not in Operators

Line 23 in *caesarCipher.py* also uses the `in` operator:

```
23.     if symbol in SYMBOLS:
```

An `in` operator can connect two strings, and it will evaluate to `True` if the first string is inside the second string or evaluate to `False` if not. The `in`

operator can also be paired with `not`, which will do the opposite. Enter the following into the interactive shell:

```
>>> 'hello' in 'hello world!'
True
>>> 'hello' not in 'hello world!'
False
>>> 'ello' in 'hello world!'
True
❶ >>> 'HELLO' in 'hello world!'
False
❷ >>> '' in 'Hello'
True
```

Notice that the `in` and `not in` operators are case sensitive ❶. Also, a blank string is always considered to be in any other string ❷.

Expressions using the `in` and `not in` operators are handy to use as conditions of `if` statements to execute some code if a string exists inside another string.

Returning to *caesarCipher.py*, line 23 checks whether the string in `symbol` (which the `for` loop on line 21 set to a single character from the `message` string) is in the `SYMBOLS` string (the symbol set of all characters that can be encrypted or decrypted by this cipher program). If `symbol` is in `SYMBOLS`, the execution enters the block that follows starting on line 24. If it isn't, the execution skips this block and instead enters the block following line 39's `else` statement. The cipher program needs to run different code depending on whether the symbol is in the symbol set.

The find() String Method

Line 24 finds the index in the `SYMBOLS` string where `symbol` is:

```
24.         symbolIndex = SYMBOLS.find(symbol)
```

This code includes a method call. *Methods* are just like functions except they're attached to a value with a period (or in line 24, a variable containing a value). The name of this method is `find()`, and it's being called on the string value stored in `SYMBOLS`.

Most data types (such as strings) have methods. The `find()` method takes one string argument and returns the integer index of where the argument appears in the method's string. Enter the following into the interactive shell:

```
>>> 'hello'.find('e')
1
>>> 'hello'.find('o')
4
>>> spam = 'hello'
>>> spam.find('h')
❶ 0
```

You can use the `find()` method on either a string or a variable containing a string value. Remember that indexing in Python starts with `0`, so when the index returned by `find()` is for the first character in the string, a `0` is returned ❶.

If the string argument can't be found, the `find()` method returns the integer `-1`. Enter the following into the interactive shell:

```
>>> 'hello'.find('x')
-1
❶ >>> 'hello'.find('H')
-1
```

Notice that the `find()` method is also case sensitive ❶.

The string you pass as an argument to `find()` can be more than one character. The integer that `find()` returns will be the index of the first character where the argument is found. Enter the following into the interactive shell:

```
>>> 'hello'.find('ello')
1
>>> 'hello'.find('lo')
3
>>> 'hello hello'.find('e')
1
```

The `find()` string method is like a more specific version of using the `in` operator. It not only tells you whether a string exists in another string but also tells you where.

Encrypting and Decrypting Symbols

Now that you understand `if`, `elif`, and `else` statements; the `in` operator; and the `find()` string method, it will be easier to understand how the rest of the Caesar cipher program works.

The cipher program can only encrypt or decrypt symbols that are in the symbol set:

```
23.     if symbol in SYMBOLS:
24.         symbolIndex = SYMBOLS.find(symbol)
```

So before running the code on line 24, the program must figure out whether `symbol` is in the symbol set. Then it can find the index in `SYMBOLS` where `symbol` is located. The index returned by the `find()` call is stored in `symbolIndex`.

Now that we have the current symbol's index stored in `symbolIndex`, we can do the encryption or decryption math on it. The Caesar cipher adds the key number to the symbol's index to encrypt it or subtracts the key number

from the symbol's index to decrypt it. This value is stored in `translatedIndex` because it will be the index in `SYMBOLS` of the translated symbol.

caesarCipher.py

```
26.         # Perform encryption/decryption:
27.         if mode == 'encrypt':
28.             translatedIndex = symbolIndex + key
29.         elif mode == 'decrypt':
30.             translatedIndex = symbolIndex - key
```

The `mode` variable contains a string that tells the program whether it should be encrypting or decrypting. If this string is `'encrypt'`, then the condition for line 27's `if` statement will be `True`, and line 28 will be executed to add the `key` to `symbolIndex` (and the block after the `elif` statement will be skipped). Otherwise, if `mode` is `'decrypt'`, then line 30 is executed to subtract the `key`.

Handling Wraparound

When we were implementing the Caesar cipher with paper and pencil in Chapter 1, sometimes adding or subtracting the key would result in a number greater than or equal to the size of the symbol set or less than zero. In those cases, we have to add or subtract the length of the symbol set so that it will "wrap around," or return to the beginning or end of the symbol set. We can use the code `len(SYMBOLS)` to do this, which returns `66`, the length of the `SYMBOLS` string. Lines 33 to 36 handle this wraparound in the cipher program.

```
32.         # Handle wraparound, if needed:
33.         if translatedIndex >= len(SYMBOLS):
34.             translatedIndex = translatedIndex - len(SYMBOLS)
35.         elif translatedIndex < 0:
36.             translatedIndex = translatedIndex + len(SYMBOLS)
```

If `translatedIndex` is greater than or equal to `66`, the condition on line 33 is `True` and line 34 is executed (and the `elif` statement on line 35 is skipped). Subtracting the length of `SYMBOLS` from `translatedIndex` points the index of the variable back to the beginning of the `SYMBOLS` string. Otherwise, Python will check whether `translatedIndex` is less than `0`. If that condition is `True`, line 36 is executed, and `translatedIndex` wraps around to the end of the `SYMBOLS` string.

You might be wondering why we didn't just use the integer value `66` directly instead of `len(SYMBOLS)`. By using `len(SYMBOLS)` instead of `66`, we can add to or remove symbols from `SYMBOLS` and the rest of the code will still work.

Now that you have the index of the translated symbol in `translatedIndex`, `SYMBOLS[translatedIndex]` will evaluate to the translated symbol. Line 38 adds this encrypted/decrypted symbol to the end of the `translated` string using string concatenation:

```
38.         translated = translated + SYMBOLS[translatedIndex]
```

Eventually, the `translated` string will be the whole encoded or decoded message.

Handling Symbols Outside of the Symbol Set

The `message` string might contain characters that are not in the `SYMBOLS` string. These characters are outside of the cipher program's symbol set and can't be encrypted or decrypted. Instead, they will just be appended to the `translated` string as is, which happens in lines 39 to 41:

```
39.     else:
40.         # Append the symbol without encrypting/decrypting:
41.         translated = translated + symbol
```

The `else` statement on line 39 has four spaces of indentation. If you look at the indentation of the lines above, you'll see that it's paired with the `if` statement on line 23. Although there's a lot of code in between this `if` and `else` statement, it all belongs in the same block of code.

If line 23's `if` statement's condition were `False`, the block would be skipped, and the program execution would enter the `else` statement's block starting at line 41. This `else` block has just one line in it. It adds the unchanged `symbol` string to the end of `translated`. As a result, symbols outside of the symbol set, such as `'%'` or `'('`, are added to the translated string without being encrypted or decrypted.

Displaying and Copying the Translated String

Line 43 has no indentation, which means it's the first line after the block that started on line 21 (the `for` loop's block). By the time the program execution reaches line 44, it has looped through each character in the `message` string, encrypted (or decrypted) the characters, and added them to `translated`:

```
43. # Output the translated string:
44. print(translated)
45. pyperclip.copy(translated)
```

Line 44 calls the `print()` function to display the `translated` string on the screen. Notice that this is the only `print()` call in the entire program. The computer does a lot of work encrypting every letter in `message`, handling wraparound, and handling non-letter characters. But the user doesn't need to see this. The user just needs to see the final string in `translated`.

Line 45 calls `copy()`, which takes one string argument and copies it to the clipboard. Because `copy()` is a function in the `pyperclip` module, we must tell Python this by putting `pyperclip.` in front of the function name. If we type `copy(translated)` instead of `pyperclip.copy(translated)`, Python will give us an error message because it won't be able to find the function.

Python will also give an error message if you forget the `import pyperclip` line (line 4) before trying to call `pyperclip.copy()`.

That's the entire Caesar cipher program. When you run it, notice how your computer can execute the entire program and encrypt the string in less than a second. Even if you enter a very long string to store in the `message`

variable, your computer can encrypt or decrypt the message within a second or two. Compare this to the several minutes it would take to do this with a cipher wheel. The program even automatically copies the encrypted text to the clipboard so the user can simply paste it into an email to send to someone.

Encrypting Other Symbols

One problem with the Caesar cipher that we've implemented is that it can't encrypt characters outside its symbol set. For example, if you encrypt the string `'Be sure to bring the $$$.'` with the key `20`, the message will encrypt to `'VyQ?A!yQ.9Qv!381Q.2yQ$$$T'`. This encrypted message doesn't hide that you are referring to $$$. However, we can modify the program to encrypt other symbols.

By changing the string that is stored in `SYMBOLS` to include more characters, the program will encrypt them as well, because on line 23, the condition `symbol in SYMBOLS` will be `True`. The value of `symbolIndex` will be the index of `symbol` in this new, larger `SYMBOLS` constant variable. The "wraparound" will need to add or subtract the number of characters in this new string, but that's already handled because we use `len(SYMBOLS)` instead of typing `66` directly into the code (which is why we programmed it this way).

For example, you could expand line 16 to be:

```
SYMBOLS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz12345
  67890 !?.`~@#$%^&*()_+-=[]{}|;:<>,/'
```

Keep in mind that a message must be encrypted and decrypted with the same symbol set to work.

Summary

You've learned several programming concepts and read through quite a few chapters to get to this point, but now you have a program that implements a secret cipher. And more important, you understand how this code works.

Modules are Python programs that contain useful functions. To use these functions, you must first import them using an `import` statement. To call functions in an imported module, put the module name and a period before the function name, like so: *`module.function`*`()`.

Constant variables are written in uppercase letters by convention. These variables are not meant to have their values changed (although nothing prevents the programmer from writing code that does so). Constants are helpful because they give a "name" to specific values in your program.

Methods are functions that are attached to a value of a certain data type. The `find()` string method returns an integer of the position of the string argument passed to it inside the string it is called on.

You learned about several new ways to manipulate which lines of code run and how many times each line runs. A `for` loop iterates over all the characters in a string value, setting a variable to each character on each iteration. The `if`, `elif`, and `else` statements execute blocks of code based on whether a condition is `True` or `False`.

The `in` and `not in` operators check whether one string is or isn't in another string and evaluate to `True` or `False` accordingly.

Knowing how to program gives you the ability to write down a process like encrypting or decrypting with the Caesar cipher in a language that a computer can understand. And once the computer understands how to execute the process, it can do it much faster than any human can and with no mistakes (unless mistakes are in your programming). Although this is an incredibly useful skill, it turns out the Caesar cipher can easily be broken by someone who knows how to program. In Chapter 6, you'll use the skills you've learned to write a Caesar cipher hacker so you can read ciphertext that other people have encrypted. Let's move on and learn how to hack encryption.

PRACTICE QUESTIONS

Answers to the practice questions can be found on the book's website at *https://www.nostarch.com/crackingcodes/*.

1. Using *caesarCipher.py*, encrypt the following sentences with the given keys:
   a. `'"You can show black is white by argument," said Filby, "but you will never convince me."'` with key `8`
   b. `'1234567890'` with key `21`
2. Using *caesarCipher.py*, decrypt the following ciphertexts with the given keys:
   a. `'Kv?uqwpfu?rncwukdng?gpqwijB'` with key `2`
   b. `'XCBSw88S18A1S 2SB41SE .8zSEwAS50D5A5x81V'` with key `22`
3. Which Python instruction would import a module named *watermelon.py*?
4. What do the following pieces of code display on the screen?
   a.

```
spam = 'foo'
for i in spam:
    spam = spam + i
print(spam)
```

(continued)

b.

```
if 10 < 5:
    print('Hello')
elif False:
    print('Alice')
elif 5 != 5:
    print('Bob')
else:
    print('Goodbye')
```

c.

```
print('f' not in 'foo')
```

d.

```
print('foo' in 'f')
```

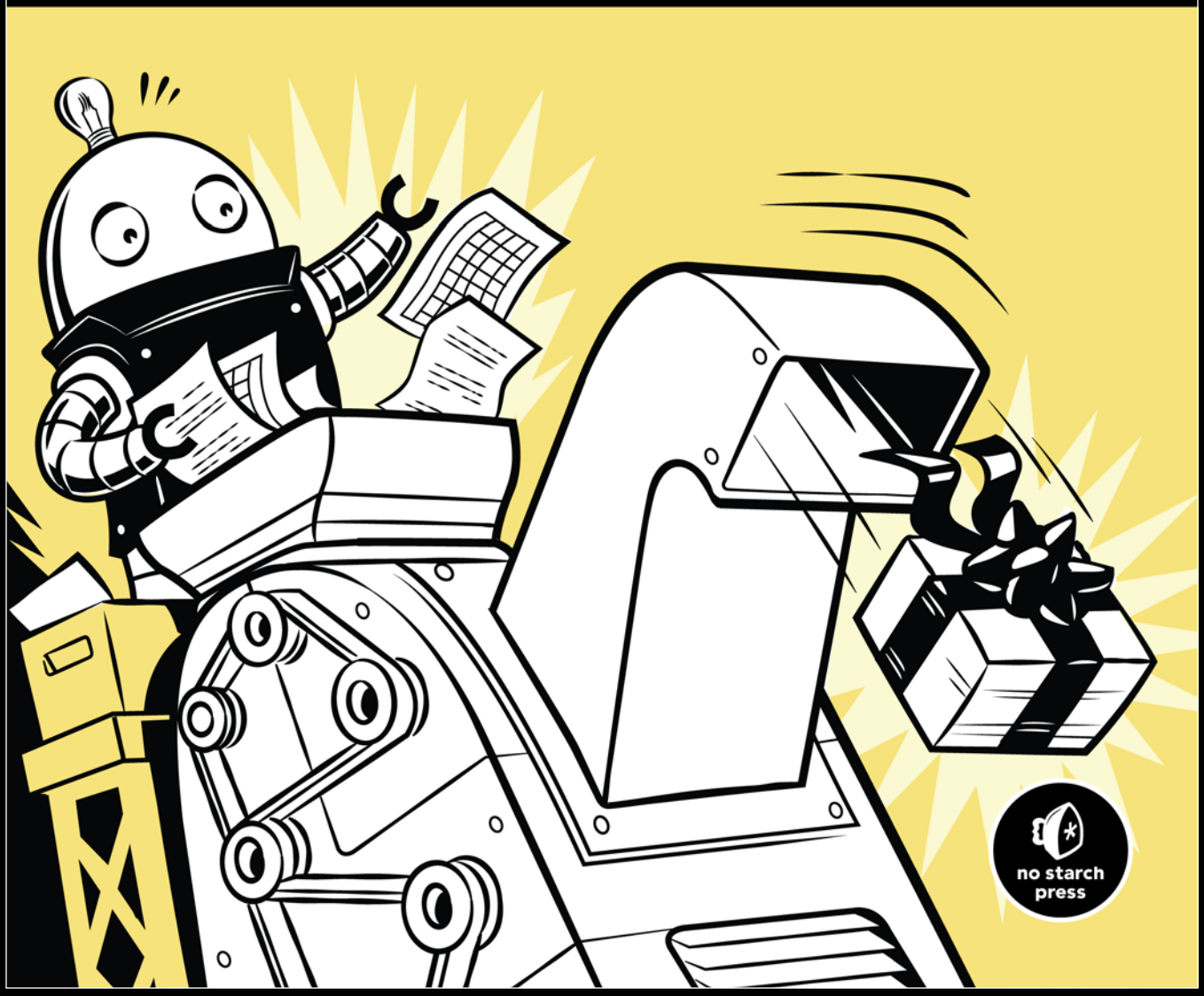
e.

```
print('hello'.find('oo'))
```

PRACTICAL SQL

A BEGINNER'S GUIDE TO *STORYTELLING* WITH *DATA*

ANTHONY DEBARROS



no starch press

1

CREATING YOUR FIRST DATABASE AND TABLE

SQL is more than just a means for extracting knowledge from data. It's also a language for *defining* the structures that hold data so we can organize *relationships* in the data. Chief among those structures is the table.

A table is a grid of rows and columns that store data. Each row holds a collection of columns, and each column contains data of a specified type: most commonly, numbers, characters, and dates. We use SQL to define the structure of a table and how each table might relate to other tables in the database. We also use SQL to extract, or *query*, data from tables.

Understanding tables is fundamental to understanding the data in your database. Whenever I start working with a fresh database, the first thing I do is look at the tables within. I look for clues in the table names and their column structure. Do the tables contain text, numbers, or both? How many rows are in each table?

Next, I look at how many tables are in the database. The simplest database might have a single table. A full-bore application that handles

customer data or tracks air travel might have dozens or hundreds. The number of tables tells me not only how much data I'll need to analyze, but also hints that I should explore relationships among the data in each table.

Before you dig into SQL, let's look at an example of what the contents of tables might look like. We'll use a hypothetical database for managing a school's class enrollment; within that database are several tables that track students and their classes. The first table, called `student_enrollment`, shows the students that are signed up for each class section:

```
student_id    class_id      class_section    semester
----------    ----------    -------------    ---------
CHRISPA004    COMPSCI101    3                Fall 2017
DAVISHE010    COMPSCI101    3                Fall 2017
ABRILDA002    ENG101        40               Fall 2017
DAVISHE010    ENG101        40               Fall 2017
RILEYPH002    ENG101        40               Fall 2017
```

This table shows that two students have signed up for `COMPSCI101`, and three have signed up for `ENG101`. But where are the details about each student and class? In this example, these details are stored in separate tables called `students` and `classes`, and each table relates to this one. This is where the power of a *relational database* begins to show itself.

The first several rows of the `students` table include the following:

```
student_id    first_name    last_name    dob
----------    ----------    ---------    ----------
ABRILDA002    Abril         Davis        1999-01-10
CHRISPA004    Chris         Park         1996-04-10
DAVISHE010    Davis         Hernandez    1987-09-14
RILEYPH002    Riley         Phelps       1996-06-15
```

The `students` table contains details on each student, using the value in the `student_id` column to identify each one. That value acts as a unique *key* that connects both tables, giving you the ability to create rows such as the following with the `class_id` column from `student_enrollment` and the `first_name` and `last_name` columns from `students`:

```
class_id      first_name    last_name
----------    ----------    ---------
COMPSCI101    Davis         Hernandez
COMPSCI101    Chris         Park
ENG101        Abril         Davis
ENG101        Davis         Hernandez
ENG101        Riley         Phelps
```

The `classes` table would work the same way, with a `class_id` column and several columns of detail about the class. Database builders prefer to organize data using separate tables for each main *entity* the database manages in order to reduce redundant data. In the example, we store each student's name and date of birth just once. Even if the student signs up for multiple

classes—as Davis Hernandez did—we don't waste database space entering his name next to each class in the `student_enrollment` table. We just include his student ID.

Given that tables are a core building block of every database, in this chapter you'll start your SQL coding adventure by creating a table inside a new database. Then you'll load data into the table and view the completed table.

Create a Database

The PostgreSQL program you downloaded in the Introduction is a *database management system*, a software package that allows you to define, manage, and query databases. When you installed PostgreSQL, it created a *database server*—an instance of the application running on your computer—that includes a default database called `postgres`. The database is a collection of objects that includes tables, functions, user roles, and much more. According to the PostgreSQL documentation, the default database is "meant for use by users, utilities and third party applications" (see *https://www.postgresql.org/docs/current/static/app-initdb.html*). In the exercises in this chapter, we'll leave the default as is and instead create a new one. We'll do this to keep objects related to a particular topic or application organized together.

To create a database, you use just one line of SQL, shown in Listing 1-1. This code, along with all the examples in this book, is available for download via the resources at *https://www.nostarch.com/practicalSQL/*.

```
CREATE DATABASE analysis;
```

Listing 1-1: Creating a database named `analysis`

This statement creates a database on your server named `analysis` using default PostgreSQL settings. Note that the code consists of two keywords—`CREATE` and `DATABASE`—followed by the name of the new database. The statement ends with a semicolon, which signals the end of the command. The semicolon ends all PostgreSQL statements and is part of the ANSI SQL standard. Sometimes you can omit the semicolon, but not always, and particularly not when running multiple statements in the admin. So, using the semicolon is a good habit to form.

Executing SQL in pgAdmin

As part of the Introduction to this book, you also installed the graphical administrative tool pgAdmin (if you didn't, go ahead and do that now). For much of our work, you'll use pgAdmin to run (or execute) the SQL statements we write. Later in the book in Chapter 16, I'll show you how to run SQL statements in a terminal window using the PostgreSQL command-line program `psql`, but getting started is a bit easier with a graphical interface.

We'll use pgAdmin to run the SQL in Listing 1-1 that creates the database. Then, we'll connect to the new database and create a table. Follow these steps:

1. Run PostgreSQL. If you're using Windows, the installer set PostgreSQL to launch every time you boot up. On macOS, you must double-click *Postgres.app* in your Applications folder.
2. Launch pgAdmin. As you did in the Introduction, in the left vertical pane (the object browser) expand the plus sign to the left of the Servers node to show the default server. Depending on how you installed PostgreSQL, the default server may be named *localhost* or *PostgreSQL x*, where *x* is the version of the application.
3. Double-click the server name. If you supplied a password during installation, enter it at the prompt. You'll see a brief message that pgAdmin is establishing a connection.
4. In pgAdmin's object browser, expand **Databases** and click once on the `postgres` database to highlight it, as shown in Figure 1-1.
5. Open the Query Tool by choosing **Tools ▸ Query Tool**.
6. In the SQL Editor pane (the top horizontal pane), type or copy the code from Listing 1-1.
7. Click the lightning bolt icon to execute the SQL. PostgreSQL creates the database, and in the Output pane in the Query Tool under Messages you'll see a notice indicating the query returned successfully, as shown in Figure 1-2.

