

Introduction to Programming for the Independent Student:

A Self-Starter's Course on the Principles and Practice of
Bending Computers to Your Will

Second Edition (2022)

Christopher K. Monson

Copyright © 2020, 2022 by Christopher K. Monson

All rights reserved.

The moral rights of this author have been asserted.

Cover art Copyright © 2020 by Catherine E. Monson

About This Book

Learn the basics of Computer Science and programming by building interactive programs—including simple animations and games—that run in a standard web browser.

This book uses the ubiquitous and popular JavaScript (ECMAScript) programming language as a basis for teaching, covering sufficient syntax and idioms to build simple interactive animations and games.

It highlights fundamentals of computer science along the way, such as boolean algebra, recursive algorithms, and event-driven programming. All concepts are taught with beginners in mind. It has also been designed for teacher who have no prior exposure to computer programming, making this an excellent choice for homeschoolers: complete, detailed explanations are given for every exercise, lab, and test question.

If using this book as a high school text, it is designed to have a workload appropriate for a 1-credit, 1-semester course. It will work best for students who have completed enough pre-algebra to have been introduced to the concept of functions.

When used as a curriculum, each chapter should take about a week to get through, with plenty of reading and hands-on learning every week. A midterm is provided at the end of weeks 5 and 10. Every chapter has a set of exercises to complete, again, with full solutions provided at the end of the book.

I hope you enjoy what has been a fun book to write. The concepts taught here are sometimes simple, sometimes mind-bending, and always powerful enablers for anyone who wants to learn to do just a little more with the devices we have all around us.

I think it's worth the journey. I hope you do, too.

About The Author

Dr. Chris Monson has been writing software and designing systems since 1992. He received degrees in both Electrical Engineering and Computer Science from Brigham Young University long ago: before the invention of either electricity or computers, if his children are to be believed.

He has a Ph.D. in Machine Learning and Optimization, with a focus on Bayesian techniques and modeling. After obtaining his final diploma, he worked at Google for about 11 years as a senior software engineer, 3 years as the CTO at Data Machines Corporation, and as of 2022 is working at Meta in the field of Augmented and Virtual Reality. On the side, he occasionally teaches the Cloud Computing Security course at Johns Hopkins University.

He still writes code and loves it.

*For CC, who is better at math than she thinks, and more
inspiring than she knows.*

Contents

Introduction and Installation	1
Why JavaScript?	1
What You Need	2
The Browser, the Web, and Programming	6
You Fetch and Interpret Stuff	11
1 Programming and Developer Tools	13
Your First JavaScript Expressions	13
Programs in Files	19
Your First JavaScript Program	21
More JavaScript Syntax	27
Putting It All Together	33
Exercises	33
2 Function Calls and the Debugger	37
Functions in Algebra	37
Calling Functions in JavaScript	45
Functions That Produce Values	46
A Debugging Interlude	48
Exercises	55
3 Writing Functions and Handling Events	59
Writing Functions	59
Variable Scopes	65
Functions Are Values	71
Summary	78
Exercises	79
4 Objects	83
Objects Are Containers	83
The Console	87
The Standard Library	89

Drawing Pictures	94
Listing	101
Exercises	102
5 Reacting Logically	105
Boolean Logic	105
Conditional Expressions	120
If and Else	124
Else If	128
Exercises	134
Midterm 1	138
6 Iteration Through Recursion	143
Gauss and the Sum of Integers	143
Recursive Dumb Sums	145
Drawing Lines	150
A Recursive Grid	154
Onward	160
Exercises	160
7 Arrays, Loops, Switches, and Randomness	163
Arrays	163
While Loops	166
Do-While Loops	173
For Loops	173
Tiling a Canvas	176
Nested Loops	178
Color Cycling Using Else-If	179
Switch It Up	181
Arrayed in Color	184
Random Numbers	187
Exercises	190
8 Timers, Closures, and Animation	199
One Thing At A Time	199
Timeouts Revisited	200
Closures	203
Simple Timeout-Based Animation	206
Summary	210
Exercises	211
9 Smoother Animation Using Time and Animation Frames	215
Animation Frames	215
Smoothness and Time	218