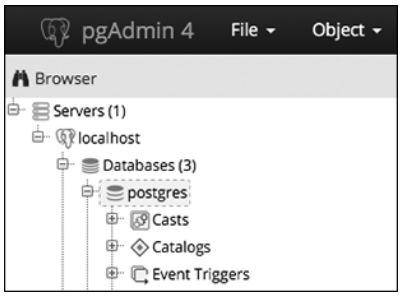


Figure 1-1: Connecting to the default `postgres` database

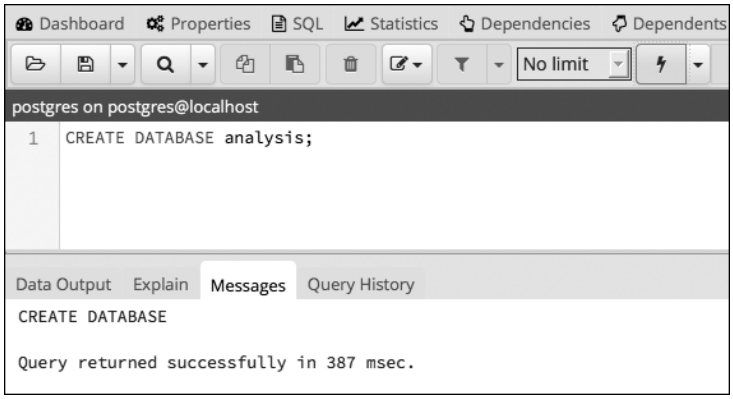


Figure 1-2: Creating the `analysis` database

8. To see your new database, right-click **Databases** in the object browser. From the pop-up menu, select **Refresh**, and the `analysis` database will appear in the list, as shown in Figure 1-3.

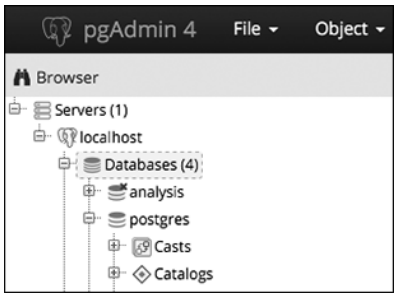


Figure 1-3: The `analysis` database displayed in the object browser

Good work! You now have a database called `analysis`, which you can use for the majority of the exercises in this book. In your own work, it's generally a best practice to create a new database for each project to keep tables with related data together.

Connecting to the Analysis Database

Before you create a table, you must ensure that pgAdmin is connected to the `analysis` database rather than to the default `postgres` database.

To do that, follow these steps:

1. Close the Query Tool by clicking the **X** at the top right of the tool.
2. In the object browser, click once on the `analysis` database.
3. Reopen the Query Tool by choosing **Tools ▸ Query Tool**.
4. You should now see the label `analysis on postgres@localhost` at the top of the Query Tool window. (Again, instead of `localhost`, your version may show `PostgreSQL`.)

Now, any code you execute will apply to the `analysis` database.

Create a Table

As I mentioned earlier, tables are where data lives and its relationships are defined. When you create a table, you assign a name to each *column* (sometimes referred to as a *field* or *attribute*) and assign it a *data type*. These are the values the column will accept—such as text, integers, decimals, and dates—and the definition of the data type is one way SQL enforces the integrity of data. For example, a column defined as `date` will take data in one of several standard formats, such as `YYYY-MM-DD`. If you try to enter characters not in a date format, for instance, the word `peach`, you'll receive an error.

Data stored in a table can be accessed and analyzed, or queried, with SQL statements. You can sort, edit, and view the data, and easily alter the table later if your needs change.

Let's make a table in the `analysis` database.

The CREATE TABLE Statement

For this exercise, we'll use an often-discussed piece of data: teacher salaries. Listing 1-2 shows the SQL to create a table called `teachers`:

```
❶ CREATE TABLE teachers (
❷     id bigserial,
❸     first_name varchar(25),
      last_name varchar(50),
      school varchar(50),
❹     hire_date date,
❺     salary numeric
❻ );
```

Listing 1-2: Creating a table named `teachers` with six columns

This table definition is far from comprehensive. For example, it's missing several *constraints* that would ensure that columns that must be filled do indeed have data or that we're not inadvertently entering duplicate values. I cover constraints in detail in Chapter 7, but in these early chapters I'm omitting them to focus on getting you started on exploring data.

The code begins with the two SQL keywords ❶ `CREATE` and `TABLE` that, together with the name `teachers`, signal PostgreSQL that the next bit of code describes a table to add to the database. Following an opening parenthesis, the statement includes a comma-separated list of column names along with their data types. For style purposes, each new line of code is on its own line and indented four spaces, which isn't required, but it makes the code more readable.

Each column name represents one discrete data element defined by a data type. The `id` column ❷ is of data type `bigserial`, a special integer type that auto-increments every time you add a row to the table. The first row receives the value of `1` in the `id` column, the second row `2`, and so on. The `bigserial` data type and other serial types are PostgreSQL-specific implementations, but most database systems have a similar feature.

Next, we create columns for the teacher's first and last name, and the school where they teach ❸. Each is of the data type `varchar`, a text column with a maximum length specified by the number in parentheses. We're assuming that no one in the database will have a last name of more than 50 characters. Although this is a safe assumption, you'll discover over time that exceptions will always surprise you.

The teacher's `hire_date` ❹ is set to the data type `date`, and the `salary` column ❺ is a `numeric`. I'll cover data types more thoroughly in Chapter 3, but this table shows some common examples of data types. The code block wraps up ❻ with a closing parenthesis and a semicolon.

Now that you have a sense of how SQL looks, let's run this code in pgAdmin.

Making the teachers Table

You have your code and you're connected to the database, so you can make the table using the same steps we did when we created the database:

1. Open the pgAdmin Query Tool (if it's not open, click once on the `analysis` database in pgAdmin's object browser, and then choose **Tools ▸ Query Tool**).
2. Copy the `CREATE TABLE` script from Listing 1-2 into the SQL Editor.
3. Execute the script by clicking the lightning bolt icon.

If all goes well, you'll see a message in the pgAdmin Query Tool's bottom output pane that reads, `Query returned successfully with no result in 84 msec`. Of course, the number of milliseconds will vary depending on your system.

Now, find the table you created. Go back to the main pgAdmin window and, in the object browser, right-click the `analysis` database and choose **Refresh**. Choose **Schemas ▸ public ▸ Tables** to see your new table, as shown in Figure 1-4.

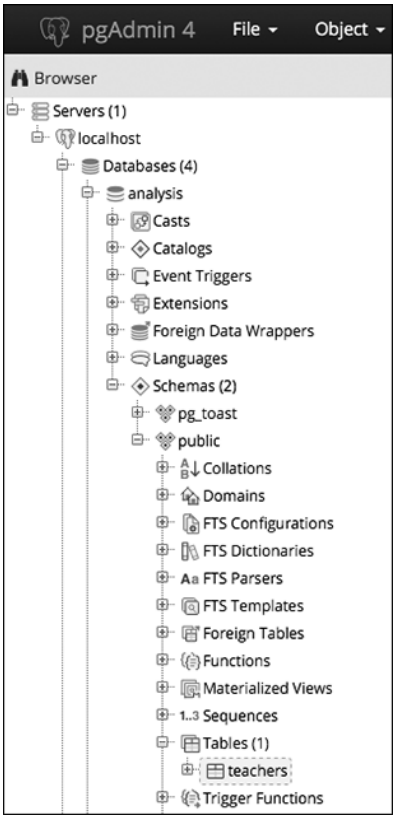


Figure 1-4: The `teachers` table in the object browser

Expand the `teachers` table node by clicking the plus sign to the left of its name. This reveals more details about the table, including the column names, as shown in Figure 1-5. Other information appears as well, such as indexes, triggers, and constraints, but I'll cover those in later chapters. Clicking on the table name and then selecting the **SQL** menu in the pgAdmin workspace will display the SQL used to make the `teachers` table.

Congratulations! So far, you've built a database and added a table to it. The next step is to add data to the table so you can write your first query.

Insert Rows into a Table

You can add data to a PostgreSQL table in several ways. Often, you'll work with a large number of rows, so the easiest method is to import data from a text file or another database directly into a table. But just to get started, we'll add a few rows using an `INSERT INTO ... VALUES` statement that specifies the target columns and the data values. Then we'll view the data in its new home.

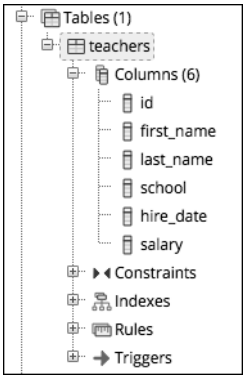


Figure 1-5: Table details for teachers

The INSERT Statement

To insert some data into the table, you first need to erase the `CREATE TABLE` statement you just ran. Then, following the same steps as you did to create the database and table, copy the code in Listing 1-3 into your pgAdmin Query Tool:

```
❶ INSERT INTO teachers (first_name, last_name, school, hire_date, salary)
❷ VALUES ('Janet', 'Smith', 'F.D. Roosevelt HS', '2011-10-30', 36200),
         ('Lee', 'Reynolds', 'F.D. Roosevelt HS', '1993-05-22', 65000),
         ('Samuel', 'Cole', 'Myers Middle School', '2005-08-01', 43500),
         ('Samantha', 'Bush', 'Myers Middle School', '2011-10-30', 36200),
         ('Betty', 'Diaz', 'Myers Middle School', '2005-08-30', 43500),
         ('Kathleen', 'Roush', 'F.D. Roosevelt HS', '2010-10-22', 38500);❸
```

Listing 1-3: Inserting data into the teachers table

This code block inserts names and data for six teachers. Here, the PostgreSQL syntax follows the ANSI SQL standard: after the `INSERT INTO` keywords is the name of the table, and in parentheses are the columns to be filled ❶. In the next row is the `VALUES` keyword and the data to insert into each column in each row ❷. You need to enclose the data for each row in a set of parentheses, and inside each set of parentheses, use a comma to separate each column value. The order of the values must also match the order of the columns specified after the table name. Each row of data ends with a comma, and the last row ends the entire statement with a semicolon ❸.

Notice that certain values that we're inserting are enclosed in single quotes, but some are not. This is a standard SQL requirement. Text and dates require quotes; numbers, including integers and decimals, don't require quotes. I'll highlight this requirement as it comes up in examples. Also, note the date format we're using: a four-digit year is followed by the month and date, and each part is joined by a hyphen. This is the international standard for date formats; using it will help you avoid confusion. (Why is it best to use the format `YYYY-MM-DD`? Check out *https://xkcd.com/1179/* to see a great comic about it.) PostgreSQL supports many additional date formats, and I'll use several in examples.

You might be wondering about the `id` column, which is the first column in the table. When you created the table, your script specified that column

to be the `serial` data type. So as PostgreSQL inserts each row, it automatically fills the `id` column with an auto-incrementing integer. I'll cover that in detail in Chapter 3 when I discuss data types.

Now, run the code. This time the message in the Query Tool should include the words `Query returned successfully: 6 rows affected.`

Viewing the Data

You can take a quick look at the data you just loaded into the `teachers` table using pgAdmin. In the object browser, locate the table and right-click. In the pop-up menu, choose **View/Edit Data ▸ All Rows**. As Figure 1-6 shows, you'll see the six rows of data in the table with each column filled by the values in the SQL statement.

Data Output | Explain | Messages | Query History

	id bigint	first_name character vary	last_name character vary	school character varying	hire_date date	salary numeric
1	1	Janet	Smith	F.D. Roosevelt ...	2011-10-30	36200
2	2	Lee	Reynolds	F.D. Roosevelt ...	1993-05-22	65000
3	3	Samuel	Cole	Myers Middle S...	2005-08-01	43500
4	4	Samantha	Bush	Myers Middle S...	2011-10-30	36200
5	5	Betty	Diaz	Myers Middle S...	2005-08-30	43500
6	6	Kathleen	Roush	F.D. Roosevelt ...	2010-10-22	38500

Figure 1-6: Viewing table data directly in pgAdmin

Notice that even though you didn't insert a value for the `id` column, each teacher has an id number assigned.

pgAdmin lets you view data using this interface in a few ways, but we'll focus on writing SQL to handle those tasks.

When Code Goes Bad

There may be a universe where code always works, but unfortunately, we haven't invented a machine capable of transporting us there. Errors happen. Whether you make a typo or mix up the order of operations, computer languages are unforgiving about syntax. For example, if you forget a comma in the code in Listing 1-3, PostgreSQL squawks back an error:

```
ERROR:  syntax error at or near "("
LINE 5:     ('Samuel', 'Cole', 'Myers Middle School', '2005-08-01', 43...
            ^
********** Error **********
```

Fortunately, the error message hints at what's wrong and where: a syntax error is near an open parenthesis on line 5. But sometimes error messages can be more obscure. In that case, you do what the best coders do: a quick internet search for the error message. Most likely, someone else has experienced the same issue and might know the answer.

Formatting SQL for Readability

SQL requires no special formatting to run, so you're free to use your own psychedelic style of uppercase, lowercase, and random indentations. But that won't win you any friends when others need to work with your code (and sooner or later someone will). For the sake of readability and being a good coder, it's best to follow these conventions:

- Uppercase SQL keywords, such as `SELECT`. Some SQL coders also uppercase the names of data types, such as `TEXT` and `INTEGER`. I use lowercase characters for data types in this book to separate them in your mind from keywords, but you can uppercase them if desired.
- Avoid `CamelCase` and instead use `lowercase_and_underscores` for object names, such as tables and column names (see more details about case in Chapter 7).
- Indent clauses and code blocks for readability using either two or four spaces. Some coders prefer tabs to spaces; use whichever works best for you or your organization.

We'll explore other SQL coding conventions as we go through the book, but these are the basics.

Wrapping Up

You accomplished quite a bit in this first chapter: you created a database and a table, and then loaded data into it. You're on your way to adding SQL to your data analysis toolkit! In the next chapter, you'll use this set of teacher data to learn the basics of querying a table using `SELECT`.

Try It Yourself

Here are two exercises to help you explore concepts related to databases, tables, and data relationships:

1. Imagine you're building a database to catalog all the animals at your local zoo. You want one table to track the kinds of animals in the collection and another table to track the specifics on each animal. Write `CREATE TABLE` statements for each table that include some of the columns you need. Why did you include the columns you chose?
2. Now create `INSERT` statements to load sample data into the tables. How can you view the data via the pgAdmin tool? Create an additional `INSERT` statement for one of your tables. Purposely omit one of the required commas separating the entries in the `VALUES` clause of the query. What is the error message? Would it help you find the error in the code?

Serious Cryptography

A Practical Introduction to Modern Encryption

Jean-Philippe Aumasson

Foreword by Matthew D. Green

no starch press

4

BLOCK CIPHERS

During the Cold War, the US and Soviets developed their own ciphers. The US government created the Data Encryption Standard (DES), which was adopted as a federal standard from 1979 to 2005, while the KGB developed GOST 28147-89, an algorithm kept secret until 1990 and still used today. In 2000, the US-based National Institute of Standards and Technology (NIST) selected the successor to DES, called the *Advanced Encryption Standard (AES)*, an algorithm developed in Belgium and now found in most electronic devices. AES, DES, and GOST 28147-89 have something in common: they're all *block ciphers*, a type of cipher that combines a core algorithm working on blocks of data with a mode of operation, or a technique to process sequences of data blocks.

This chapter reviews the core algorithms that underlie block ciphers, discusses their modes of operation, and explains how they all work together. It also discusses how AES works and concludes with coverage of a classic attack tool from the 1970s, the meet-in-the-middle attack, and a favorite attack technique of the 2000s—padding oracles.

What Is a Block Cipher?

A block cipher consists of an encryption algorithm and a decryption algorithm:

- The *encryption algorithm* (**E**) takes a key, K, and a plaintext block, P, and produces a ciphertext block, C. We write an encryption operation as $C = \mathbf{E}(K, P)$.
- The *decryption algorithm* (**D**) is the inverse of the encryption algorithm and decrypts a message to the original plaintext, P. This operation is written as $P = \mathbf{D}(K, C)$.

Since they're the inverse of each other, the encryption and decryption algorithms usually involve similar operations.

Security Goals

If you've followed earlier discussions about encryption, randomness, and indistinguishability, the definition of a secure block cipher will come as no surprise. Again, we'll define security as random-lookingness, so to speak.

In order for a block cipher to be secure, it should be a *pseudorandom permutation (PRP)*, meaning that as long as the key is secret, an attacker shouldn't be able to compute an output of the block cipher from any input. That is, as long as K is secret and random from an attacker's perspective, they should have no clue about what $\mathbf{E}(K, P)$ looks like, for any given P.

More generally, attackers should be unable to discover any *pattern* in the input/output values of a block cipher. In other words, it should be impossible to tell a block cipher from a truly random permutation, given black-box access to the encryption and decryption functions for some fixed and unknown key. By the same token, they should be unable to recover a secure block cipher's secret key; otherwise, they would be able to use that key to tell the block cipher from a random permutation. Of course that also implies that attackers can't predict the plaintext that corresponds to a given ciphertext produced by the block cipher.

Block Size

Two values characterize a block cipher: the block size and the key size. Security depends on both values. Most block ciphers have either 64-bit or 128-bit blocks—DES's blocks have 64 (2^6) bits, and AES's blocks have 128 (2^7) bits. In computing, lengths that are powers of two simplify data processing, storage, and addressing. But why 2^6 and 2^7 and not 2^4 or 2^{16} bits?

For one thing, it's important that blocks are not too large in order to minimize both the length of ciphertext and the memory footprint. With regard to the length of the ciphertext, block ciphers process blocks, not bits. This means that in order to encrypt a 16-bit message when blocks are 128 bits, you'll first need to convert the message into a 128-bit block, and only then will the block cipher process it and return a 128-bit ciphertext. The wider the blocks, the longer this overhead. As for the *memory footprint*, in order to process a 128-bit block, you need at least 128 bits of memory. This is small enough to fit in the registers of most CPUs or to be implemented using dedicated hardware circuits. Blocks of 64, 128, or even 512 bits are short enough to allow for efficient implementations in most cases. But larger blocks (for example, several kilobytes long) can have a noticeable impact on the cost and performance of implementations.

When ciphertexts' length or memory footprint is critical, you may have to use 64-bit blocks, because these will produce shorter ciphertexts and consume less memory. Otherwise, 128-bit or larger blocks are better, mainly because 128-bit blocks can be processed more efficiently than 64-bit ones on modern CPUs and are also more secure. In particular, CPUs can leverage special CPU instructions in order to efficiently process one or more 128-bit blocks in parallel—for example, the Advanced Vector Extensions (AVX) family of instructions in Intel CPUs.

The Codebook Attack

While blocks shouldn't be too large, they also shouldn't be too small; otherwise, they may be susceptible to *codebook attacks*, which are attacks against block ciphers that are only efficient when smaller blocks are used. The codebook attack works like this with 16-bit blocks:

1. Get the 65536 (2^{16}) ciphertexts corresponding to each 16-bit plaintext block.
2. Build a lookup table—the *codebook*—mapping each ciphertext block to its corresponding plaintext block.
3. To decrypt an unknown ciphertext block, look up its corresponding plaintext block in the table.

When 16-bit blocks are used, the lookup table needs only $2^{16} \times 16 = 2^{20}$ bits of memory, or 128 kilobytes. With 32-bit blocks, memory needs grow to 16 gigabytes, which is still manageable. But with 64-bit blocks, you'd have to store 2^{70} bits (a zetabit, or 128 exabytes), so forget about it. Codebook attacks won't be an issue for larger blocks.

How to Construct Block Ciphers

There are hundreds of block ciphers but only a handful of techniques to construct one. First, a block cipher used in practice isn't a gigantic algorithm but a repetition of *rounds*, a short sequence of operations that is weak on its

own but strong in number. Second, there are two main techniques to construct a round: substitution–permutation networks (as in AES) and Feistel schemes (as in DES). In this section, we look at how these work, after viewing an attack that works when all rounds are identical to each other.

A Block Cipher's Rounds

Computing a block cipher boils down to computing a sequence of *rounds*. In a block cipher, a round is a basic transformation that is simple to specify and to implement, and which is iterated several times to form the block cipher's algorithm. This construction, consisting of a small component repeated many times, is simpler to implement and to analyze than a construction that would consist of a single huge algorithm.

For example, a block cipher with three rounds encrypts a plaintext by computing $C = \mathbf{R}_3(\mathbf{R}_2(\mathbf{R}_1(P)))$, where the rounds are $\mathbf{R}_1$, $\mathbf{R}_2$, and $\mathbf{R}_3$ and P is a plaintext. Each round should also have an inverse in order to make it possible for a recipient to compute back to plaintext. Specifically, $P = \mathbf{iR}_1(\mathbf{iR}_2(\mathbf{iR}_3(C)))$, where $\mathbf{iR}_1$ is the inverse of $\mathbf{R}_1$, and so on.

The round functions—$\mathbf{R}_1$, $\mathbf{R}_2$, and so on—are usually identical algorithms, but they are parameterized by a value called the *round key*. Two round functions with two distinct round keys will behave differently, and therefore will produce distinct outputs if fed with the same input.

Round keys are keys derived from the main key, K, using an algorithm called a *key schedule*. For example, $\mathbf{R}_1$ takes the round key K_1, $\mathbf{R}_2$ takes the round key K_2, and so on.

Round keys should always be different from each other in every round. For that matter, not all round keys should be equal to the key K. Otherwise, all the rounds would be identical and the block cipher would be less secure, as described next.

The Slide Attack and Round Keys

In a block cipher, no round should be identical to another round in order to avoid a *slide attack*. Slide attacks look for two plaintext/ciphertext pairs (P_1, C_1) and (P_2, C_2), where $P_2 = \mathbf{R}(P_1)$ if $\mathbf{R}$ is the cipher's round (see Figure 4-1). When rounds are identical, the relation between the two plaintexts, $P_2 = \mathbf{R}(P_1)$, implies the relation $C_2 = \mathbf{R}(C_1)$ between their respective ciphertexts. Figure 4-1 shows three rounds, but the relation $C_2 = \mathbf{R}(C_1)$ will hold no matter the number of rounds, be it 3, 10, or 100. The problem is that knowing the input and output of a single round often helps recover the key. (For details, read the 1999 paper by Biryukov and Wagner called "Advanced Slide Attacks," available at *https://www.iacr.org/archive/eurocrypt2000/1807/18070595-new.pdf*)

The use of different round keys as parameters ensures that the rounds will behave differently and thus foil slide attacks.