Another Wrinkle In Time	223
Exercises	226
10 Click and Key Events	227
Are We There Yet?	227
Adding a Button	228
Actually Pausing Stuff	231
Changing Button Text	236
Canvas Clicks	238
Key Events	240
Summary and Full Listing	241
Exercises	243
Midterm 2	244
11 Behavioral Abstractions and Multi-File Programs	247
More Files	247
Hiding the Plumbing	248
An Animation Function	250
Animating With Abstraction	251
Abstractions That Provide Behavior	259
Summary and Listings	264
Exercises	266
12 Our First Game: State, Configuration, Clocks, and Winning	269
A Matter of State	270
Where To Store State	271
Animation Augmentation	273
Writing the Game	279
Listings	293
Exercises	297
13 Snakes On a Page	301
Game Scaffolding	302
Designing the Game	303
Positions Are Picky	304
Representations	310
Drawing the Snake and Food	311
Moving the Snake	312
Changing Directions	317
Limiting Motion	318
Eating	319
Crash	321
Listings	323

Exercises	327
14 Abstractions With Classes and Objects	329
Objects as Data Containers	329
Animation Class	333
Using Instances With Events	340
Keep it Classy	343
Exercises	344
15 Practical Web Programming	345
A More Complete Document	345
Head Scripts	346
Adding Elements Dynamically	350
Countdown	351
Dates	352
Summary and Conclusions	356
Listings	356
Exercises	358
Final Exam	360
16 Answer Key	361
Solutions	363
Chapter 1 Solutions	363
Chapter 2 Solutions	370
Chapter 3 Solutions	378
Chapter 4 Solutions	383
Chapter 5 Solutions	389
Chapter 6 Solutions	411
Chapter 7 Solutions	419
Chapter 8 Solutions	445
Chapter 9 Solutions	452
Chapter 10 Solutions	455
Chapter 11 Solutions	465
Chapter 12 Solutions	467
Chapter 13 Solutions	474
Chapter 14 Solutions	478
Chapter 15 Solutions	481

Introduction and Installation

Welcome to the amazing world of computer programming! This text will teach you foundational concepts for programming, and then you will start writing basic games. You know enough to get started if you can already

- Touch type on a computer,
- Manipulate basic algebraic expressions, like “solve for x in $x + 3 = 4$ ”, and
- Understand what an algebraic function is, often written like $f(x)$.

Even if you are a little rusty with typing, or have forgotten some algebra, don’t worry too much; we will review the important parts as we need them. By the time we’re through, you will be able to create your own computer programs completely from scratch using the JavaScript programming language, and they will run in a web browser.

This book is all about laying foundations while getting stuff done. Because it is an introductory text, this is one of those times when skipping around won’t usually work: every concept builds on those before it, and with rare exception, every chapter assumes that you have successfully completed the previous ones. Remember, you are essentially learning a new language, and that takes practice, not just memorization. If you choose to do the exercises and labs in this book, they will really help you get the kind of practice that will lead you to success.

The exercises are neither plentiful nor mind-numbingly diabolical, like they can be in some classes. Here they are straightforward and purposeful, designed to help you feel more comfortable with what you have learned. It’s worth giving them a try, in order. A complete answer, with a detailed explanation of how to arrive at that answer and—when applicable—why it’s a good answer, is given for each one. Nothing is left as an exercise for the reader. The answer key is detailed and informative.

Why JavaScript?

Why would we use JavaScript instead of another popular programming language like Python? Python is indeed a lovely language for learning programming, but JavaScript has the advantage of already being everywhere in a way that no other language is right now. If you have a phone, it contains a JavaScript interpreter (possibly more than one!). If you use a web

browser, you are running JavaScript all the time. That means a couple of things for you, the most important of which is that you probably already have everything you need to get started. You really can't say that about many other programming languages.

The key here is this: you can learn to program using what you probably already have: a web browser and a text editor.

By the way, little asides set apart from the text, like this one, can often be ignored. They are just additional information, but do not contain anything critical to your success in the course. This particular aside is a good example of that: feel free to ignore it.

Web Assembly (WASM)^a is changing things here, but is not quite ubiquitous and stable enough to warrant using a transpiled language for this text, at least not yet. It's a very exciting technology that is opening up all kinds of fun language choices in the browser, and I look forward to that continuing, but for now we're keeping things simple, using the browser's native programming language.

^a<https://webassembly.org>

The enormity of JavaScript's success was probably not anticipated when Brendan Eich¹ created it in 1995 in order to give web browsers the ability to do things more interesting than just display text, links, and images. That small, hastily-assembled invention ended up transforming the way the entire world consumes information and creates applications, and it is now the world's most used programming language by a substantial margin.