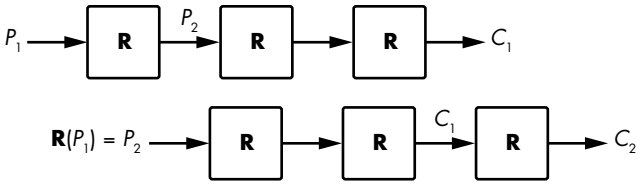


Figure 4-1: The principle of the slide attack, against block ciphers with identical rounds

NOTE *One potential byproduct and benefit of using round keys is protection against side-channel attacks, or attacks that exploit information leaked from the implementation of a cipher (for example, electromagnetic emanations). If the transformation from the main key,* K*, to a round key,* K_i*, is not invertible, then if an attacker finds* K_i*, they can't use that key to find* K*. Unfortunately, few block ciphers have a one-way key schedule. The key schedule of AES allows attackers to compute* K *from any round key,* K_i*, for example.*

Substitution–Permutation Networks

If you've read textbooks about cryptography, you'll undoubtedly have read about *confusion* and *diffusion*. Confusion means that the input (plaintext and encryption key) undergoes complex transformations, and diffusion means that these transformations depend equally on all bits of the input. At a high level, confusion is about depth whereas diffusion is about breadth. In the design of a block cipher, confusion and diffusion take the form of substitution and permutation operations, which are combined within substitution–permutation networks (SPNs).

Substitution often appears in the form of *S-boxes*, or *substitution boxes*, which are small lookup tables that transform chunks of 4 or 8 bits. For example, the first of the eight S-boxes of the block cipher Serpent is composed of the 16 elements (3 8 f 1 a 6 5 b e d 4 2 7 0 9 c), where each element represents a 4-bit nibble. This particular S-box maps the 4-bit nibble 0000 to 3 (0011), the 4-bit nibble 0101 (5 in decimal) to 6 (0110), and so on.

NOTE *S-boxes must be carefully chosen to be cryptographically strong: they should be as nonlinear as possible (inputs and outputs should be related with complex equations) and have no statistical bias (meaning, for example, that flipping an input bit should potentially affect any of the output bits).*

The permutation in a substitution–permutation network can be as simple as changing the order of the bits, which is easy to implement but doesn't mix up the bits very much. Instead of a reordering of the bits, some ciphers use basic linear algebra and matrix multiplications to mix up the bits: they perform a series of multiplication operations with fixed values (the matrix's

coefficients) and then add the results. Such linear algebra operations can quickly create dependencies between all the bits within a cipher and thus ensure strong diffusion. For example, the block cipher FOX transforms a 4-byte vector (a, b, c, d) to (a', b', c', d'), defined as follows:

$$
\begin{aligned}
a' &= a + b + c + (2 \times d) \\
b' &= a + (253 \times b) + (2 \times c) + d \\
c' &= (253 \times a) + (2 \times b) + c + d \\
d' &= (2 \times a) + b + (253 \times c) + d
\end{aligned}
$$

In the above equations, the numbers 2 and 253 are interpreted as binary polynomials rather than integers; hence, additions and multiplications are defined a bit differently than what we're used to. For example, instead of having 2 + 2 = 4, we have 2 + 2 = 0. Regardless, the point is that each byte in the initial state affects all 4 bytes in the final state.

Feistel Schemes

In the 1970s, IBM engineer Horst Feistel designed a block cipher called Lucifer that works as follows:

1. Split the 64-bit block into two 32-bit halves, L and R.
2. Set L to $L \oplus \mathbf{F}(R)$, where **F** is a substitution–permutation round.
3. Swap the values of L and R.
4. Go to step 2 and repeat 15 times.
5. Merge L and R into the 64-bit output block.

This construction became known as a *Feistel scheme*, as shown in Figure 4-2. The left side is the scheme as just described; the right side is a functionally equivalent representation where, instead of swapping L and R, rounds alternate the operations $L = L \oplus \mathbf{F}(R)$ and $R = R \oplus \mathbf{F}(L)$.

I've omitted the keys from Figure 4-2 to simplify the diagrams, but note that the first **F** takes a first round key, K_1, and the second **F** takes another round key, K_2. In DES, the **F** functions take a 48-bit round key, which is derived from the 56-bit key, K.

In a Feistel scheme, the **F** function can be either a pseudorandom permutation (PRP) or a pseudorandom function (PRF). A PRP yields distinct outputs for any two distinct inputs, whereas a PRF will have values X and Y for which $\mathbf{F}(X) = \mathbf{F}(Y)$. But in a Feistel scheme, that difference doesn't matter as long as **F** is cryptographically strong.

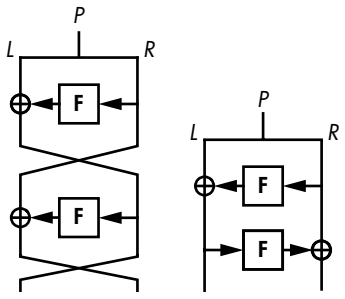


Figure 4-2: The Feistel scheme block cipher construction in two equivalent forms

How many rounds should there be in a Feistel scheme? Well, DES performs 16 rounds, whereas GOST 28147-89 performs 32 rounds. If the **F** function is as strong as possible, four rounds are in theory sufficient, but real ciphers use more rounds to defend against potential weaknesses in **F**.

The Advanced Encryption Standard (AES)

AES is the most-used cipher in the universe. Prior to the adoption of AES, the standard cipher in use was DES, with its ridiculous 56-bit security, as well as the upgraded version of DES known as Triple DES, or 3DES.

Although 3DES provides a higher level of security (112-bit security), it's inefficient because the key needs to be 168 bits long in order to get 112-bit security, and it's slow in software (DES was created to be fast in integrated circuits, not on mainstream CPUs). AES fixes both issues.

NIST standardized AES in 2000 as a replacement for DES, at which point it became the world's de facto encryption standard. Most commercial encryption products today support AES, and the NSA has approved it for protecting top-secret information. (Some countries do prefer to use their own cipher, largely because they don't want to use a US standard, but AES is actually more Belgian than it is American.)

NOTE *AES used to be called Rijndael (a portmanteau for its inventors' names, Rijmen and Daemen, pronounced like "rain-dull") when it was one of the 15 candidates in the AES competition, the process held by NIST from 1997 to 2000 to specify "an unclassified, publicly disclosed encryption algorithm capable of protecting sensitive government information well into the next century," as stated in the 1997 announcement of the competition in the Federal Register. The AES competition was kind of a "Got Talent" competition for cryptographers, where anyone could participate by submitting a cipher or breaking other contestants' ciphers.*

AES Internals

AES processes blocks of 128 bits using a secret key of 128, 192, or 256 bits, with the 128-bit key being the most common because it makes encryption slightly faster and because the difference between 128- and 256-bit security is meaningless for most applications.

Whereas some ciphers work with individual bits or 64-bit words, AES manipulates *bytes*. It views a 16-byte plaintext as a two-dimensional array of bytes ($s = s_0, s_1, \ldots, s_{15}$), as shown in Figure 4-3. (The letter s is used because this array is called the *internal state*, or just *state*.) AES transforms the bytes, columns, and rows of this array to produce a final value that is the ciphertext.

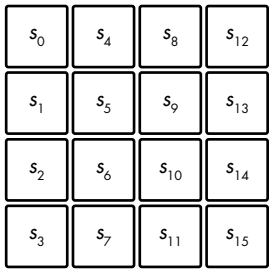


Figure 4-3: The internal state of AES viewed as a 4 × 4 array of 16 bytes

In order to transform its state, AES uses an SPN structure like the one shown in Figure 4-4, with 10 rounds for 128-bit keys, 12 for 192-bit keys, and 14 for 256-bit keys.

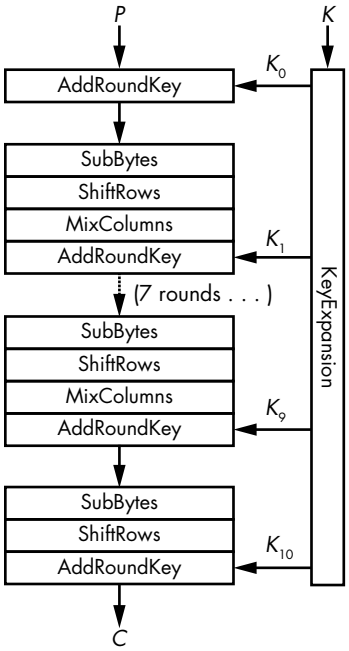


Figure 4-4: The internal operations of AES

Figure 4-4 shows the four building blocks of an AES round (note that all but the last round are a sequence of SubBytes, ShiftRows, MixColumns, and AddRoundKey):

AddRoundKey XORs a round key to the internal state.

SubBytes Replaces each byte (s_0, s_1, . . . , s_{15}) with another byte according to an S-box. In this example, the S-box is a lookup table of 256 elements.

ShiftRows Shifts the ith row of i positions, for i ranging from 0 to 3 (see Figure 4-5).

MixColumns Applies the same linear transformation to each of the four columns of the state (that is, each group of cells with the same shade of gray, as shown on the left side of Figure 4-5).

Remember that in an SPN, the *S* stands for substitution and the *P* for permutation. Here, the substitution layer is SubBytes and the permutation layer is the combination of ShiftRows and MixColumns.

The key schedule function *KeyExpansion*, shown in Figure 4-4, is the AES key schedule algorithm. This expansion creates 11 round keys (K_0, K_1, . . . , K_{10}) of 16 bytes each from the 16-byte key, using the same S-box as SubBytes and a combination of XORs. One important property of

KeyExpansion is that given any round key, K_i, an attacker can determine all other round keys as well as the main key, K, by reversing the algorithm. The ability to get the key from any round key is usually seen as an imperfect defense against side-channel attacks, where an attacker may easily recover a round key.

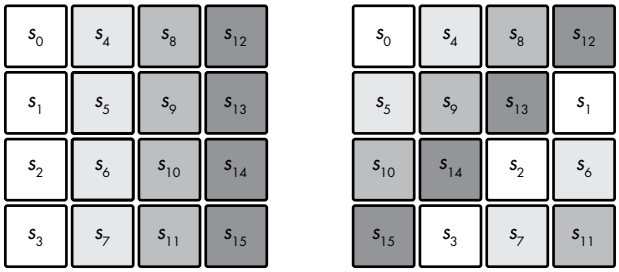


Figure 4-5: ShiftRows rotates bytes within each row of the internal state.

Without these operations, AES would be totally insecure. Each operation contributes to AES's security in a specific way:

- Without KeyExpansion, all rounds would use the same key, K, and AES would be vulnerable to slide attacks.
- Without AddRoundKey, encryption wouldn't depend on the key; hence, anyone could decrypt any ciphertext without the key.
- SubBytes brings nonlinear operations, which add cryptographic strength. Without it, AES would just be a large system of linear equations that is solvable using high-school algebra.
- Without ShiftRows, changes in a given column would never affect the other columns, meaning you could break AES by building four 2^{32}-element codebooks for each column. (Remember that in a secure block cipher, flipping a bit in the input should affect all the output bits.)
- Without MixColumns, changes in a byte would not affect any other bytes of the state. A chosen-plaintext attacker could then decrypt any ciphertext after storing 16 lookup tables of 256 bytes each that hold the encrypted values of each possible value of a byte.

Notice in Figure 4-4 that the last round of AES doesn't include the MixColumns operation. That operation is omitted in order to save useless computation: because MixColumns is linear (meaning, predictable), you could cancel its effect in the very last round by combining bits in a way that doesn't depend on their value or the key. SubBytes, however, can't be inverted without the state's value being known prior to AddRoundKey.

To decrypt a ciphertext, AES unwinds each operation by taking its inverse function: the inverse lookup table of SubBytes reverses the SubBytes transformation, ShiftRow shifts in the opposite direction,

MixColumns's inverse is applied (as in the matrix inverse of the matrix encoding its operation), and AddRoundKey's XOR is unchanged because the inverse of an XOR is another XOR.

AES in Action

To try encrypting and decrypting with AES, you can use Python's cryptography library, as in Listing 4-1.

```
#!/usr/bin/env python

from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
from binascii import hexlify as hexa
from os import urandom

# pick a random 16-byte key using Python's crypto PRNG
k = urandom(16)
print "k = %s" % hexa(k)
# create an instance of AES-128 to encrypt a single block
cipher = Cipher(algorithms.AES(k), modes.ECB(), backend = default_backend())
aes_encrypt = cipher.encryptor()

# set plaintext block p to the all-zero string
p = '\x00'*16
# encrypt plaintext p to ciphertext c
c = aes_encrypt.update(p) + aes_encrypt.finalize()
print "enc(%s) = %s" % (hexa(p), hexa(c))
# decrypt ciphertext c to plaintext p
aes_decrypt = cipher.decryptor()
p = aes_decrypt.update(c) + aes_decrypt.finalize()
print "dec(%s) = %s" % (hexa(c), hexa(p))
```

Listing 4-1: Trying AES with Python's cryptography library

Running this script produces something like the following output:

```
$ ./aes_block.py
k = 2c6202f9a582668aa96d511862d8a279
enc(00000000000000000000000000000000) = 12b620bb5eddcde9a07523e59292a6d7
dec(12b620bb5eddcde9a07523e59292a6d7) = 00000000000000000000000000000000
```

You'll get different results because the key is randomized at every new execution.

Implementing AES

Real AES software works differently than the algorithm shown in Figure 4-4. You won't find production-level AES code calling a `SubBytes()` function, then a `ShiftRows()` function, and then a `MixColumns()` function because that would be inefficient. Instead, fast AES software uses special techniques called table-based implementations and native instructions.

Table-Based Implementations

Table-based implementations of AES replace the sequence SubBytes-ShiftRows-MixColumns with a combination of XORs and lookups in tables hardcoded into the program and loaded in memory at execution time. This is possible because MixColumns is equivalent to XORing four 32-bit values, where each depends on a single byte from the state and on SubBytes. Thus, you can build four tables with 256 entries each, one for each byte value, and implement the sequence SubBytes-MixColumns by looking up four 32-bit values and XORing them together.

For example, the table-based C implementation in the OpenSSL toolkit looks like Listing 4-2.

```
    /* round 1: */
    t0 = Te0[s0 >> 24] ^ Te1[(s1 >> 16) & 0xff] ^ Te2[(s2 >> 8) & 0xff] ^ Te3[s3 & 0xff] ^ rk[ 4];
    t1 = Te0[s1 >> 24] ^ Te1[(s2 >> 16) & 0xff] ^ Te2[(s3 >> 8) & 0xff] ^ Te3[s0 & 0xff] ^ rk[ 5];
    t2 = Te0[s2 >> 24] ^ Te1[(s3 >> 16) & 0xff] ^ Te2[(s0 >> 8) & 0xff] ^ Te3[s1 & 0xff] ^ rk[ 6];
    t3 = Te0[s3 >> 24] ^ Te1[(s0 >> 16) & 0xff] ^ Te2[(s1 >> 8) & 0xff] ^ Te3[s2 & 0xff] ^ rk[ 7];
    /* round 2: */
    s0 = Te0[t0 >> 24] ^ Te1[(t1 >> 16) & 0xff] ^ Te2[(t2 >> 8) & 0xff] ^ Te3[t3 & 0xff] ^ rk[ 8];
    s1 = Te0[t1 >> 24] ^ Te1[(t2 >> 16) & 0xff] ^ Te2[(t3 >> 8) & 0xff] ^ Te3[t0 & 0xff] ^ rk[ 9];
    s2 = Te0[t2 >> 24] ^ Te1[(t3 >> 16) & 0xff] ^ Te2[(t0 >> 8) & 0xff] ^ Te3[t1 & 0xff] ^ rk[10];
    s3 = Te0[t3 >> 24] ^ Te1[(t0 >> 16) & 0xff] ^ Te2[(t1 >> 8) & 0xff] ^ Te3[t2 & 0xff] ^ rk[11];
--snip--
```

Listing 4-2: The table-based C implementation of AES in OpenSSL

A basic table-based implementation of AES encryption needs four kilobytes' worth of tables because each table stores 256 32-bit values, which occupy $256 \times 32 = 8192$ bits, or one kilobyte. Decryption requires another four tables, and thus four more kilobytes. But there are tricks to reduce the storage from four kilobytes to one, or even fewer.

Alas, table-based implementations are vulnerable to *cache-timing attacks*, which exploit timing variations when a program reads or writes elements in cache memory. Depending on the relative position in cache memory of the elements accessed, access time varies. Timings thus leak information about which element was accessed, which in turn leaks information on the secrets involved.

Cache-timing attacks are difficult to avoid. One obvious solution would be to ditch lookup tables altogether by writing a program whose execution time doesn't depend on its inputs, but that's almost impossible to do and still retain the same speed, so chip manufacturers have opted for a radical solution: instead of relying on potentially vulnerable software, they rely on *hardware*.

Native Instructions

AES native instructions (AES-NI) solve the problem of cache-timing attacks on AES software implementations. To understand how AES-NI works, you need to think about the way software runs on hardware: to run a program, a

microprocessor translates binary code into a series of instructions executed by integrated circuit components. For example, a `MUL` assembly instruction between two 32-bit values will activate the transistors implementing a 32-bit multiplier in the microprocessor. To implement a crypto algorithm, we usually just express a combination of such basic operations—additions, multiplications, XORs, and so on—and the microprocessor activates its adders, multipliers, and XOR circuits in the prescribed order.

AES native instructions take this to a whole new level by providing developers with dedicated assembly instructions that compute AES. Instead of coding an AES round as a sequence of assembly instructions, when using AES-NI, you just call the instruction `AESENC` and the chip will compute the round for you. Native instructions allow you to just tell the processor to run an AES round instead of requiring you to program rounds as a combination of basic operations.

A typical assembly implementation of AES using native instructions looks like Listing 4-3.

```
PXOR       %xmm5,  %xmm0
AESENC     %xmm6,  %xmm0
AESENC     %xmm7,  %xmm0
AESENC     %xmm8,  %xmm0
AESENC     %xmm9,  %xmm0
AESENC     %xmm10, %xmm0
AESENC     %xmm11, %xmm0
AESENC     %xmm12, %xmm0
AESENC     %xmm13, %xmm0
AESENC     %xmm14, %xmm0
AESENCLAST %xmm15, %xmm0
```

Listing 4-3: AES native instructions

This code encrypts the 128-bit plaintext initially in the register `xmm0`, assuming that registers `xmm5` to `xmm15` hold the precomputed round keys, with each instruction writing its result into `xmm0`. The initial `PXOR` instruction XORs the first round key prior to computing the first round, and the final `AESENCLAST` instruction performs the last round slightly different from the others (MixColumns is omitted).

NOTE *AES is about ten times faster on platforms that implement native instructions, which as I write this, are virtually all laptop, desktop, and server microprocessors, as well as most mobile phones and tablets. In fact, on the latest Intel microarchitecture the `AESENC` instruction has a latency of four cycles with a reciprocal throughput of one cycle, meaning that a call to `AESENC` takes four cycles to complete and that a new call can be made every cycle. To encrypt a series of blocks consecutively it thus takes 4 × 10 = 40 cycles to complete the 10 rounds or 40 / 16 = 2.5 cycles per byte. At 2 GHz (2×10^9 cycles per second), that gives a throughput of about 736 megabytes per second. If the blocks to encrypt or decrypt are independent of each other, as certain modes of operation allow, then four blocks can be processed in parallel to take full advantage of the `AESENC` circuit in order to reach a latency of 10 cycles per block instead of 40, or about 3 gigabytes per second.*

Is AES Secure?

AES is as secure as a block cipher can be, and it will never be broken. Fundamentally, AES is secure because all output bits depend on all input bits in some complex, pseudorandom way. To achieve this, the designers of AES carefully chose each component for a particular reason—MixColumns for its maximal diffusion properties and SubBytes for its optimal non-linearity—and they have shown that this composition protects AES against whole classes of cryptanalytic attacks.

But there's no proof that AES is immune to all possible attacks. For one thing, we don't know what all possible attacks are, and we don't always know how to prove that a cipher is secure against a given attack. The only way to really gain confidence in the security of AES is to crowdsource attacks: have many skilled people attempt to break AES and, hopefully, fail to do so.

After more than 15 years and hundreds of research publications, the theoretical security of AES has only been scratched. In 2011 cryptanalysts found a way to recover an AES-128 key by performing about 2^{126} operations instead of 2^{128}, a speed-up of a factor four. But this "attack" requires an insane amount of plaintext–ciphertext pairs—about 2^{88} bits worth. In other words, it's a nice finding but not one you need to worry about.

The upshot is that you should care about a million things when implementing and deploying crypto, but AES security is not one of those. The biggest threat to block ciphers isn't in their core algorithms but in their modes of operation. When an incorrect mode is chosen, or when the right one is misused, even a strong cipher like AES won't save you.

Modes of Operation

In Chapter 1, I explained how encryption schemes combine a permutation with a mode of operation to handle messages of any length. In this section, I'll cover the main modes of operations used by block ciphers, their security and function properties, and how (not) to use them. I'll begin with the dumbest one: electronic codebook.

The Electronic Codebook (ECB) Mode

The simplest of the block cipher encryption modes is electronic codebook (ECB), which is barely a mode of operation at all. ECB takes plaintext blocks $P_1, P_2, \ldots, P_N$ and processes each independently by computing $C_1 = \mathbf{E}(K, P_1)$, $C_2 = \mathbf{E}(K, P_2)$, and so on, as shown in Figure 4-6. It's a simple operation but also an insecure one. I repeat: ECB is insecure and you should not use it!

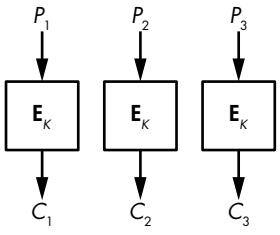


Figure 4-6: The ECB mode

Marsh Ray, a cryptographer at Microsoft, once said, "Everybody knows ECB mode is bad because we can see the penguin." He was referring to a famous illustration of ECB's insecurity

that uses an image of Linux's mascot, Tux, as shown in Figure 4-7. You can see the original image of Tux on the left, and the image encrypted in ECB mode using AES (though the underlying cipher doesn't matter) on the right. It's easy to see the penguin's shape in the encrypted version because all the blocks of one shade of gray in the original image are encrypted to the same new shade of gray in the new image; in other words, ECB encryption just gives you the same image but with different colors.