In many respects, it's not the most endearing language, though it does have some great qualities. It has its share of warts, it sometimes isn't the most expressive thing to be working with, and it can make certain things harder than they need to be (not unlike English, really). It is, however, powerful enough, expressive enough, and easy enough to give us a running start on programming games, which will give us a running start on computer science and programming in general.

Another extremely important reason for using it as a starting language is the huge set of built-in browser facilities that it gives us ready access to. Web browsers are incredibly capable programs. They can process speech, produce music, draw graphics, do animation, respond to keyboards and game controllers, and on and on. JavaScript is still (for now) the main gateway for interacting with all of that.

What You Need

The short version:

1. Install Chrome or use a Chromebook or similar device.

¹https://en.wikipedia.org/wiki/Brendan_Eich

2. Install a programming text editor like `VSCode`, or if you want one in Chrome, install `Text` (search for “chrome text editor” to find this).

That’s it. That’s the short version. If you can do the above without help, you now have all of the necessary tools. Get those two things and you can skip to the next chapter. Or you can read the rest of the introduction. I hear it can be entertaining and informative.

The slightly longer version includes details about how to accomplish the above, what options you have, and why.

To get set up for programming in JavaScript *in principle*, you need

- A *programming* text editor, and
- A web browser *with developer tools* (like Chrome or Firefox).

Note that as of the time of this publication (late 2022), a browser on a mobile device like an iPhone, iPad, Android tablet (or phone) **will not work as smoothly for you**; they don’t have developer tools. You can get by without them, but you will need to skip the section on debugging, and will need to employ some workarounds for looking at the values of variables. The first chapter or so might be extra confusing without them.

When you write programs while reading this book, you will follow these basic steps:

- Write code in a file and folder of your choosing, then
- Open that file in your browser and see it work.

You will also need a physical—not on-screen—keyboard. An on-screen keyboard leaves programmers feeling frustrated and empty inside.

Why do you need a physical keyboard? It turns out that programming uses a lot of symbols, and most virtual on-screen keyboards lack them; they are short on space and the symbols used all the time in programming are usually hard to get to. That includes simple things like the `Tab` key and curly braces. Furthermore, they are much slower to type on in general, usually supporting a maximum of two thumbs (I have way more than that), and that affects your ability to get your thoughts into a file.

Programming is all about communicating what’s in your head to the computer, and that communication is greatly aided if you can type on a physical keyboard. It’s also best if you can do so without looking at your hands all the time. If you want to learn to program but can’t touch type without peeking, now would be a great time to practice that skill a bit. Most people can get a lot better at typing after just a couple of days of practice, using one of the many freely available online tutors, some of which don’t require an online account and don’t have ads^a. It is absolutely worth it to spend time on this.

^a<https://www.bouncingchairs.net/entrotype>

We obviously can’t cover all possible setups together in this book, so I am presenting just one way to do things here. Don’t let that stop you from exploring other options, and of course you

can always change later.

A Web Browser

Install Chrome² if you don't already have it. If you are using a Chrome OS device like a Chromebook, you're already done. Enjoy a refreshing beverage.

Chrome on Android or iOS is not recommended because it is limited in various important ways, including the inability to install extensions like a text editor, access to developer tools, and other surprises. Some of these limitations, unfortunately, also apply to student Chromebooks and machines with family protection installed, so the book has been designed to make it possible to proceed even if you are in a restricted environment. It just won't be quite as easy. If you can, get access to an unrestricted computer for the purposes this course. If not, carry on, we'll call out places where you can proceed even if restricted, it will just take a little more effort.

A Programming Text Editor

Text editing is at the heart of most programming, so we obviously need a text editor. What might be less obvious is that not all text editors are equally good for this task. Programming languages are a lot more *structured* than the text we write to one another in emails and essays and online flame wars, so it is far more useful to have a “smart” text editor for programming: one that helps readers to understand code at a glance, and that handles some repetitive tasks for you. This course recommends a couple of editors below, but if you already have an editor that understands JavaScript and HTML, then by all means use what you're comfortable with. Otherwise, the setup section below assumes you need to install one and are using Chrome as your browser for this course. If you are unsure, just follow instructions below to get started. **You can always change later!** You're just writing text and loading it into a browser, after all. The actual tools used to do so are generally interchangeable.

Here I have two recommendations depending on your situation:

- **Text**³ or something similar. If you are using a Chromebook, it might already be installed. If you are just using Chrome and want to install it, you can: a web search for “chrome text editor” produces results for an editor called `Text`. `Text`'s github page has a link to a stable version on the Chrome Web Store.
- **VSCode**⁴: If you are using a general-purpose computer (like one running Mac OS, Windows, or Linux, for example), VSCode is your best bet.