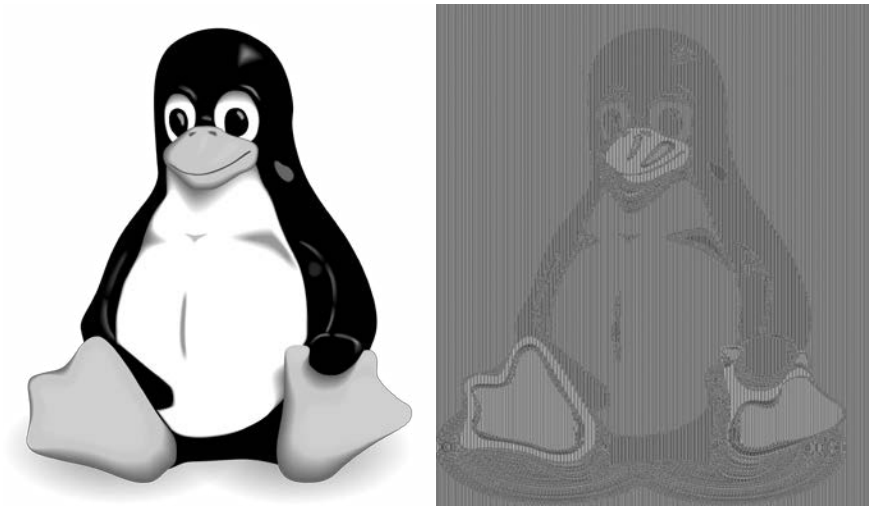


Figure 4-7: The original image (left) and the ECB-encrypted image (right)

The Python program in Listing 4-4 also shows ECB's insecurity. It picks a pseudorandom key and encrypts a 32-byte message `p` containing two blocks of null bytes. Notice that encryption yields two identical blocks and that repeating encryption with the same key and the same plaintext yields the same two blocks again.

```
#!/usr/bin/env python

from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
from binascii import hexlify as hexa
from os import urandom

BLOCKLEN = 16
def blocks(data):
    split = [hexa(data[i:i+BLOCKLEN]) for i in range(0, len(data), BLOCKLEN)]
    return ' '.join(split)

k = urandom(16)
print "k = %s" % hexa(k)

# create an instance of AES-128 to encrypt and decrypt
cipher = Cipher(algorithms.AES(k), modes.ECB(), backend=default_backend())
aes_encrypt = cipher.encryptor()
```

```
# set plaintext block p to the all-zero string
p = '\x00'*BLOCKLEN*2

# encrypt plaintext p to ciphertext c
c = aes_encrypt.update(p) + aes_encrypt.finalize()
print "enc(%s) = %s" % (blocks(p), blocks(c))
```

Listing 4-4: Using AES in ECB mode in Python

Running this script gives ciphertext blocks like this, for example:

```
$ ./aes_ecb.py
k = 50a0ebeff8001250e87d31d72a86e46d
enc(00000000000000000000000000000000 00000000000000000000000000000000) =
5eb4b7af094ef7aca472bbd3cd72f1ed 5eb4b7af094ef7aca472bbd3cd72f1ed
```

As you can see, when the ECB mode is used, identical ciphertext blocks reveal identical plaintext blocks to an attacker, whether those are blocks within a single ciphertext or across different ciphertexts. This shows that block ciphers in ECB mode aren't semantically secure.

Another problem with ECB is that it only takes complete blocks of data, so if blocks were 16 bytes, as in AES, you could only encrypt chunks of 16 bytes, 32 bytes, 48 bytes, or any other multiple of 16 bytes. There are a few ways to deal with this, as you'll see with the next mode, CBC. (I won't tell you how these tricks work with ECB because you shouldn't be using ECB in the first place.)

The Cipher Block Chaining (CBC) Mode

Cipher block chaining (CBC) is like ECB but with a small twist that makes a big difference: instead of encrypting the ith block, P_i, as $C_i = \mathbf{E}(K, P_i)$, CBC sets $C_i = \mathbf{E}(K, P_i \oplus C_{i-1})$, where C_{i-1} is the previous ciphertext block—thereby *chaining* the blocks C_{i-1} and C_i. When encrypting the first block, P_1, there is no previous ciphertext block to use, so CBC takes a random initial value (IV), as shown in Figure 4-8.

The CBC mode makes each ciphertext block dependent on all the previous blocks, and ensures that identical plaintext blocks won't be identical ciphertext blocks. The random initial value guarantees that two identical plaintexts will encrypt to distinct ciphertexts when calling the cipher twice with two distinct initial values.

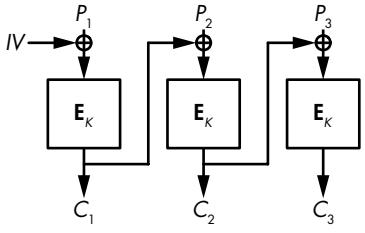


Figure 4-8: The CBC mode

Listing 4-5 illustrates these two benefits. This program takes an all-zero, 32-byte message (like the one in Listing 4-4), encrypts it twice with CBC, and shows the two ciphertexts. The line `iv = urandom(16)`, shown in bold, picks a new random IV for each new encryption.

```
#!/usr/bin/env python

from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
from binascii import hexlify as hexa
from os import urandom

BLOCKLEN = 16
# the blocks() function splits a data string into space-separated blocks
def blocks(data):
    split = [hexa(data[i:i+BLOCKLEN]) for i in range(0, len(data), BLOCKLEN)]
    return ' '.join(split)
k = urandom(16)
print "k = %s" % hexa(k)
# pick a random IV
iv = urandom(16)
print "iv = %s" % hexa(iv)
# pick an instance of AES in CBC mode
aes = Cipher(algorithms.AES(k), modes.CBC(iv), backend=default_backend()).encryptor()

p = '\x00'*BLOCKLEN*2
c = aes.update(p) + aes.finalize()
print "enc(%s) = %s" % (blocks(p), blocks(c))
# now with a different IV and the same key
iv = urandom(16)
print "iv = %s" % hexa(iv)
aes = Cipher(algorithms.AES(k), modes.CBC(iv), backend=default_backend()).encryptor()
c = aes.update(p) + aes.finalize()
print "enc(%s) = %s" % (blocks(p), blocks(c))
```

Listing 4-5: Using AES in CBC mode

The two plaintexts are the same (two all-zero blocks), but the encrypted blocks should be distinct, as in this example execution:

```
$ ./aes_cbc.py
k = 9cf0d31ad2df24f3cbbefc1e6933c872
iv = 0a75c4283b4539c094fc262aff0d17af
enc(00000000000000000000000000000000 00000000000000000000000000000000) =
370404dcab6e9ecbc3d24ca5573d2920 3b9e5d70e597db225609541f6ae9804a
iv = a6016a6698c3996be13e8739d9e793e2
enc(00000000000000000000000000000000 00000000000000000000000000000000) =
655e1bb3e74ee8cf9ec1540afd8b2204 b59db5ac28de43b25612dfd6f031087a
```

Alas, CBC is often used with a constant IV instead of a random one, which exposes identical plaintexts and plaintexts that start with identical blocks. For example, say the two-block plaintext $P_1 \parallel P_2$ is encrypted in CBC mode to the two-block ciphertext $C_1 \parallel C_2$. If $P_1 \parallel P_2'$ is encrypted with the same IV, where P_2' is some block distinct from P_2, then the ciphertext will

look like $C_1 \| C_2'$, with C_2' different from C_2 but with the same first block C_1. Thus, an attacker can guess that the first block is the same for both plaintexts, even though they only see the ciphertexts.

NOTE *In CBC mode, decryption needs to know the IV used to encrypt, so the IV is sent along with the ciphertext, in the clear.*

With CBC, decryption can be much faster than encryption due to parallelism. While encryption of a new block, P_i, needs to wait for the previous block, C_{i-1}, decryption of a block computes $P_i = \mathbf{D}(K, C_i) \oplus C_{i-1}$, where there's no need for the previous plaintext block, P_{i-1}. This means that all blocks can be decrypted in parallel simultaneously, as long as you also know the previous ciphertext block, which you usually do.

How to Encrypt Any Message in CBC Mode

Let's circle back to the block termination issue and look at how to process a plaintext whose length is not a multiple of the block length. For example, how would we encrypt an 18-byte plaintext with AES-CBC when blocks are 16 bytes? What do we do with the two bytes left? We'll look at two widely used techniques to deal with this problem. The first one, padding, makes the ciphertext a bit longer than the plaintext, while the second one, *ciphertext stealing*, produces a ciphertext of the same length as the plaintext.

Padding a Message

Padding is a technique that allows you to encrypt a message of any length, even one smaller than a single block. Padding for block ciphers is specified in the PKCS#7 standard and in RFC 5652, and is used almost everywhere CBC is used, such as in some HTTPS connections.

Padding is used to expand a message to fill a complete block by adding extra bytes to the plaintext. Here are the rules for padding 16-byte blocks:

- If there's one byte left—for example, if the plaintext is 1 byte, 17 bytes, or 33 bytes long—pad the message with 15 bytes 0f (15 in decimal).
- If there are two bytes left, pad the message with 14 bytes 0e (14 in decimal).
- If there are three bytes left, pad the message with 13 bytes 0d (13 in decimal).

If there are 15 plaintext bytes and a single byte missing to fill a block, padding adds a single 01 byte. If the plaintext is already a multiple of 16, the block length, add 16 bytes 10 (16 in decimal). You get the idea. The trick generalizes to any block length up to 255 bytes (for larger blocks, a byte is too small to encode values greater than 255).

Decryption of a padded message works like this:

1. Decrypt all the blocks as with unpadded CBC.
2. Make sure that the last bytes of the last block conform to the padding rule: that they finish with at least one 01 byte, at least two 02 bytes, or at least three 03 bytes, and so on. If the padding isn't valid—for example, if the last bytes are 01 02 03—the message is rejected. Otherwise, decryption strips the padding bytes and returns the plaintext bytes left.

One downside of padding is that it makes ciphertext longer by at least one byte and at most a block.

Ciphertext Stealing

Ciphertext stealing is another trick used to encrypt a message whose length isn't a multiple of the block size. Ciphertext stealing is more complex and less popular than padding, but it offers at least three benefits:

- Plaintexts can be of any *bit* length, not just bytes. You can, for example, encrypt a message of 131 bits.
- Ciphertexts are exactly the same length as plaintexts.
- Ciphertext stealing is not vulnerable to padding oracle attacks, powerful attacks that sometimes work against CBC with padding (as we'll see in "Padding Oracle Attacks" on page 74).

In CBC mode, ciphertext stealing extends the last incomplete plaintext block with bits from the previous ciphertext block, and then encrypts the resulting block. The last, incomplete ciphertext block is made up of the first blocks from the previous ciphertext block; that is, the bits that have not been appended to the last plaintext block.

In Figure 4-9, we have three blocks, where the last block, P_3, is incomplete (represented by a zero). P_3 is XORed with the last bits from the previous ciphertext block, and the encrypted result is returned as C_2. The last ciphertext block, C_3, then consists of the first bits from the previous ciphertext block. Decryption is simply the inverse of this operation.

There aren't any major problems with ciphertext stealing, but it's inelegant and hard to get right, especially when NIST's standard specifies three different ways to implement it (see Special Publication 800-38A).

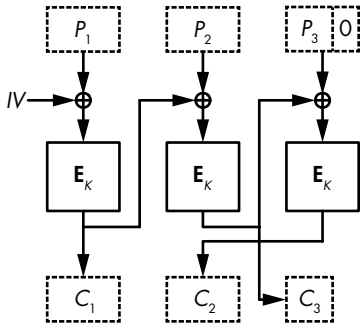


Figure 4-9: Ciphertext stealing for CBC-mode encryption

The Counter (CTR) Mode

To avoid the troubles and retain the benefits of ciphertext stealing, you should use counter mode (CTR). CTR is hardly a block cipher mode: it turns a block cipher into a stream cipher that just takes bits in and spits bits out and doesn't embarrass itself with the notion of blocks. (I'll discuss stream ciphers in detail in Chapter 5.)

In CTR mode (see Figure 4-10), the block cipher algorithm won't transform plaintext data. Instead, it will encrypt blocks composed of a *counter* and a *nonce*. A counter is an integer that is incremented for each block. No two blocks should use the same counter within a message, but different messages can use the same sequence of counter values (1, 2, 3, . . .). A nonce is a number used only once. It is the same for all blocks in a single message, but no two messages should use the same nonce.

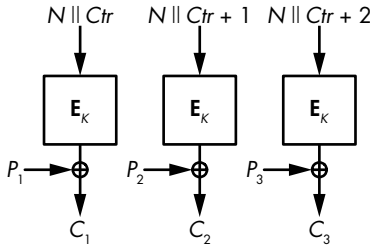


Figure 4-10: The CTR mode

As shown in Figure 4-10, in CTR mode, encryption XORs the plaintext and the stream taken from "encrypting" the nonce, *N*, and counter, *Ctr*. Decryption is the same, so you only need the encryption algorithm for both encryption and decryption. The Python script in Listing 4-6 gives you a hands-on example.

```
#!/usr/bin/env python

from Crypto.Cipher import AES
from Crypto.Util import Counter
from binascii import hexlify as hexa
from os import urandom
from struct import unpack

k = urandom(16)
print "k = %s" % hexa(k)

# pick a starting value for the counter
nonce = unpack('<Q', urandom(8))[0]
# instantiate a counter function
ctr = Counter.new(128, initial_value=nonce)

# pick an instance of AES in CTR mode, using ctr as counter
aes = AES.new(k, AES.MODE_CTR, counter=ctr)

# no need for an entire block with CTR
p = '\x00\x01\x02\x03'

# encrypt p
c = aes.encrypt(p)
print "enc(%s) = %s" % (hexa(p), hexa(c))
```

```
# decrypt using the encrypt function
ctr = Counter.new(128, initial_value=nonce)
aes = AES.new(k, AES.MODE_CTR, counter=ctr)
p = aes.encrypt(c)
print "enc(%s) = %s" % (hexa(c), hexa(p))
```

Listing 4-6: Using AES in CTR mode

The example execution encrypts a 4-byte plaintext and gets a 4-byte ciphertext. It then decrypts that ciphertext using the encryption function:

```
$ ./aes_ctr.py
k = 130a1aa77fa58335272156421cb2a3ea
enc(00010203) = b23d284e
enc(b23d284e) = 00010203
```

As with the initial value in CBC, CTR's nonce is supplied by the encrypter and sent with the ciphertext in the clear. But unlike CBC's initial value, CTR's nonce doesn't need to be random, it simply needs to be unique. A nonce should be unique for the same reason that a one-time pad shouldn't be reused: when calling the pseudorandom stream, S, if you encrypt P_1 to $C_1 = P_1 \oplus S$ and P_2 to $C_2 = P_2 \oplus S$ using the same nonce, then $C_1 \oplus C_2$ reveals $P_1 \oplus P_2$.

A random nonce will do the trick only if it's long enough; for example, if the nonce is n bits, chances are that after $2^{n/2}$ encryptions and as many nonces you'll run into duplicates. Sixty-four bits are therefore insufficient for a random nonce, since you can expect a repetition after approximately 2^{32} nonces, which is an unacceptably low number.

The counter is guaranteed unique if it's incremented for every new plaintext, and if it's long enough; for example, a 64-bit counter.

One particular benefit to CTR is that it can be faster than in any other mode. Not only is it parallelizable, but you can also start encrypting even before knowing the message by picking a nonce and computing the stream that you'll later XOR with the plaintext.

How Things Can Go Wrong

There are two must-know attacks on block ciphers: meet-in-the-middle attacks, a technique discovered in the 1970s but still used in many cryptanalytic attacks (not to be confused with man-in-the-middle attacks), and padding oracle attacks, a class of attacks discovered in 2002 by academic cryptographers, then mostly ignored, and finally rediscovered a decade later along with several vulnerable applications.

Meet-in-the-Middle Attacks

The 3DES block cipher is an upgraded version of the 1970s standard DES that takes a key of $56 \times 3 = 168$ bits (an improvement on DES's 56-bit key). But the security level of 3DES is 112 bits instead of 168 bits, because of the *meet-in-the-middle (MitM)* attack.

As you can see in Figure 4-11, 3DES encrypts a block using the DES encryption and decryption functions: first encryption with a key, K_1, then decryption with a key, K_2, and finally encryption with another key, K_3. If $K_1 = K_2$, the first two calls cancel themselves out and 3DES boils down to a single DES with key K_3. 3DES does encrypt-decrypt-encrypt rather than encrypting thrice to allow systems to emulate DES when necessary using the new 3DES interface.

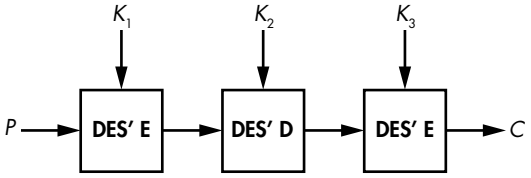


Figure 4-11: The 3DES block cipher construction

Why use triple DES and not just double DES, that is, $\mathbf{E}(K_1, \mathbf{E}(K_2, P))$? It turns out that the MitM attack makes double DES only as secure as single DES. Figure 4-12 shows the MitM attack in action.

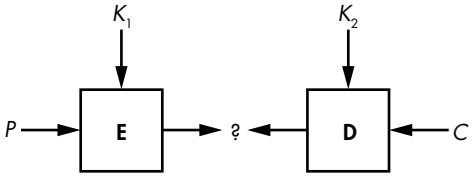


Figure 4-12: The meet-in-the-middle attack

The meet-in-the-middle attack works as follows to attack double DES:

1. Say you have P and $C = \mathbf{E}(K_2, \mathbf{E}(K_1, P))$ with two unknown 56-bit keys, K_1 and K_2. (DES takes 56-bit keys, so double DES takes 112 key bits in total.) You build a key–value table with 2^{56} entries of $\mathbf{E}(K_1, P)$, where $\mathbf{E}$ is the DES encryption function and K_1 is the value stored.
2. For all 2^{56} values of K_2, compute $\mathbf{D}(K_2, C)$ and check whether the resulting value appears in the table as an index (thus as a middle value, represented by a question mark in Figure 4-12).
3. If a middle value is found as an index of the table, you fetch the corresponding K_1 from the table and verify that the (K_1, K_2) found is the right one by using other pairs of P and C. Encrypt P using K_1 and K_2 and then check that the ciphertext obtained is the given C.

This method recovers K_1 and K_2 by performing about 2^{57} instead of 2^{112} operations: step 1 encrypts 2^{56} blocks and then step 2 decrypts at most 2^{56} blocks, for $2^{56} + 2^{56} = 2^{57}$ operations in total. You also need to store 2^{56} elements of 15 bytes each, or about 128 petabytes. That's a lot, but there's a trick that allows you to run the same attack with only negligible memory (as you'll see in Chapter 6).

As you can see, you can apply the MitM attack to 3DES in almost the same way you would to double DES, except that the third stage will go through all 2^{112} values of K_2 and K_3. The whole attack thus succeeds after performing about 2^{112} operations, meaning that 3DES gets only 112-bit security despite having 168 bits of key material.

Padding Oracle Attacks

Let's conclude this chapter with one of the simplest and yet most devastating attacks of the 2000s: the padding oracle attack. Remember that padding fills the plaintext with extra bytes in order to fill a block. A plaintext of 111 bytes, for example, is a sequence of six 16-byte blocks followed by 15 bytes. To form a complete block, padding adds a 01 byte. For a 110-byte plaintext, padding adds two 02 bytes, and so on.

A *padding oracle* is a system that behaves differently depending on whether the padding in a CBC-encrypted ciphertext is valid. You can see it as a black box or an API that returns either a *success* or an *error* value. A padding oracle can be found in a service on a remote host sending error messages when it receives malformed ciphertexts. Given a padding oracle, padding oracle attacks record which inputs have a valid padding and which don't, and exploit this information to decrypt chosen ciphertext values.

Say you want to decrypt ciphertext block C_2. I'll call X the value you're looking for, namely $\mathbf{D}(K, C_2)$, and P_2 the block obtained after decrypting in CBC mode (see Figure 4-13). If you pick a random block C_1 and send the two-block ciphertext $C_1 \mid\mid C_2$ to the oracle, decryption will only succeed if $C_1 \oplus P_2 = X$ ends with valid padding—a single 01 byte, two 02 bytes, or three 03 bytes, and so on.

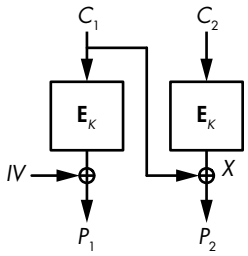


Figure 4-13: Padding oracle attacks recover X by choosing C_1 and checking the validity of padding.

Based on this observation, padding oracle attacks on CBC encryption can decrypt a block C_2 like this (bytes are denoted in array notation: $C_1[0]$ is C_1's first byte, $C_1[1]$ its second byte, and so on up to $C_1[15]$, C_1's last byte):

1. Pick a random block C_1 and vary its last byte until the padding oracle accepts the ciphertext as valid. Usually, in a valid ciphertext, $C_1[15] \oplus X[15] = 01$, so you'll find $X[15]$ after trying around 128 values of $C_1[15]$.
2. Find the value $X[14]$ by setting $C_1[15]$ to $X[15] \oplus 02$ and searching for the $C_1[14]$ that gives correct padding. When the oracle accepts the ciphertext as valid, it means you have found $C_1[14]$ such that $C_1[14] \oplus X[14] = 02$.
3. Repeat steps 1 and 2 for all 16 bytes.

The attack needs on average 128 queries to the oracle for each of the 16 bytes, which is about 2000 queries in total. (Note that each query must use the same initial value.)

NOTE *In practice, implementing a padding oracle attack is a bit more complicated than what I've described, because you have to deal with wrong guesses at step 1. A ciphertext may have valid padding not because* P_2 *ends with a single 01 but because it ends with two 02 bytes or three 03 bytes. But that's easily managed by testing the validity of ciphertexts where more bytes are modified.*

Further Reading

There's a lot to say about block ciphers, be it in how algorithms work or in how they can be attacked. For instance, Feistel networks and SPNs aren't the only ways to build a block cipher. The block ciphers IDEA and FOX use the Lai–Massey construction, and Threefish uses ARX networks, a combination of addition, word rotations, and XORs.

There are also many more modes than just ECB, CBC, and CTR. Some modes are folklore techniques that nobody uses, like CFB and OFB, while others are for specific applications, like XTS for tweakable encryption or GCM for authenticated encryption.

I've discussed Rijndael, the AES winner, but there were 14 other algorithms in the race: CAST-256, CRYPTON, DEAL, DFC, E2, FROG, HPC, LOKI97, Magenta, MARS, RC6, SAFER+, Serpent, and Twofish. I recommend that you look them up to see how they work, how they were designed, how they have been attacked, and how fast they are. It's also worth checking out the NSA's designs (Skipjack, and more recently, SIMON and SPECK) and more recent "lightweight" block ciphers such as KATAN, PRESENT, or PRINCE.

3RD EDITION

PRACTICAL PACKET ANALYSIS

USING *WIRESHARK* TO SOLVE REAL-WORLD NETWORK PROBLEMS

CHRIS SANDERS

no starch press

4

WORKING WITH CAPTURED PACKETS

Now that you've been introduced to Wireshark, you're ready to start capturing and analyzing packets. In this chapter, you'll learn how to work with capture files, packets, and time-display formats. We'll also cover more advanced options for capturing packets and dive into the world of filters.

Working with Capture Files

You'll find that a good portion of your packet analysis will happen after your capture. Usually, you'll perform several captures at various times, save them, and analyze them all at once. Therefore, Wireshark allows you to save your capture files to be analyzed later. You can also merge multiple capture files.

Saving and Exporting Capture Files

To save a packet capture, select **File ▸ Save As**. You should see the Save file as dialog, as shown in Figure 4-1. You'll be asked for a location to save your packet capture and for the file format you wish to use. If you don't specify a file format, Wireshark will use the default *.pcapng* file format.

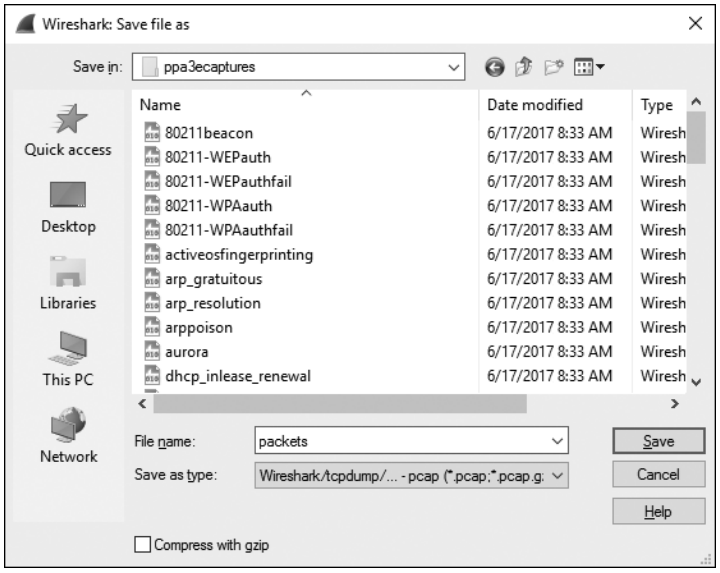


Figure 4-1: The Save file as dialog allows you to save your packet captures.

In many cases, you may only want to save a subset of the packets in your capture. To do so, select **File ▸ Export Specified Packets**. The dialog that appears is shown in Figure 4-2. This is a great way to thin bloated packet-capture files. You can choose to save only packets in a specific number range, marked packets, or packets visible as the result of a display filter (marked packets and filters are discussed later in this chapter).

You can export your Wireshark capture data into several formats for viewing in other media or for importing into other packet analysis tools. Formats include plaintext, PostScript, comma-separated values (CSV), and XML. To export your packet capture in one of these formats, choose **File ▸ Export Packet Dissections** and then select the format for the exported file. You'll see a Save As dialog containing options related to the format you've chosen.

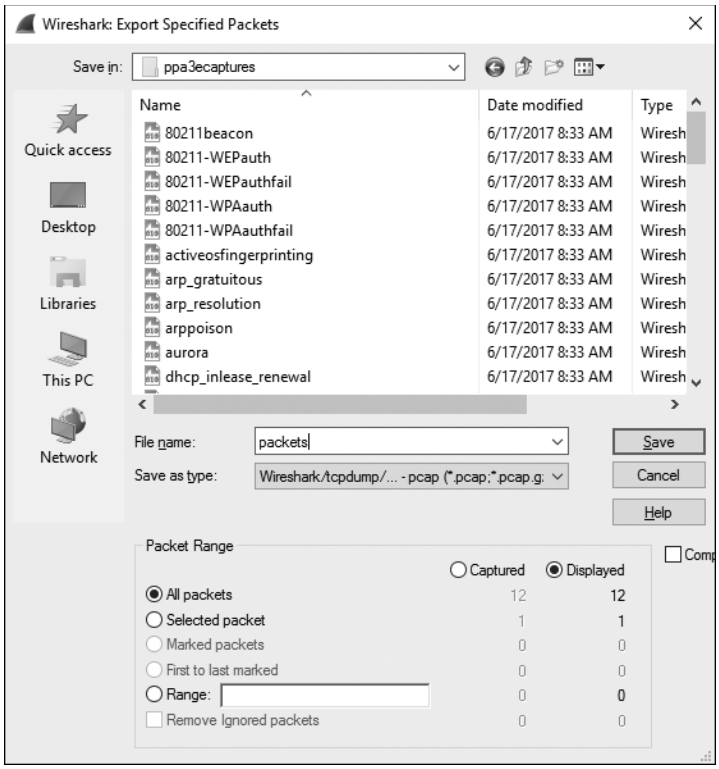


Figure 4-2: The Export Specified Packets dialog allows you to have more granular control over the packets you choose to save.

Merging Capture Files

Certain types of analysis require the ability to merge multiple capture files. This is a common practice when comparing two data streams or combining streams of the same traffic that were captured separately.

To merge capture files, open one of the files you want to merge and choose **File ▸ Merge** to bring up the Merge with capture file dialog, shown in Figure 4-3. Select the new file you wish to merge into the already open file and then select the method to use for merging the files. You can prepend the selected file to the currently open one, append it, or merge the files chronologically based on their timestamps.

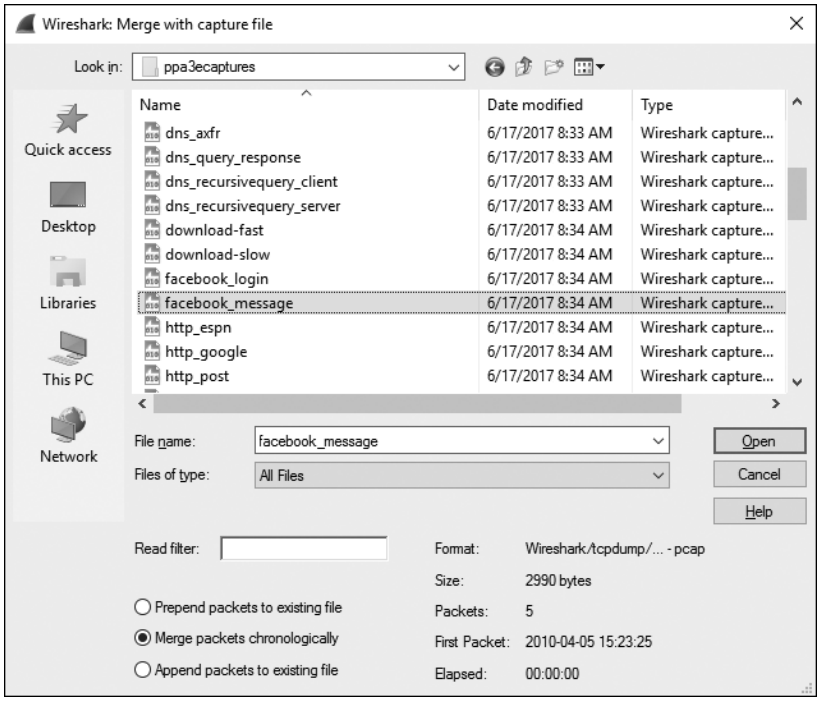


Figure 4-3: The Merge with capture file dialog allows you to merge two capture files.

Working with Packets

You will eventually encounter a situation involving a very large number of packets. As the number of packets grows into the thousands and even millions, you will need to navigate through packets more efficiently. For this purpose, Wireshark allows you to find and mark packets that match certain criteria. You can also print packets for easy reference.

Finding Packets

To find packets that match particular criteria, open the Find Packet bar, shown circled in Figure 4-4, by pressing CTRL-F. This bar should appear between the Filter bar and the Packet List pane.

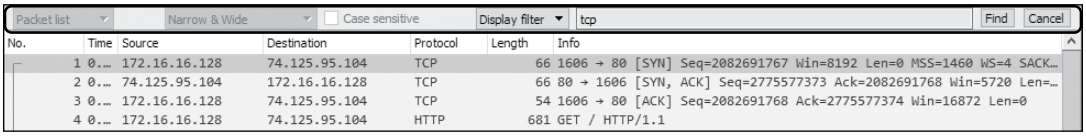


Figure 4-4: Finding packets in Wireshark based on specified criteria—in this case, packets matching the display filter expression `tcp`

This pane offers three options for finding packets:

- The Display filter option allows you to enter an expression-based filter that will find only those packets that satisfy that expression. This option is used in Figure 4-4.
- The Hex value option searches for packets with a hexadecimal value you specify.
- The String option searches for packets with a text string you specify. You can specify the pane the search is performed in or make the search string case sensitive.

Table 4-1 shows examples of these search types.

Table 4-1: Search Types for Finding Packets

Search type	Examples

Display filter	`not ip`
	`ip.addr==192.168.0.1`
	`arp`

Hex value	`00ff`
	`ffff`
	`00ABB1f0`

String	`Workstation1`
	`UserB`
	`domain`

Once you've decided which search type you will use, enter your search criteria in the text box and click **Find** to find the first packet that meets your criterion. To find the next matching packet, click **Find** again or press CTRL-N; find the previous matching packet by pressing CTRL-B.

Marking Packets

After you have found packets that match your criterion, you can mark those of particular interest. For example, marking packets will let you save only these packets. Also, you can find your marked packets quickly by their black background and white text, as shown in Figure 4-5.

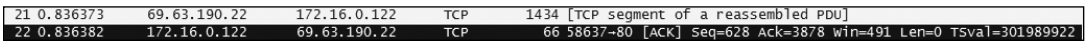


Figure 4-5: A marked packet is highlighted on your screen. In this example, the second packet is marked and appears darker.

To mark a packet, either right-click it in the Packet List pane and choose **Mark Packet** from the pop-up or click a packet in the Packet List pane and press CTRL-M. To unmark a packet, toggle this setting off by pressing CTRL-M again. You can mark as many packets as you wish in a capture. To jump forward and backward between marked packets, press SHIFT-CTRL-N and SHIFT-CTRL-B, respectively.

Printing Packets

Although most analysis will take place on the computer screen, you may need to print captured data. I occasionally print out packets and tape them to my desk so I can quickly reference their contents while doing other analysis. Being able to print packets to a PDF file is also very convenient, especially when preparing reports.

To print captured packets, open the Print dialog by choosing **File ▸ Print** from the main menu, as shown in Figure 4-6.

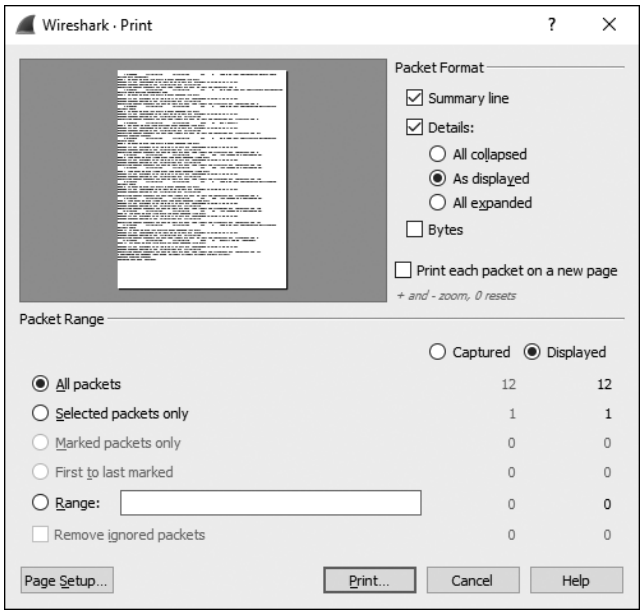


Figure 4-6: The Print dialog allows you to print the packets you specify.

As with the Export Specified Packets dialog, you can print a specific packet range, marked packets only, or packets displayed as the result of a filter. You can also select the level of detail you wish to print for each packet. Once you have selected the options, click **Print**.

Setting Time Display Formats and References

Time is of the essence—especially in packet analysis. Everything that happens on a network is time sensitive, and you will need to examine trends and network latency in capture files frequently. Wireshark supplies several configurable options related to time. In this section, we'll look at time display formats and references.

Time Display Formats

Each packet that Wireshark captures is given a timestamp, which is applied to the packet by the operating system. Wireshark can show the absolute timestamp, which indicates the exact moment when the packet was captured, as well as the time in relation to the last captured packet and the beginning and end of the capture.

Options related to time display are found under the View heading on the main menu. The Time Display Format section, shown in Figure 4-7, lets you configure the presentation format as well as the precision of the time display.

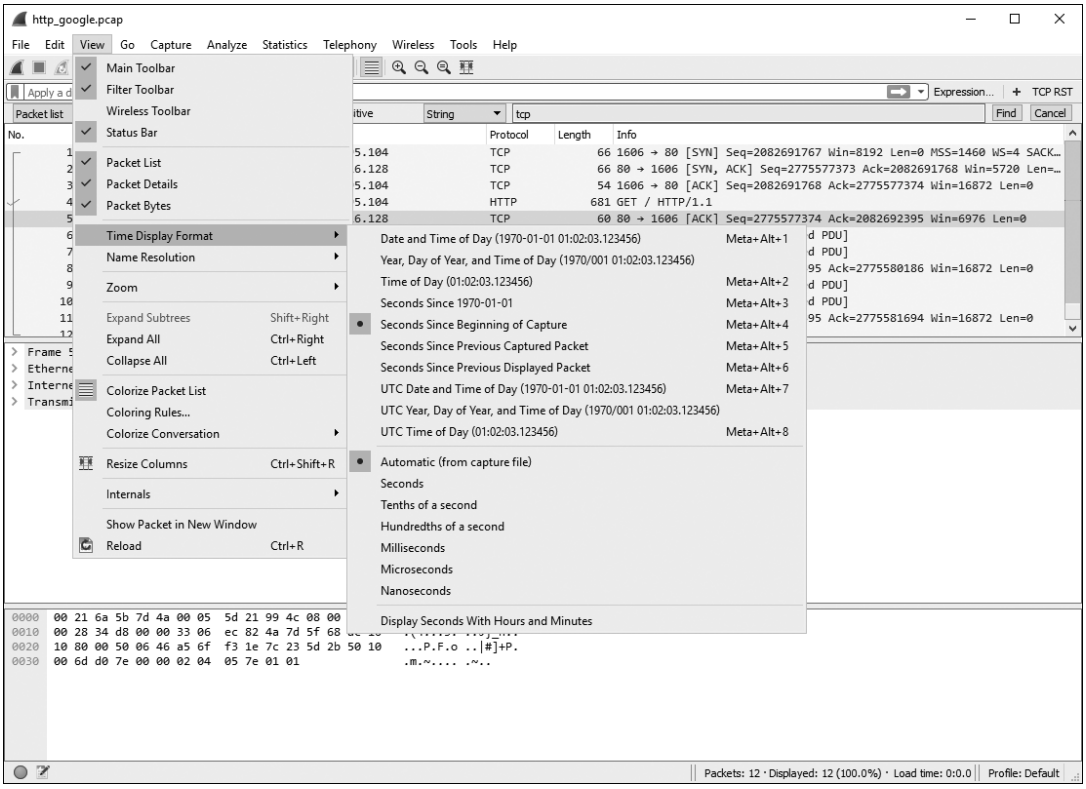


Figure 4-7: Several time display formats are available.

The presentation format options let you choose various settings for time display. These include date and time of day, UTC date and time of day, seconds since epoch, seconds since beginning of capture (the default setting), seconds since previous captured packet, and more.

The precision options allow you to set the time display precision to an automatic setting, which takes the format from the capture file, or to a manual setting, such as seconds, milliseconds, microseconds, and so on. We will be changing these options later in the book, so you should familiarize yourself with them now.

NOTE *When comparing packet data from multiple devices, be sure that the devices are synchronized with the same time source, especially if you are performing forensic analysis or troubleshooting. You can use the Network Time Protocol (NTP) to ensure network devices are synced. When examining packets from devices spanning more than one time zone, consider analyzing packets in UTC instead of local time to avoid confusion when reporting your findings.*

Packet Time Referencing

Packet time referencing allows you to configure a certain packet so that all subsequent time calculations are done in relation to that packet. This feature is particularly handy when you are examining a series of sequential events that are triggered at some point other than the start of the capture file.

To set a time reference to a packet, right-click the reference packet in the Packet List pane and choose **Set/Unset Time Reference**. To toggle this reference off, repeat the same action. You can also toggle a packet as a time reference on and off by selecting the packet you wish to reference in the Packet List pane and pressing CTRL-T.

When you enable a time reference on a packet, the Time column in the Packet List pane will display `*REF*`, as shown in Figure 4-8.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	172.16.16.128	74.125.95.104	TCP	66	1606 → 80 [SYN] Seq=2082691767 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
2	0.030107	74.125.95.104	172.16.16.128	TCP	66	80 → 1606 [SYN, ACK] Seq=2775577373 Ack=2082691768 Win=5720 Len=0 MSS=1406…
3	0.030182	172.16.16.128	74.125.95.104	TCP	54	1606 → 80 [ACK] Seq=2082691768 Ack=2775577374 Win=16872 Len=0
4	*REF*	172.16.16.128	74.125.95.104	HTTP	681	GET / HTTP/1.1
5	0.048778	74.125.95.104	172.16.16.128	TCP	60	80 → 1606 [ACK] Seq=2775577374 Ack=2082692395 Win=6976 Len=0
6	0.070954	74.125.95.104	172.16.16.128	TCP	1460	[TCP segment of a reassembled PDU]
7	0.071217	74.125.95.104	172.16.16.128	TCP	1460	[TCP segment of a reassembled PDU]
8	0.071247	172.16.16.128	74.125.95.104	TCP	54	1606 → 80 [ACK] Seq=2082692395 Ack=2775580186 Win=16872 Len=0

Figure 4-8: Packet 4 with the packet time reference toggle enabled

Setting a packet time reference is useful only when the time display format of a capture is set to display the time in relation to the beginning of the capture. Any other setting will produce no usable results and indeed will generate a set of times that can be very confusing.

Time Shifting

In some cases, you might encounter packets from multiple sources that are not synchronized to the same time source. This is especially common when examining capture files taken from two locations that contain the same stream of data. While most administrators desire a state in which every device on their network is synced, it's not uncommon for there to be a few seconds of time skew between certain types of devices. Wireshark provides the ability to shift the timestamp on packets to alleviate this problem during your analysis.

To shift the timestamp on one or more packets, select **Edit ▸ Time Shift** or press CTRL-SHIFT-T. On the Time Shift screen that opens, you can specify a time range to shift the entire capture file by, or you can specify a time to set individual packets to. In the example shown in Figure 4-9, I've chosen to shift the timestamp of every packet in the capture by adding two minutes and five seconds to each packet.

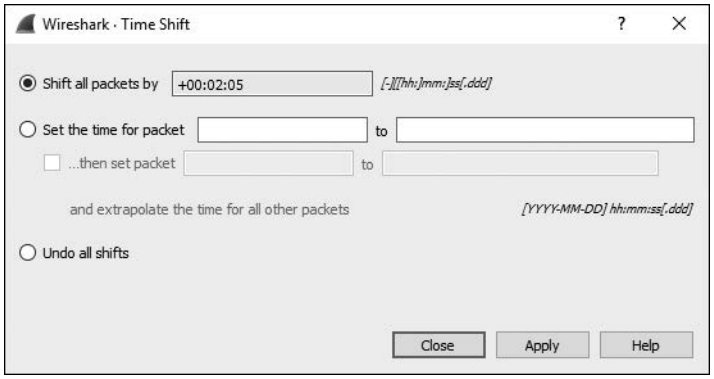


Figure 4-9: The Time Shift dialog

Setting Capture Options

We looked at the Capture Interfaces dialog while walking through a very basic packet capture in the last chapter. Wireshark offers quite a few additional capture options that we didn't address then. To access these options, choose **Capture ▸ Options**.

The Capture Interfaces dialog has a lot of bells and whistles, all designed to give you more flexibility while capturing packets. It's divided into three tabs: Input, Output, and Options. We'll examine each separately.

Input Tab

The main purpose of the Input tab (Figure 4-10) is to display all the interfaces available for capturing packets and some basic information for each interface. This includes the friendly name of the interface provided by the operating system, a traffic graph showing the throughput on the interface, and additional configuration options such as promiscuous mode status and buffer size. At the far right (not pictured), there is also a column for the applied capture filter, which we'll talk about in "Capture Filters" on page 65.

In this section, you can click most of these options and edit them inline. For example, if you want to disable promiscuous mode on an interface, you can click that field and change it from enabled to disabled via the provided drop-down menu.

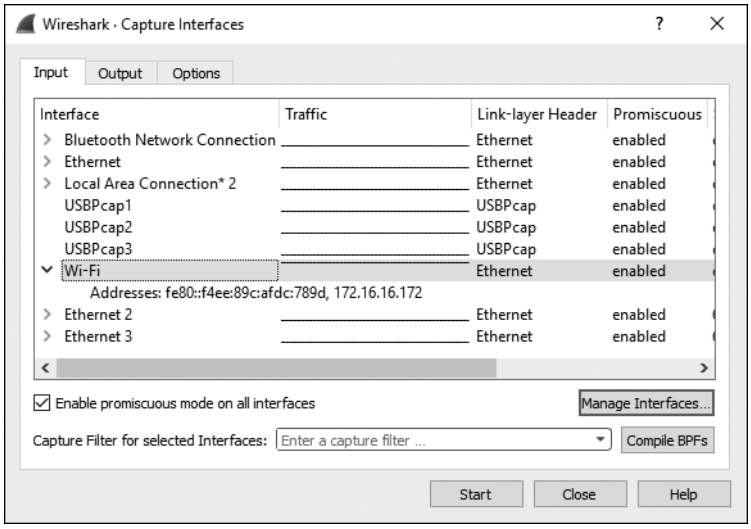


Figure 4-10: The Capture Interfaces Input options tab

Output Tab

The Output tab (Figure 4-11) allows you to automatically store captured packets in a file, rather than capturing them first and then saving the file. Doing so offers you more flexibility in managing how packets are saved. You can choose to save them as a single file or a file set or even use a ring buffer (which we'll cover in a moment) to manage the number of files created. To enable this option, enter a complete file path and name in the File text box. Alternatively, use the Browse... button to select a directory and provide a filename.

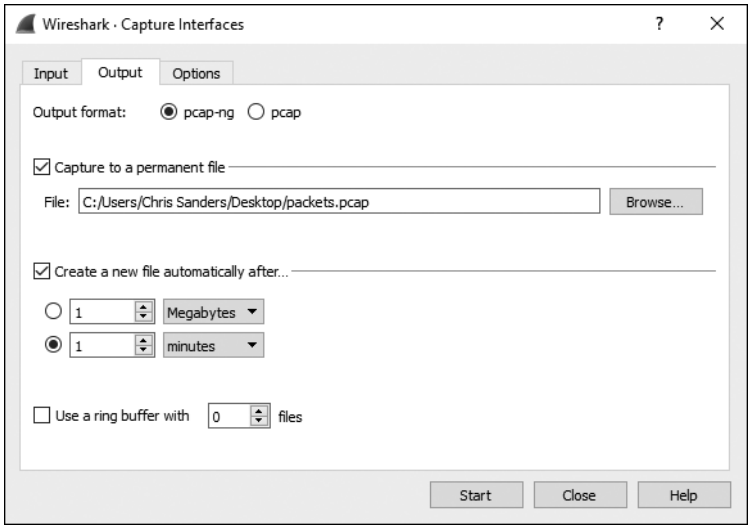
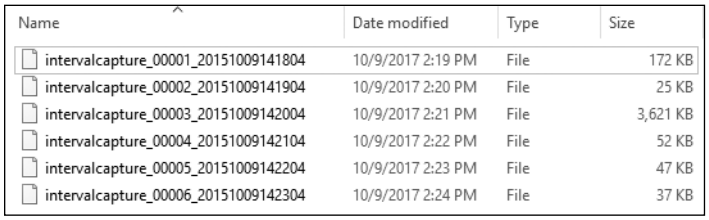


Figure 4-11: The Capture Interfaces Output options tab

When you are capturing a large amount of traffic or performing long-term captures, file sets can prove particularly useful. A *file set* is a grouping of multiple files separated by a particular condition. To save to a file set, check the **Create a new file automatically after...** option.

Wireshark uses various triggers to manage saving to file sets based upon a file size or time condition. To enable one of these triggers, select the radio button next to the size- or time-based option and then specify the value and unit on which to trigger. For instance, you can set a trigger that creates a new file after every 1MB of traffic captured or, as shown in Figure 4-12, after every minute of traffic captured.



Name	Date modified	Type	Size
intervalcapture_00001_20151009141804	10/9/2017 2:19 PM	File	172 KB
intervalcapture_00002_20151009141904	10/9/2017 2:20 PM	File	25 KB
intervalcapture_00003_20151009142004	10/9/2017 2:21 PM	File	3,621 KB
intervalcapture_00004_20151009142104	10/9/2017 2:22 PM	File	52 KB
intervalcapture_00005_20151009142204	10/9/2017 2:23 PM	File	47 KB
intervalcapture_00006_20151009142304	10/9/2017 2:24 PM	File	37 KB

Figure 4-12: A file set created by Wireshark at one-minute intervals

The Use a ring buffer option lets you specify a certain number of files your file set will hold before Wireshark begins to overwrite files. Although the term *ring buffer* has multiple meanings, for our purposes, it is essentially a file set that specifies that once the last file it can hold has been written, when more data must be saved, the first file is overwritten. In other words, it establishes a first in, first out (FIFO) method of writing files. You can check this option and specify the maximum number of files you wish to cycle through. For example, say you choose to use multiple files for your capture with a new file created every hour, and you set your ring buffer to 6. Once the sixth file has been created, the ring buffer will cycle back around and overwrite the first file rather than create a seventh file. This ensures that no more than six files (or in this case, hours) of data will remain on your hard drive, while still allowing new data to be written.

Lastly, the Output tab also lets you specify whether to use the *.pcapng* file format. If you plan to interact with your saved packets using a tool that isn't capable of parsing *.pcapng*, you can select the traditional *.pcap* format.

Options Tab

The Options tab contains a number of other packet-capturing choices, including display, name resolution, and capture termination options, shown in Figure 4-13.

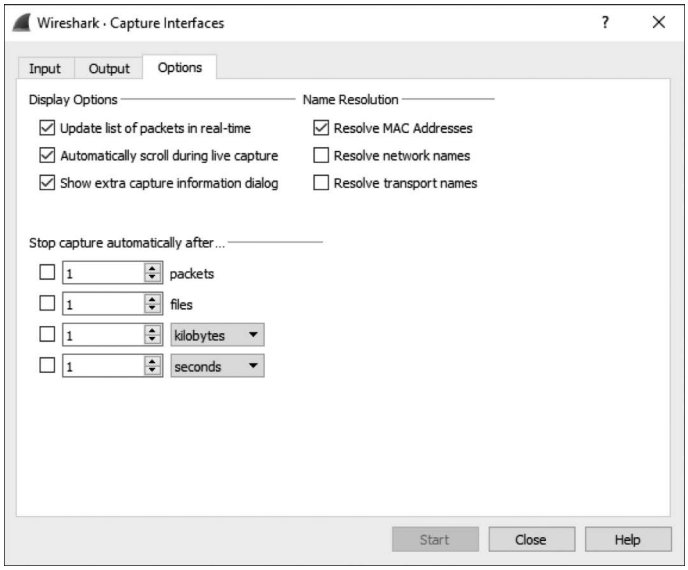


Figure 4-13: The Capture Interfaces Options tab

Display Options

The Display Options section controls how packets are shown as they are being captured. The Update list of packets in real-time option is self-explanatory and can be paired with the Automatically scroll during live capture option. When both of these options are enabled, all captured packets are displayed on the screen, with the most recently captured ones shown instantly.

WARNING *When paired, the Update list of packets in real-time and Automatically scroll during live capture options can be processor intensive, even when you are capturing a modest amount of data. Unless you have a specific need to see the packets in real time, it's best to deselect both options.*

The Show extra capture information dialog option lets you enable or suppress the display of a small window that shows the number and percentage of packets that have been captured, sorted by their protocol. I like to show the capture info dialog since I typically don't allow for the live scrolling of packets during capture.

Name Resolution Settings

The Name Resolution section options allow you to enable automatic MAC (layer 2), network (layer 3), and transport (layer 4) name resolution for your capture. We'll discuss name resolution as a general topic in more depth, including its drawbacks, in Chapter 5.

Stop Capture Settings

The Stop capture automatically after... section lets you stop the running capture when certain conditions are met. As with multiple file sets, you can trigger the capture to stop based on file size and time interval, but you can also trigger on number of packets. These options can be used with the multiple-file options on the Output tab.

Using Filters

Filters allow you to specify which packets you have available for analysis. Simply stated, a filter is an expression that defines criteria for the inclusion or exclusion of packets. If there are packets you don't want to see, you can write a filter that gets rid of them. If there are packets you want to see exclusively, you can write a filter that shows only those packets.

Wireshark offers two main types of filters:

- *Capture filters* are specified when packets are being captured and will capture only those packets that are specified for inclusion/exclusion in the given expression.
- *Display filters* are applied to an existing set of captured packets in order to hide unwanted packets or show desired packets based on the specified expression.

Let's look at capture filters first.

Capture Filters

Capture filters are applied during the packet-capturing process to limit the packets delivered to the analyst from the start. One primary reason for using a capture filter is performance. If you know that you do not need to analyze a particular form of traffic, you can simply filter it out with a capture filter and save the processing power that would typically be used in capturing those packets.

The ability to create custom capture filters comes in handy when you're dealing with large amounts of data. The analysis can be sped up by ensuring that you are looking at only the packet relevant to the issue at hand.

As an example, suppose you are troubleshooting an issue with a service running on port 262, but the server you are analyzing runs several different services on a variety of ports. Finding and analyzing only the traffic on one port would be quite a job in itself. To capture only the traffic on a specific port, you could use a capture filter. To do so, use the Capture Interfaces dialog as follows:

1. Choose the **Capture ▸ Options** button next to the interface on which you want to capture packets. This will open the Capture Interfaces dialog.
2. Find the interface you wish to use and scroll to the Capture Filter option in the far-right column.

3. You can apply the capture filter by clicking in this column to enter an expression. We want our filter to show only traffic inbound and outbound to port 262, so enter **port 262**, as shown in Figure 4-14. (We'll discuss expressions in more detail in the next section.) The color of the cell should turn green, indicating that you've entered a valid expression; it will turn red if the expression is invalid.

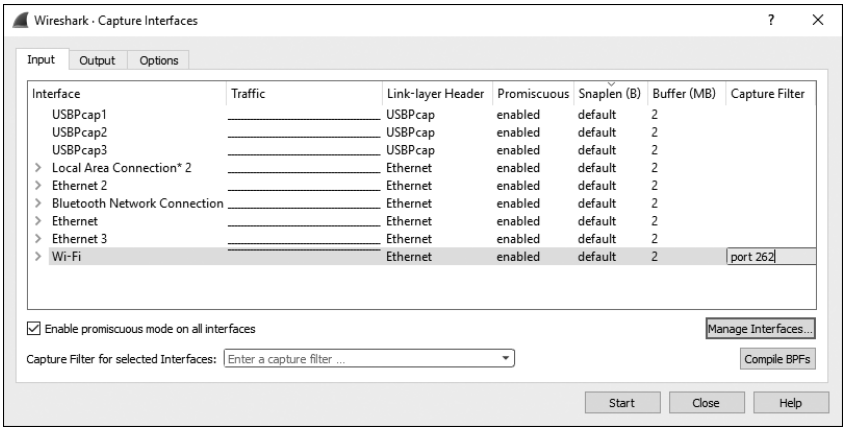


Figure 4-14: Creating a capture filter in the Capture Interfaces dialog

4. Once you have set your filter, click **Start** to begin the capture.

You should now see only port 262 traffic and be able to more efficiently analyze this particular data.

Capture/BPF Syntax

Capture filters are applied by libpcap/WinPcap and use the Berkeley Packet Filter (BPF) syntax. This syntax is common in several packet-sniffing applications, mostly because packet-sniffing applications tend to rely on the libpcap/WinPcap libraries, which allow for the use of BPFs. A knowledge of BPF syntax will be crucial as you dig deeper into networks at the packet level.

A filter created using the BPF syntax is called an *expression*, and each expression consists of one or more *primitives*. Primitives consist of one or more *qualifiers* (as listed in Table 4-2), followed by an ID name or number, as shown in Figure 4-15.

Table 4-2: The BPF Qualifiers

Qualifier	Description	Examples
Type	Identifies what the ID name or number refers to	`host`, `net`, `port`
Dir	Specifies a transfer direction to or from the ID name or number	`src`, `dst`
Proto	Restricts the match to a particular protocol	`ether`, `ip`, `tcp`, `udp`, `http`, `ftp`

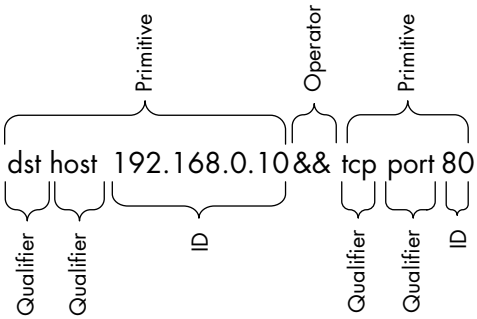


Figure 4-15: A sample capture filter

Given the components of an expression, a qualifier of `dst host` and an ID of 192.168.0.10 would combine to form a primitive. This primitive alone is an expression that would capture traffic only with a destination IP address of 192.168.0.10.

You can use logical operators to combine primitives to create more advanced expressions. Three logical operators are available:

- Concatenation operator AND (`&&`)
- Alternation operator OR (`||`)
- Negation operator NOT (`!`)

For example, the following expression will capture only traffic with a source IP address of 192.168.0.10 and a source or destination port of 80:

```
src host 192.168.0.10 && port 80
```

Hostname and Addressing Filters

Most filters you create will center on a particular network device or grouping of devices. Depending on the circumstances, filtering can be based on a device's MAC address, IPv4 address, IPv6 address, or DNS hostname.

For example, say you're curious about the traffic of a particular host that is interacting with a server on your network. From the server, you can create a filter using the `host` qualifier that captures all traffic associated with that host's IPv4 address:

```
host 172.16.16.149
```

If you are on an IPv6 network, you would filter based on an IPv6 address using the `host` qualifier, as shown here:

```
host 2001:db8:85a3::8a2e:370:7334
```

You can also filter based on a device's hostname with the `host` qualifier, like so:

```
host testserver2
```

Or, if you're concerned that the IP address for a host might change, you can filter based on its MAC address as well by adding the `ether` protocol qualifier:

```
ether host 00-1a-a0-52-e2-a0
```

The transfer direction qualifiers are often used in conjunction with filters, such as the ones in the previous examples, to capture traffic based on whether it's going to or coming from a host. For example, to capture only traffic coming from a particular host, add the `src` qualifier:

```
src host 172.16.16.149
```

To capture only data destined for 172.16.16.149, use the `dst` qualifier:

```
dst host 172.16.16.149
```

When you don't use a type qualifier (`host`, `net`, or `port`) with a primitive, the `host` qualifier is assumed. Therefore, this expression, which excludes that qualifier, is the equivalent of the preceding example:

```
dst 172.16.16.149
```

Port Filters

In addition to filtering on hosts, you can filter based on the ports used in each packet. Port filtering can be used to filter for services and applications that use known service ports. For example, here's a simple filter to capture traffic only to or from port 8080:

```
port 8080
```

To capture all traffic except that on port 8080, this would work:

```
!port 8080
```

The port filters can be combined with transfer direction qualifiers. For example, to capture only traffic going to the web server listening on the standard HTTP port 80, use the `dst` qualifier:

```
dst port 80
```

Protocol Filters

Protocol filters let you filter packets based on certain protocols. They are used to match non–application layer protocols that can't simply be defined by the use of a certain port. Thus, if you want to see only ICMP traffic, you could use this filter:

```
icmp
```

To see everything but IPv6 traffic, this will do the trick:

```
!ip6
```

Protocol Field Filters

One of the real strengths of the BPF syntax is the ability that it gives us to examine every byte of a protocol header in order to create very specific filters based on that data. The advanced filters that we'll discuss in this section will allow you to retrieve a specific number of bytes from a packet beginning at a particular location.

For example, suppose that we want to filter based on the type field of an ICMP header. The type field is located at the very beginning of a packet, which puts it at offset 0. To identify the location to examine within a packet, specify the byte offset in square brackets next to the protocol qualifier—`icmp[0]` in this example. This specification will return a 1-byte integer value that we can compare against. For instance, to get only ICMP packets that represent destination unreachable (type 3) messages, we use the equal to operator in our filter expression:

```
icmp[0] == 3
```

To examine only ICMP packets that represent an echo request (type 8) or echo reply (type 0), use two primitives with the OR operator:

```
icmp[0] == 8 || icmp[0] == 0
```

These filters work great, but they filter based on only 1 byte of information within a packet header. You can also specify the length of the data to be returned in your filter expression by appending the byte length after the offset number within the square brackets, separated by a colon.

For example, say we want to create a filter that captures all ICMP destination-unreachable, host-unreachable packets, identified by type 3, code 1. These are 1-byte fields, located next to each other at offset 0 of the packet header. To do this, we create a filter that checks 2 bytes of data beginning at offset 0 of the packet header, and we compare that data against the hex value 0301 (type 3, code 1), like this:

```
icmp[0:2] == 0x0301
```

A common scenario is to capture only TCP packets with the RST flag set. We will cover TCP extensively in Chapter 8. For now, you just need to know that the flags of a TCP packet are located at offset 13. This is an interesting field because it is collectively 1 byte in size as the flags field, but each particular flag is identified by a single bit within this byte. As I will discuss further in Appendix B, each bit in a byte represents some base 2 number. The bit the flag is stored in is specified by the number the bit represents, so the first bit would represent 1, the second 2, the third 4, and so on. Multiple flags can be set simultaneously in a TCP packet. Therefore, we can't efficiently filter by using a single `tcp[13]` value because several values may represent the RST bit being set.

Instead, we must specify the location within the byte that we wish to examine by appending a single ampersand (`&`), followed by the number that represents where the flag is stored. The RST flag is at the bit representing the number 4 within this byte, and the fact that this bit is set to 4 tells us that the RST flag is set. The filter looks like this:

```
tcp[13] & 4 == 4
```

To see all packets with the PSH flag set, which is identified by the bit location representing the number 8 in the TCP flags at offset 13, our filter would use that location instead:

```
tcp[13] & 8 == 8
```

Sample Capture Filter Expressions

You will often find that the success or failure of your analysis depends on your ability to create filters appropriate for your current situation. Table 4-3 shows a few common capture filters that you might use frequently.

Table 4-3: Commonly Used Capture Filters

Filter	Description
`tcp[13] & 32 == 32`	TCP packets with the URG flag set
`tcp[13] & 16 == 16`	TCP packets with the ACK flag set
`tcp[13] & 8 == 8`	TCP packets with the PSH flag set
`tcp[13] & 4 == 4`	TCP packets with the RST flag set
`tcp[13] & 2 == 2`	TCP packets with the SYN flag set
`tcp[13] & 1 == 1`	TCP packets with the FIN flag set
`tcp[13] == 18`	TCP SYN-ACK packets
`ether host` *00:00:00:00:00:00*	Traffic to or from your MAC address
`!ether host` *00:00:00:00:00:00*	Traffic not to or from your MAC address
`broadcast`	Broadcast traffic only
`icmp`	ICMP traffic

Filter	Description
`icmp[0:2] == 0x0301`	ICMP destination unreachable, host unreachable
`ip`	IPv4 traffic only
`ip6`	IPv6 traffic only
`udp`	UDP traffic only

Display Filters

A display filter is one that, when applied to a capture file, tells Wireshark to display only packets that match that filter. You can enter a display filter in the Filter text box above the Packet List pane.

Display filters are used more often than capture filters because they allow you to filter the packet data you see without actually omitting the rest of the data in the capture file. That way, if you need to revert to the original capture, you can simply clear the filter expression. They are also a lot more powerful thanks to Wireshark's extensive library of packet dissectors.

As an example, in some situations, you might use a display filter to clear irrelevant broadcast traffic from a capture file by filtering out ARP broadcasts from the Packet List pane when those packets don't relate to the current problem being analyzed. However, because those ARP broadcast packets may be useful later, it's better to filter them temporarily than it is to delete them.

To filter out all ARP packets in the capture window, place your cursor in the Filter text box at the top of the Packet List pane and enter **`!arp`** to remove all ARP packets from the list (Figure 4-16). To remove the filter, click the **X** button, and to save the filter for later, click the plus (+) button.

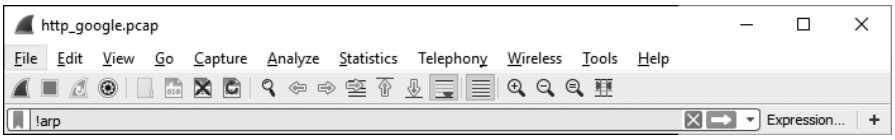


Figure 4-16: Creating a display filter using the Filter text box above the Packet List pane

There are two ways to apply display filters. One is to apply them directly using the appropriate syntax, as we did in this example. Another is to use the Display Filter Expression dialog to build your filter iteratively; this is the easier method when you are first starting to use filters. Let's explore both methods, starting with the easier first.

The Display Filter Expression Dialog

The Display Filter Expression dialog, shown in Figure 4-17, makes it easy for novice Wireshark users to create capture and display filters. To access this dialog, click the **Expression** button on the Filter toolbar.

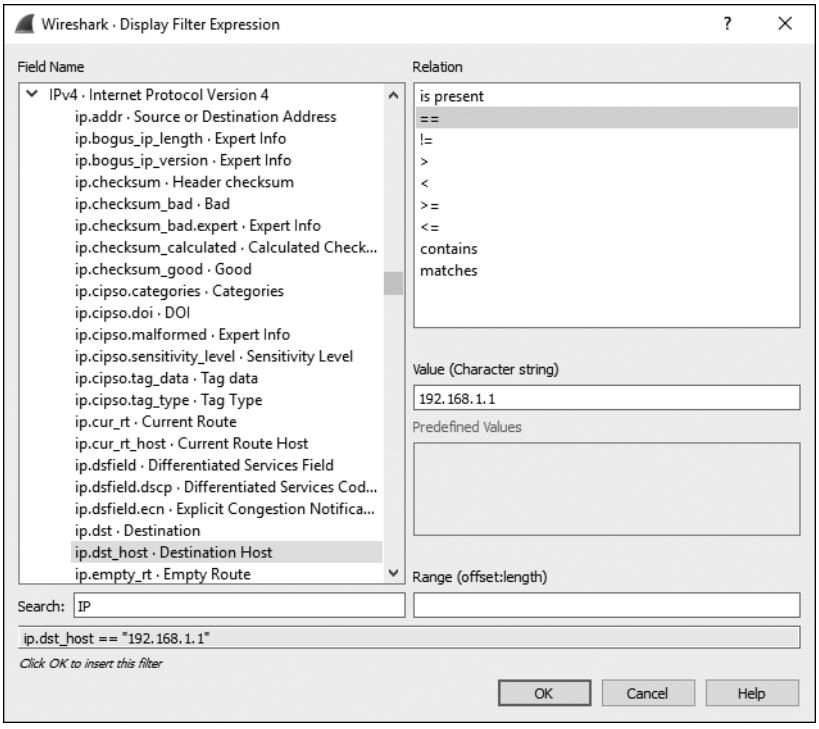


Figure 4-17: The Display Filter Expression dialog allows for the easy creation of filters in Wireshark.

The left side of the dialog lists all possible protocol fields, and these fields specify all possible filter criteria. To create a filter, follow these steps:

1. To view the criteria fields associated with a protocol, expand that protocol by clicking the arrow symbol next to it. Once you find the criterion you want to base your filter on, click to select it.
2. Choose how your selected field will relate to the criterion value you supply. This relation is specified as equal to, greater than, less than, and so on.
3. Create your filter expression by specifying a criterion value that will relate to your selected field. You can define this value or select it from predefined ones programmed into Wireshark.
4. Your complete filter will be displayed at the bottom of the screen. When you've finished, click **OK** to insert it into the filter bar.

The Display Filter Expression dialog is great for novice users, but once you get the hang of things, you'll find that manually entering filter expressions greatly increases your efficiency. The display filter expression syntax structure is simple, yet extremely powerful.

The Filter Expression Syntax Structure

When you begin using Wireshark more, you will want to start using the display filter syntax directly in the main window to save time. Fortunately, the syntax used for display filters follows a standard scheme and is easy to navigate. In most cases, this scheme is protocol-centric and follows the format *protocol.feature.subfeature*, as you saw when looking at the Display Filter Expression dialog. Now we will look at a few examples.

You will most often use a capture or display filter to see packets based on a specific protocol alone. For example, say you are troubleshooting a TCP problem and you want to see only TCP traffic in a capture file. If so, a simple `tcp` filter will do the job.

Now let's look at things from the other side of the fence. Imagine that in the course of troubleshooting your TCP problem, you have used the ping utility quite a bit, thereby generating a lot of ICMP traffic. You could remove this ICMP traffic from your capture file with the filter `expression !icmp`.

Comparison operators allow you to compare values. For example, when troubleshooting TCP/IP networks, you will often need to view all packets that reference a particular IP address. The equal to comparison operator (`==`) will allow you to create a filter showing all packets with an IP address of 192.168.0.1:

```
ip.addr==192.168.0.1
```

Now suppose that you need to view only packets that are less than 128 bytes. You can use the less than or equal to operator (`<=`) to accomplish this goal:

```
frame.len<=128
```

Table 4-4 shows Wireshark's comparison operators.

Table 4-4: Wireshark Filter Expression Comparison Operators

Operator	Description
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Logical operators allow you to combine multiple filter expressions into one statement, dramatically increasing the effectiveness of your filters.

For example, say that you're interested in displaying only packets to two IP addresses. You can use the `or` operator to create one expression that will display packets containing either IP address, like this:

```
ip.addr==192.168.0.1 or ip.addr==192.168.0.2
```

Table 4-5 lists Wireshark's logical operators.

Table 4-5: Wireshark Filter Expression Logical Operators

Operator	Description
`and`	Both conditions must be true.
`or`	Either one of the conditions must be true.
`xor`	One and only one condition must be true.
`not`	Neither one of the conditions is true.

Sample Display Filter Expressions

Although the concepts related to creating filter expressions are fairly simple, you will need to use several specific keywords and operators when creating new filters for various problems. Table 4-6 shows some of the display filters that I use most often. For a complete list, see the Wireshark display filter reference at *http://www.wireshark.org/docs/dfref/*.

Table 4-6: Commonly Used Display Filters

Filter	Description
`!tcp.port==3389`	Filter out RDP traffic
`tcp.flags.syn==1`	TCP packets with the SYN flag set
`tcp.flags.reset==1`	TCP packets with the RST flag set
`!arp`	Clear ARP traffic
`http`	All HTTP traffic
`tcp.port==23 || tcp.port==21`	Telnet or FTP traffic
`smtp || pop || imap`	Email traffic (SMTP, POP, or IMAP)

Saving Filters

Once you begin creating a lot of capture and display filters, you will find that you use certain ones frequently. Fortunately, you don't need to type these in each time you want to use them, because Wireshark lets you save your filters for later use. To save a custom capture filter, follow these steps:

1. Select **Capture ▸ Capture Filters** to open the Capture Filter dialog.
2. Create a new filter by clicking the plus (+) button on the lower left side of the dialog.

3. Enter a name for your filter in the Filter Name box.
4. Enter the actual filter expression in the Filter String box.
5. Click the **OK** button to save your filter expression in the list.

To save a custom display filter, follow these steps:

1. Type your filter into the Filter bar above the Packet List pane in the main window and click the **ribbon** button on the left side of the bar.
2. Click the **Save this Filter** option, and a list of saved display filters will be presented in a separate dialog. There you can provide a name for your filter before clicking **OK** to save it (Figure 4-18).

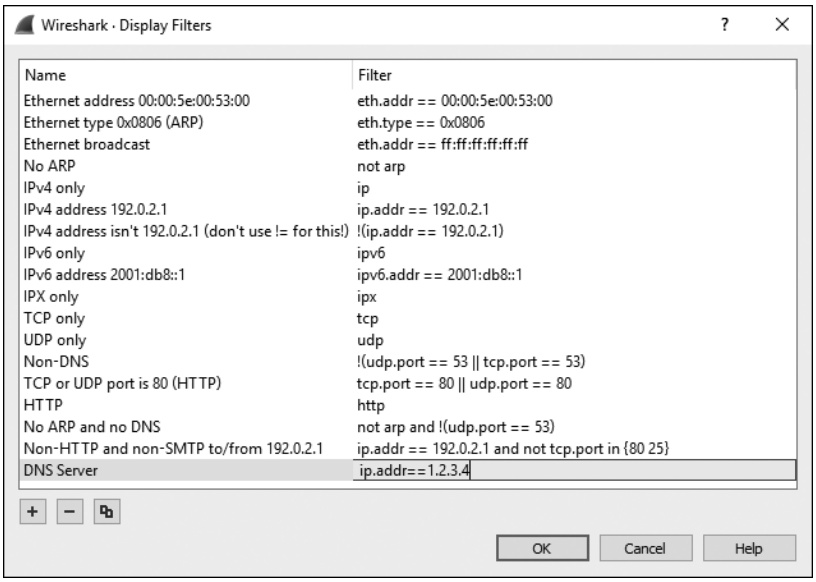


Figure 4-18: You can save display filters directly from the main toolbar.

Adding Display Filters to a Toolbar

If you have filters that you find yourself flipping on and off frequently, one of the easiest ways to interact with them is to add filter toggles to the Filter bar just above the Packet List pane. To do this, complete the following steps:

1. Type your filter into the Filter bar above the Packet List pane in the main window and click the plus (+) button on the right side of the bar.
2. A new bar will display below the Filter bar where you can provide a name for your filter in the Label field (Figure 4-19). This is the label that will be used to represent the filter on the toolbar. Once you've input something in this field, click **OK** to create a shortcut to this expression in the Filter toolbar.

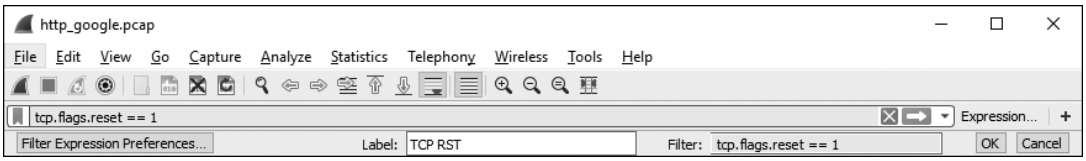


Figure 4-19: Adding a filter expression shortcut to the Filter toolbar

As you can see in Figure 4-20, we've created a shortcut to a filter that will quickly show any TCP packets with the RST flag enabled. Additions to the filtering toolbar are saved to your configuration profile (as discussed in Chapter 3), making them a powerful way to enhance your ability to identify problems in packet captures in various scenarios.

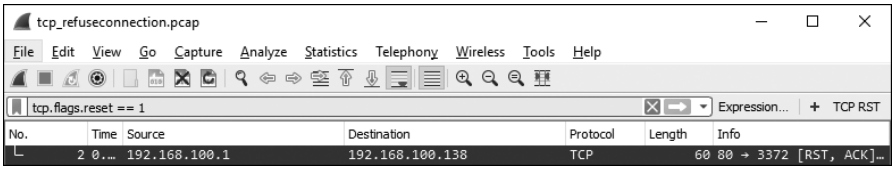


Figure 4-20: Filtering using a toolbar shortcut

Wireshark includes several built-in filters that are great examples of what a filter should look like. You'll want to use them (together with the Wireshark help pages) when creating your own filters. We'll use filters in examples throughout this book.

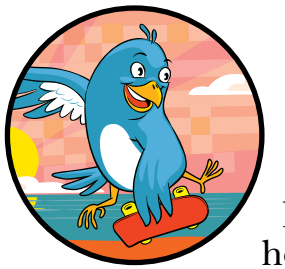
CODING IPHONE APPS FOR KIDS

A PLAYFUL INTRODUCTION TO SWIFT

GLORIA WINQUIST AND MATT MCCARTHY

2

LEARNING TO CODE IN A PLAYGROUND



A "Hello, world!" app is no small accomplishment, but now it's time to really learn how to write some code. Xcode provides a special type of document called a *playground*, which is a great place to learn how to program. In a playground, you can write and run code immediately to see the results, without going through the trouble of writing a whole app, as we did in Chapter 1.

Let's open a playground. Open Xcode and select **Get started with a playground**, as shown in the Welcome to Xcode dialog in Figure 2-1. If this window doesn't automatically open, select **Welcome to Xcode** from the Window option in the menu or press ⌘-SHIFT-1.

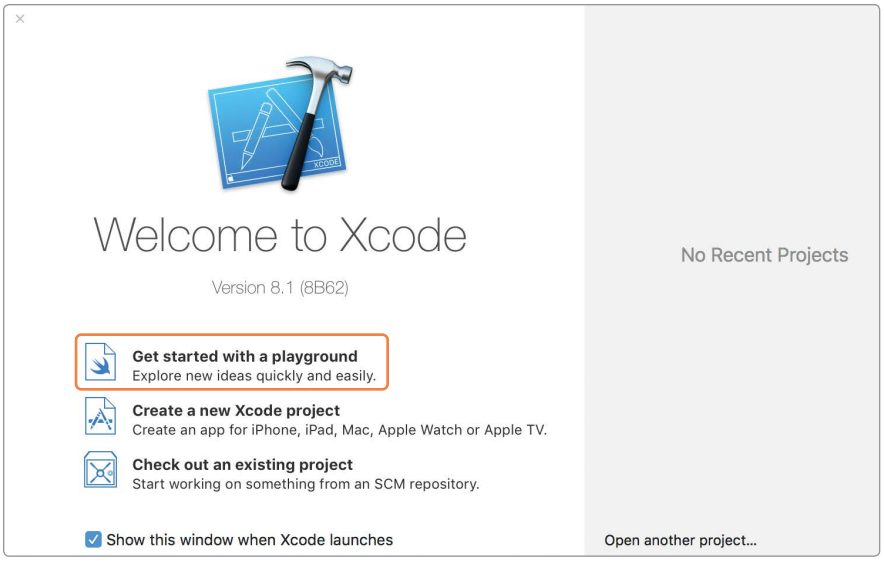


Figure 2-1: Getting started with a playground

You'll be asked to name your playground (Figure 2-2). In this example, we'll keep the default name *MyPlayground*. Make sure that you choose iOS as the platform to run the playground.

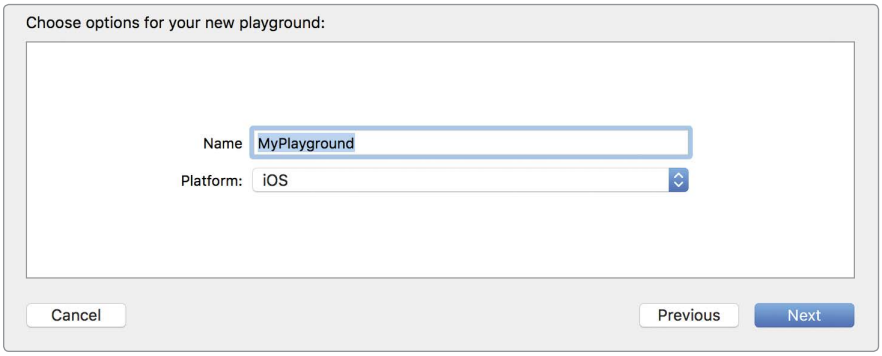
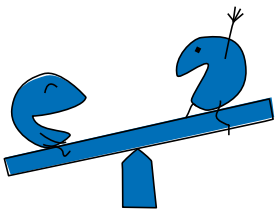


Figure 2-2: Naming the playground and selecting the platform

When the playground first opens, you'll see two panels in the window, just like in Figure 2-3. On the left is the playground editor, where you'll write your code. On the right is the results sidebar, which displays the results of your code.

The line `var str = "Hello, playground"` in Figure 2-3 creates a variable named `str`. A *variable* is like a container; you can use it to hold almost anything—a simple number, a string of letters, or a complex object (we'll explain what that is later). Let's take a closer look at how variables work.



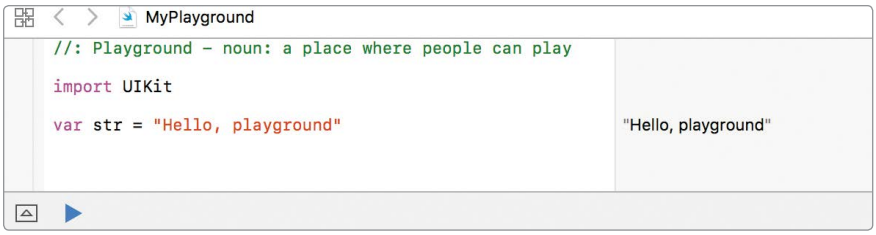


Figure 2-3: Playground editor and results sidebar

CONSTANTS AND VARIABLES

Here's the line of code from Figure 2-3 again:

```
var str = "Hello, playground"                    "Hello, playground"
```

It does two things. First, it creates a variable named `str`. This is called a *declaration* because we are declaring that we would like to create a variable. To create a variable, you type the word `var` and then type a name for your variable—this case, `str`. There are some rules when naming variables, which we'll go over later, so for now stick with this example.

Second, this line of code gives a value of `"Hello, playground"` to `str` using the `=` operator. This is called an *assignment* because we are assigning a value to our newly created variable. Remember, you can think of a variable as a container that holds something. So now we have a container named `str` that holds `"Hello, playground"`.

You can read this line of code as "the variable `str` equals `Hello, playground`." As you can see, Swift is often very readable; this line of code practically tells you in English what it's doing.

Variables are handy because if you want to print the words "Hello, playground" all you have to do is use the command `print` on `str`, like in the following code:

```
print(str)                                       "Hello, playground\n"
```

This prints `"Hello, playground\n"` in the results sidebar. The `\n` is added automatically to the end of whatever you print. It is known as the *newline* character and tells the computer to go to a new line.

To see the results of your program as it would actually run, bring up the debug area, which will appear below the two panels, as shown in Figure 2-4. To do this, go to **View ▸ Debug Area ▸ Show Debug Area** in the Xcode menu or press ⌘-SHIFT-Y. When `str` is printed in the console of the debug area, you can see that the quotes around `Hello, playground` and the newline character don't appear. This is what `str` would really look like if you were to officially run this program!

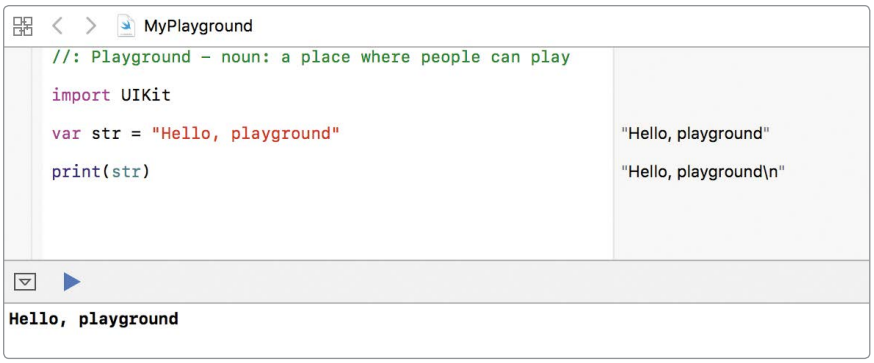


Figure 2-4: Viewing the real output of your program in the debug area

Variables can change (or *vary*!) in your programs, so you can change the value of a variable to hold something else. Let's try that now. Add the following lines to your playground.

```
❶ str = "Hello, world"
print(str)
```

```
"Hello, world"
"Hello, world\n"
```

To change the value of a variable, type its name and use the `=` operator to set it to a new value. We do this at ❶ to change the value of `str` to `"Hello, world"`. The computer throws away whatever `str` used to hold, and says, "Okay, boss, `str` is now `Hello, world`" (that is, it would say that if it could talk!).

Notice that when we change the value of `str`, we don't write `var` again. The computer remembers that we declared `str` in a previous line of code and knows that `str` already exists. So we don't need to create `str` again. We just want to put something different in it.

You can also declare *constants*. Like variables, constants hold values. The big difference between a constant and a variable is that a constant can never change its value. Variables can vary, and constants are, well, constant! Declaring a constant is similar to declaring a variable, but we use the word `let` instead of `var`:

```
let myName = "Gloria"
```

```
"Gloria"
```

Here we create a constant called `myName` and assign it the value of `"Gloria"`.

Once you create a constant and give it a value, it will have that value until the end of time. Think of a constant as a big rock into which you've carved your value. If you try to give `myName` another value, like `"Matt"`, you'll get an error like the one in Figure 2-5.

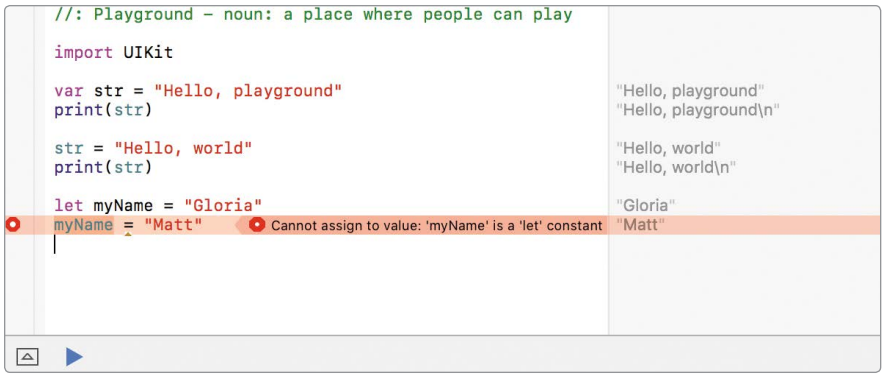
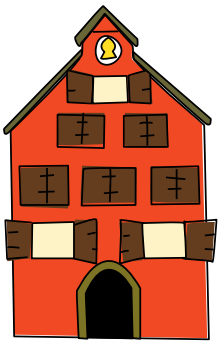


Figure 2-5: Trying to change the value of a constant won't work.

NOTE *In the playground, an error will appear as a red circle with a tiny white circle inside it. Clicking the error mark will show the error message and tell you what's wrong. If you have your debug area showing, you should also see information describing what happened and sometimes even how to fix it.*

WHEN TO USE CONSTANTS VS. VARIABLES

Now you've successfully created a variable and a constant—good job! But when should you use one over the other? In Swift, it's best practice to use constants instead of variables unless you expect the value will change. Constants help make code "safer." If you know the value of something is never going to change, why not etch it into stone and avoid any possible confusion later?



For example, say you want to keep track of the total number of windows in your classroom and the number of windows that are open today. The number of windows in your classroom isn't going to change, so you should use a constant to store this value. The number of windows that are open in your classroom will change depending on the weather and time of day, however, so you should use a variable to store this value.

```
let numberOfWindows = 8
var numberOfWindowsOpen = 3
```

```
8
3
```

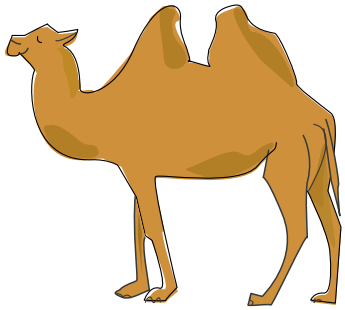
We make `numberOfWindows` a constant and set it to `8` because the total number of windows will always be 8. We make `numberOfWindowsOpen` a variable and set it to `3` because we'll want to change that value when we open or close windows.

Remember: use `var` for variables and `let` for constants!

NAMING CONSTANTS AND VARIABLES

You can name a variable or constant almost anything you want, with a few exceptions. You can't name them something that is already a word in Swift. For example, you can't name a variable `var`. Writing `var var` would just be confusing, to you and the computer. You'll get an error if you try to name a variable or constant using one of Swift's reserved words. You also can't have two variables or constants with the same name in the same block of code.

In addition to these rules, there are some other good programming guidelines to follow when naming things in Swift. Your names should always start with a lowercase letter. It's also a good idea to have *very* descriptive names (they can be as long as you want). When you use a descriptive name, it's a lot easier to figure out what that variable or constant is supposed to be. If you were looking at someone else's code, which variable name would you find easier to understand: `numKids` or `numberOfKidsInMyClass`? The first one is vague, but the second one is descriptive. It's common to see variables and constants that are a bunch of words strung together, like `numberOfKidsInMyClass`. This capitalization style, where the first letter of each word is capitalized when multiple words are joined together to make a variable name, is called *camel case*. That's because the pattern of lowercase and uppercase letters looks like the humps on a camel's back.



DATA TYPES

In Swift, you can choose what kind of data—the *data type*—you want a variable or constant to hold. Remember how we said you can think of a variable as a container that holds something? Well, the data type is like the container type. The computer needs to know what kind of things we'll put in each container. In Swift programming, once you tell the computer you want a variable or constant to hold a certain data type, it won't let you put anything but that data type in that variable or constant. If you have a basket designed to hold potatoes, it'd be a bad idea to fill that basket with water—unless you like water leaking all over your shoes!

DECLARING DATA TYPES

When you create a variable or a constant, you can tell the computer what type of data it will hold. In our example about classroom windows, we know this variable will always be an *integer* (that is, a whole number—you can't really have half a window), so we could specify an integer data type, like this:

```
var numberOfWindowsOpen: Int = 3
```

```
3
```

The colon means "is of type." In plain English, this line of code says, "the variable `numberOfWindowsOpen`, which is an integer, is equal to 3." So this line of code creates a variable, gives it a name, tells the computer its data type, and assigns it a value. Phew! One line of code did all that? Did we mention that Swift is a very *concise* language? Some languages might require several lines of code to do this same thing. Swift is designed so that you can do a bunch of things with just one line of code!

You only have to declare the data type once. When we tell the computer that a variable will hold integers, we don't have to tell it again. In fact, if we do, Xcode will give us an error. Once the data type is declared, a variable or constant will hold that type of data forever. Once an integer, always an integer!

There's one more thing you need to know about data types: a variable or constant can't hold something that isn't its data type. For example, if you try to put a decimal number into `numberOfWindowsOpen`, you'll get an error, as shown in Figure 2-6.

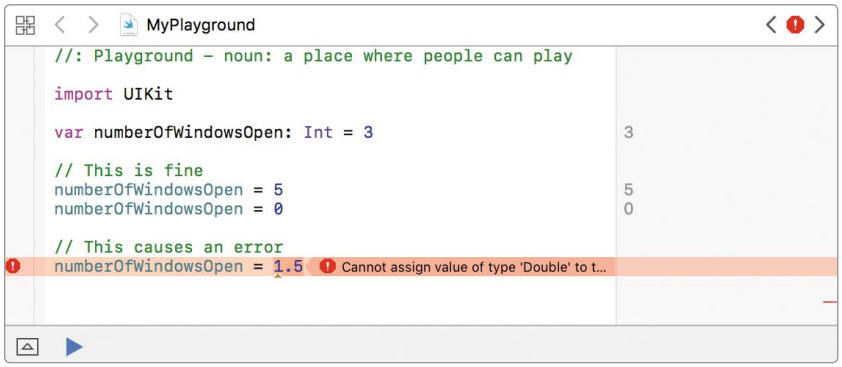


Figure 2-6: You can't put a decimal number into a variable that is supposed to hold an integer.

Setting `numberOfWindowsOpen = 5` and `numberOfWindowsOpen = 0` is valid and works. But you can't `set numberOfWindowsOpen = 1.5`.

COMMON DATA TYPES

As you just learned, a data type lets the computer know what *kind* of data it is working with and how to store it in its memory. But what are the data types? Some common ones include `Int`, `Double`, `Float`, `Bool`, and `String`.

Let's dig in and see what each one of these actually is!

Int (Integers)

We already talked a little bit about integers, but let's go over them in more detail. An integer, called an `Int` in Swift, is a whole number that has no decimal or fractional part. You can think of them as counting numbers. Integers are *signed*, meaning that they can be negative or positive (or zero).

Double and Float (Decimal Numbers)

Decimal numbers are numbers that have digits after the decimal point, like 3.14. (An integer like 3 would be written as 3.0 if you wanted it to be a decimal number.) There are two data types that can store decimal numbers: a `Double` and a `Float` (short for *floating-point number*). The `Double` data type is more common in Swift because it can hold bigger numbers, so we'll focus on those.

When you assign a `Double`, you must always have a digit to the left of the decimal place or you will get an error. For example, suppose bananas cost 19 cents each:

```
❶ var bananaPrice: Double = .19 // ERROR
❷ var bananaPrice: Double = 0.19 // CORRECT
```

```
0.19
```

The code at ❶ will result in an error because it doesn't have a digit to the left of the decimal point. The code at ❷ works fine because it has a leading zero. (The phrases `// ERROR` and `// CORRECT` are *comments*, which are notes in a program that are ignored by the computer. See "A Few Quick Comments About Comments" on page 32.)

Bool (Booleans, or True/False)

A *Boolean value* can only be one of two things: true or false. In Swift, the Boolean data type is called a `Bool`.

```
let swiftIsFun = true
var iAmSleeping = false
```

```
true
false
```

Booleans are often used in `if-else` statements to tell the computer which path a program should take. (We'll cover Booleans and `if-else` statements in more detail in Chapter 3.)

String

The `String` data type is used to store words and phrases. A *string* is a collection of characters enclosed in quotation marks. For example, `"Hello, playground"` is a string. Strings can be made up of all sorts of

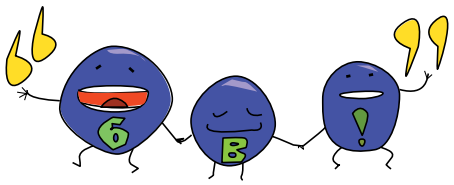
characters: letters, numbers, symbols, and more. The quotation marks are important because they tell the computer that everything in between the quotes is part of a string that you're creating.

You can use strings to build sentences by adding strings together in a process called string *concatenation*. Let's see how it works!

```
let morningGreeting = "Good Morning"
let friend = "Jude"
let specialGreeting = morningGreeting + " " + friend
```

```
"Good Morning"
"Jude"
"Good Morning Jude"
```

By adding strings together with the plus sign (+), this code creates a variable called `specialGreeting` with the string `"Good Morning Jude"` as its value. Note that we need to add a string containing a space character (`" "`) between `morningGreeting` and `friend` here or else `specialGreeting` would be `"Good MorningJude"`.



TYPE INFERENCE

You may have noticed that sometimes when we declare a variable, we include the data type:

```
var numberOfWindowsOpen: Int = 3
```

```
3
```

And sometimes we do not include the data type:

```
var numberOfWindowsOpen = 3
```

```
3
```

What gives? The computer is actually smart enough to figure out the data type, most of the time. This is called *type inference*—because the computer will *infer*, or guess, the type of data we are using based on clues that we give it. When you create a variable and give it an initial value, that value is a big clue for the computer. Here are some examples:

- If you assign a number with no decimal value (like 3), the computer will assume it's an `Int`.
- If you assign a number with a decimal value (like 3.14), the computer will assume it's a `Double`.
- If you assign the word *true* or *false* (with no quotes around it), the computer will assume it's a `Bool`.
- If you assign one or more characters with quotes around them, the computer will assume it's a `String`.

When the type is inferred, the variable or constant is set to that data type just as if you had declared the data type yourself. This is done purely for convenience. You can include the data type every time you declare a new constant or variable, and that's perfectly fine. But why not let the computer figure it out and save yourself the time and extra typing?

TRANSFORMING DATA TYPES WITH CASTING

Casting is a way to temporarily transform the data type of a variable or constant. You can think of this as casting a spell on a variable—you make its value behave like a different data type, but just for a short while. To do this, you write a new data type followed by parentheses that hold the variable you are casting. Note that this *doesn't actually change the data type.* It just gives you a temporary value for that one line of code. Here are a few examples of casting between `Int` and `Double`. Take a look at the results of your code in the results sidebar.

```
let months = 12
print(months)
❶ let doubleMonths = Double(months)
print(doubleMonths)
```

```
12
"12\n"
12
"12.0\n"
```

At ❶, we cast our `Int` variable `months` to a `Double` and store it in a new variable called `doubleMonths`. This adds a decimal place, and the result of this casting is `12.0`.

You can also cast a `Double` to an `Int`:

```
let days = 365.25
❶ Int(days)
```

```
365.25
365
```

At ❶, we cast our `Double`, `days`, to an `Int`. You can see that the decimal place and all the digits following it were removed: our number became `365`. This is because an `Int` is not capable of holding a decimal number—it can contain only whole numbers, so anything after the decimal point is chopped off.

Again, casting doesn't actually change a data type. In our example, even after casting, `days` is *still* a `Double`. We can verify this by printing `days`:

```
print(days)
```

```
"365.25\n"
```

The results sidebar shows that `days` is still equal to `365.25`.

In the next section, we'll cover some examples of where and when you would use casting. So if it's not clear right now why you would cast a variable, just hold on a bit longer!

OPERATORS

There are a number of arithmetic operators in Swift that you can use to do math. You have already seen the basic assignment operator, `=`. You're probably also familiar with addition (`+`), subtraction (`-`), multiplication (`*`), and division (`/`).

You can use these operators to perform math on the `Int`, `Float`, and `Double` data types. The numbers being operated on are called *operands*. Experiment with these math operators in your playground by entering code like the following:

```
6.2 + 1.4                    7.6
3 * 5                        15
16 - 2                       14
9 / 3                        3
```

If you enter this code in your playground, you'll see the results of each math expression in the sidebar. Writing math expressions in code is not that different from writing them normally. For example, 16 minus 2 is written as `16 - 2`.

You can even save the result of a math expression in a variable or constant so you can use it somewhere else in your code. To see how this works, enter these lines in your playground:

```
var sum = 6.2 + 1.4          7.6
❶ print(sum)                 "7.6\n"
let threeTimesFive = 3 * 5   15
```

When you print `sum` ❶, you'll see `7.6` in the sidebar.

So far, we've used only numbers in our math expressions, but math operators also work on variables and constants.

Add the following code to your playground:

```
let three = 3                3
let five = 5                 5
let half = 0.5               0.5
let quarter = 0.25           0.25
var luckyNumber = 7          7

three * luckyNumber          21
five + three                 8
half + quarter               0.75
```

As you can see, you can use math operators on variables and constants like you did on numbers.

SPACES MATTER

In Swift, the spaces around an operator are important. You can either write a blank space on both sides of the math operator or leave out the spaces altogether. But you cannot just put a space on one side of the operator and not the other. That will cause an error. Take a look at Figure 2-7.

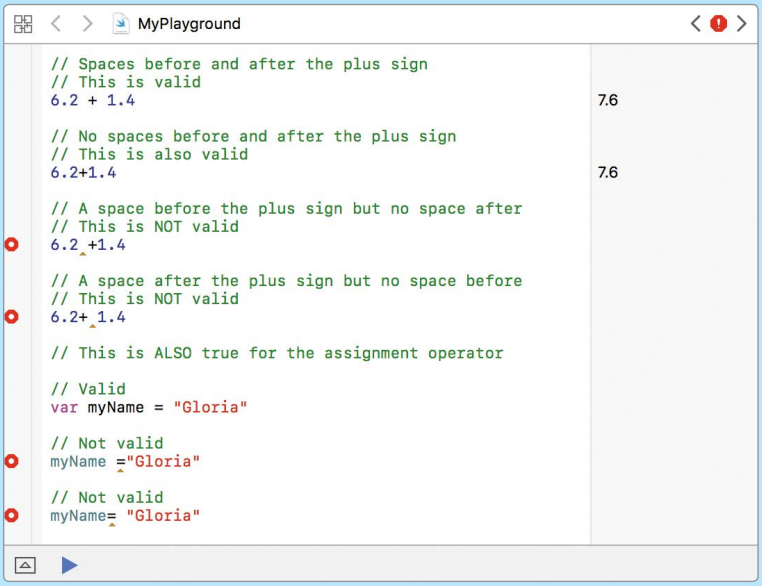


Figure 2-7: Make sure that you have the same number of spaces on each side of your operators.

There is one important thing to note: you can only use a math operator on variables or constants that are the *same* data type. In the previous code, `three` and `five` are both `Int` data types. The constants `half` and `quarter` are `Double` data types because they are decimal numbers. If you try to add or multiply an `Int` and a `Double`, you'll get an error like the one in Figure 2-8.

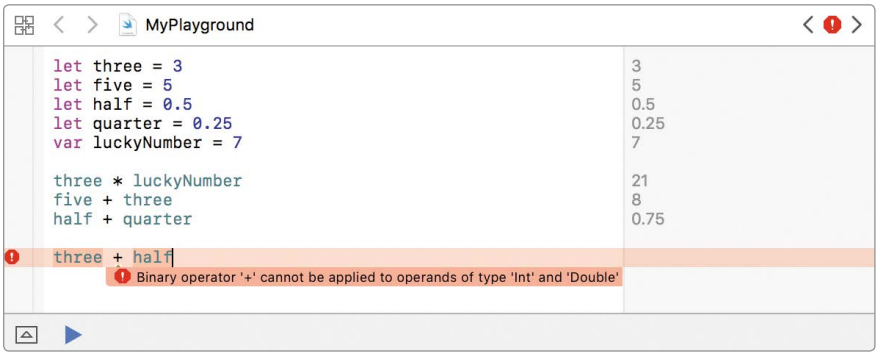


Figure 2-8: In Swift, you cannot do math on mixed data types.

But what if you really want to do math on mixed data types? For example, let's say you want to calculate one-tenth of your age:

```
var myAge = 11 // This is an Int                    | 11
let multiplier = 0.1 // This is a Double            | 0.1
var oneTenthMyAge = myAge * multiplier
```

The last line will result in an error because we're attempting to multiply an `Int` by a `Double`. But don't worry! You have a couple of options to make sure your operands are the same data type.

One option is to declare `myAge` as a `Double`, like this:

```
var myAge = 11.0 // This is a Double                | 11.0
let multiplier = 0.1 // This is a Double            | 0.1
var oneTenthMyAge = myAge * multiplier              | 1.1
```

This code works because we're multiplying two `Double` data types.

The second option is to use casting. (I told you we would come back to this!) Casting is a great solution in this case because we don't want to permanently change `myAge` to a `Double`, we just want to be able to perform math with it as if it were a `Double`. Let's take a look at an example:

```
var myAge = 11 // This is an Int                    | 11
let multiplier = 0.1 // This is a Double            | 0.1
❶ var oneTenthMyAge = Double(myAge) * multiplier    | 1.1
❷ oneTenthMyAge = myAge * multiplier
```

At ❶, we cast `myAge` to a `Double` before multiplying it. This means we no longer have mixed types, so the code works. But at ❷ we will get an error. That's because `myAge` is still an `Int`. Casting it to a `Double` at ❶ did not permanently change it to a `Double`.

Could we cast `multiplier` to an `Int`? You bet! Then we are doing math on two integers, which works fine. However, this results in a less precise calculation because we'll lose the decimal place. When you cast a variable from a `Double` to an `Int`, the computer simply removes any digits after the decimal to make it a whole number. In this case, your `multiplier` of `0.1` would cast to an `Int` of `0`. Let's cast some variables in the playground and see what we get:

```
❶ Int(multiplier)                                   | 0
❷ Int(1.9)                                          | 1
```

At ❶, casting our `Double`, `multiplier`, to an `Int` gives us `0`. This value is quite different after casting, because we lost the decimal place: `0.1` became `0`. This could be a very bad thing in our code if we were not expecting it to happen. You must be careful when casting to make sure you aren't unexpectedly changing your values. At ❷, there's another example of casting a `Double` to an `Int`, and as you can see, `1.9` does not get rounded up to `2`. Its decimal value just gets removed and we are left with `1`.

There's another math operator, the *modulo operator* (`%`), which might not be as familiar to you. The modulo operator (also called *modulus*) gives the remainder after division. For example, `7 % 2 = 1` because 7 divided by 2 has a remainder of 1. Try out the modulo operator in your playground, as follows.

```
10 % 3                          1
12 % 4                          0
34 % 5                          4

var evenNumber = 864            864
❶ evenNumber % 2                0

var oddNumber = 571             571
❷ oddNumber % 2                 1
```

As you can see, the modulo operator is useful for determining whether a number is even (`evenNumber % 2` equals 0) ❶ or odd (`oddNumber % 2` equals 1) ❷.

ORDER OF OPERATIONS

So far we've only done one math operation on each line of code, but it's common to do more than one operation on a single line. Let's look at an example.

How much money do you have if you have three five-dollar bills and two one-dollar bills? Let's calculate this on one line:

```
var myMoney = 5 * 3 + 2         17
```

This assigns a value of `17` to `myMoney`. The computer multiplies 5 times 3 and then adds 2. But how does the computer know to multiply first and *then* add 2? Does it just work from left to right? No! Take a look at this:

```
myMoney = 2 + 5 * 3             17
```

We moved the numbers around and the result is still 17. If the computer just went from left to right, it would add 2 + 5 and get 7. Then it would multiply that result, 7, times 3, and get 21. Even though we changed the order of the numbers in our math expression, the computer still multiplies first (which gives us 15) and then adds the 2 to get 17. *The computer will always do multiplication and division first, then addition and subtraction*. This is called the *order of operations*.

ORDERING OPERATIONS WITH PARENTHESES

You don't have to rely on the computer to figure out which step to do first like we did in the money example. You, the programmer, have the power to

decide! You can use parentheses to group operations together. When you put parentheses around something, you tell the computer to do that step first:

```
❶ myMoney = 2 + (5 * 3)
❷ myMoney = (2 + 5) * 3
```

```
17
21
```

At ❶, the parentheses tell the computer to multiply 5 times 3 first and then add 2. This will give you 17. At ❷, the parentheses tell the computer to add 2 plus 5 first and then multiply that by 3, which gives you 21.

You can make your code even more specific by using parentheses inside of other parentheses. The computer will evaluate the inside parentheses first and then the outside ones. Try this example:

```
myMoney = 1 + ((2 + 3) * 4)
```

```
21
```

First the computer adds 2 and 3 between the inner parentheses. Then it multiplies the result by 4, since that's within the outer set of parentheses. It will add the 1 last because it's outside both sets of parentheses. The final result is 21.

COMPOUND ASSIGNMENT OPERATORS

Another category of operators that you'll use is the *compound assignment operators*. These are "shortcut" operators that combine a math operator with the assignment operator (=). For example, this expression

```
a = a + b
```

becomes

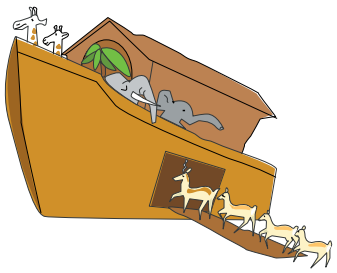
```
a += b
```

You can use these operators to update the value of a variable or constant by performing an operation on it. In plain English, an expression like `a += b` says "add `b` to `a` and store the new value in `a`." Table 2-1 shows math expressions using compound assignment operators and the same expressions in their longer forms.

Table 2-1: Short Form Assignment Operators vs. Long Form Expressions

Short form	Long form
`a += b`	`a = a + b`
`a -= b`	`a = a - b`
`a *= b`	`a = a * b`
`a /= b`	`a = a / b`

Let's see the += operator in action. Imagine that we're trying to write a program to calculate the number of animals on an ark. First we create a variable called `animalsOnArk` and set it to `0` because there aren't animals on the ark yet. As the different types of animals board the ark, we want to increase `animalsOnArk` to count all of the animals. If two giraffes board the ark, then we need to add 2 to `animalsOnArk`. If two elephants board the ark, then we need to add 2 again. If four antelope board the ark, then we need to increase `animalsOnArk` by 4.



`var animalsOnArk = 0`	`0`
`let numberOfGiraffes = 2`	`2`
`animalsOnArk += numberOfGiraffes`	`2`
`let numberOfElephants = 2`	`2`
`animalsOnArk += numberOfElephants`	`4`
`let numberOfAntelope = 4`	`4`
`animalsOnArk += numberOfAntelope`	`8`

After two giraffes, two elephants, and four antelope board the ark, the final value for `animalsOnArk` is `8`. What a zoo!

A FEW QUICK COMMENTS ABOUT COMMENTS

Most programming languages come with a way to write comments inline with the code. Comments are notes that are ignored by the computer and are there to help the humans reading the code understand what's going on. Although a program will run fine without any comments, it's a good idea to include them for sections of code that might be unclear or confusing. Even if you're not going to show your program to anybody else, your comments will help you remember what you were doing or thinking when you wrote that code. It's not uncommon to come back to a piece of code you wrote months or years ago and have no idea what you were thinking at the time.

There are two ways to add comments in Swift. The first way is to put two forward slashes (`//`) in front of the text you want to add. These comments can be placed on their own line, like this:

```
// My favorite things
```

Or they can be placed on the same line as a line of code—as long as the comment comes *after* the code:

```
var myFavoriteAnimal = "Horse" // Does not have to be a pet
```

The second way is used for long comments, or *multiline* comments, where the start and end of the comment is marked by `/*` and `*/`. (Note that we'll use `--snip--` in this book to show where there are more code lines that we've omitted for space.)

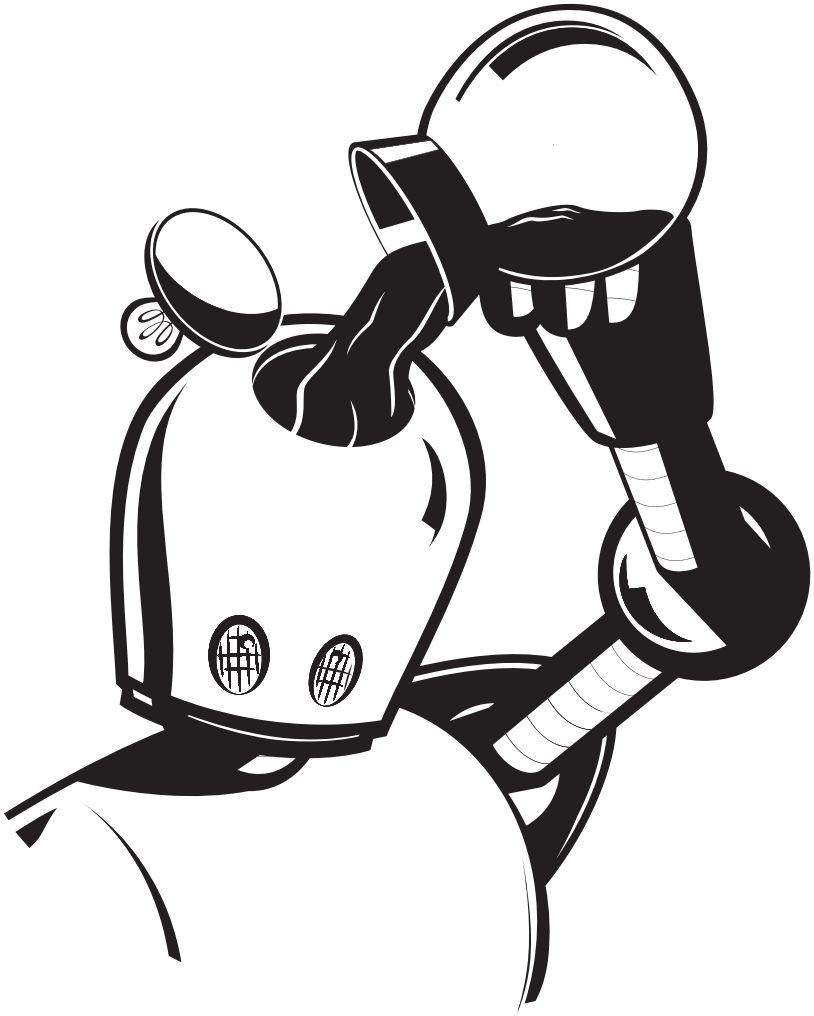
```
/*
  This block of code will add up the animals
  that walk onto an ark.
*/
{
  var animalsOnArk = 0
  let numberOfGiraffes = 2
  animalsOnArk += numberOfGiraffes
  --snip--
}
```

Multiline comments are useful when you debug your code. For example, if you don't want the computer to run part of your code because you're trying to find a bug, but you also don't want to delete all of your hard work, you can use multiline comments to *comment out* sections of code temporarily. When you format a chunk of code as a comment, the computer will ignore that code just like it ignores any other comment.

WHAT YOU LEARNED

In this chapter you learned how to write code in a Swift playground, which lets you see results right away. You created variables and constants and learned how to use the basic data types and operators that you will be seeing again and again as you write your own computer programs.

In Chapter 3, you will be using conditional statements, which tell the computer which code path you want it to go down. The code path is chosen based on a condition's value.



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