If you can't install a text editor, and you don't have one already, you really are stuck. The text editor is absolutely required. Install one now if you haven't, already!

²<https://chrome.google.com>

³<https://github.com/GoogleChromeLabs/text-app>

⁴<https://vode.visualstudio.com/>

Once you are done, make sure you can do the following:

- Create a new file,
- Save it somewhere memorable, and
- Open it again later to edit it.

If you can do all of that, you are ready to continue.

What's so special about a text editor tailored to programmers, anyway? Are programmers really that special? Well, yes, yes we are! Also, there are two major features that really set programming text editors apart from others: **syntax highlighting** and **auto indent**. Syntax highlighting means certain language constructs are given different colors and text formatting to make them stand out. Auto indent does basically what it says: when moving to the next line, you start at the same indent level as the previous line, which is usually what you want (some editors are even smarter than this). These are incredibly useful—even essential—tools for programmers.

Oh, and text editors *just produce text*. That's it. They do not produce a lot of formatting information like word processors do: the text you see is *exactly* the text you save. That is **extremely important**. If you are thinking of using a word processor for programming, run away fast, then install a good text editor. Google Docs, Word, OpenOffice, and others like them are simply *not going to work*.

If you just can't wait to get started programming now, by all means turn straight to the first chapter and go for it! You might find that some of the information below about how the web works to be interesting, maybe even helpful, but you can always come back to it. Books are pretty amazing that way.

Access to the Internet

It is possible to work through this entire course—after ensuring that you have a suitable browser and text editor—without ever accessing the internet again. That's by design, since some people have restricted access and this course is meant to be for *everyone*.

That said, there is at least one internet site that you will appreciate having access to if possible, and that is the Mozilla Developer Network (MDN)⁵, where the official JavaScript documentation resides. There are other popular sites that pop up if you do a web query for various JavaScript topics (like W3 Schools⁶ or Stack Overflow⁷), but the official documentation is what we'll rely on in the text.

⁵<https://developer.mozilla.org>

⁶<https://www.w3schools.com>

⁷<https://www.stackoverflow.com>

The Browser, the Web, and Programming

Let's get our mental models straight for the internet and how browsers work before we go further into setup, starting with this question: what is a "web browser"?

At its most basic, a web browser is just another computer program, one that fetches stuff you ask it to fetch, then interprets what it receives for your benefit. Let's talk about each of these "fetch" and "interpret" steps in turn.

Browsers Fetch Stuff

The browser gives you the ability to get information from other computers connected to your network. For many people, that network includes what we call "the internet", a whole bunch of computers (similar to yours in many ways) that are connected in such a way that they can talk to one another.

What does it mean that they "talk to one another"? Let's use a Google search as an example.

Suppose you open up a brand new blank tab in your browser, and nothing is in it. If you go to the location bar and type in `google.com/search`, the familiar Google logo and search box appear. What just happened? It turns out that a mind-boggling amount of stuff happened just so you could see this search box.

First, your browser had to figure out what to do with the text `google.com` that you typed. Computers don't really access each other by *name* on the internet, instead, they access one another by *address*. The browser thus has to first figure out what **Internet Protocol (IP) address** corresponds to the **domain name** `google.com`. It does this by asking a set of **name servers** to **resolve** the name for it. These name servers are listed somewhere in your network configuration, usually put there without your help. If you were to go into your computer's network settings, you might see one or more addresses under a "DNS" section: those are the "Domain Name Servers". Often they are addresses owned by your internet service provider, the company you pay to let you see cat pictures.

Thus, the browser first tries to connect to one of those DNS addresses. If successful, it sends over the name you typed, `google.com`, and the name server looks it up. If it finds it, it returns an address for it, say `74.125.29.138` or `2607:f8b0:400d:c0a::64`. These IP addresses (version 4 and version 6, respectively) each represent a single computer somewhere, and your computer now knows how to send messages to it.

After the lookup step, your browser has an address instead of a name so it can finally do what you *really* wanted it to do, which is to get the data available at `google.com`. What data should it get? The stuff that comes after the domain name is called the **path**, which is sent (approximately) as typed to the computer at the address we recently got. That computer, the one we're talking to, decides what information to send us based on that path. In our case, the path is `/search`, so whichever Google-owned computer we are talking to checks to see if it

knows what `/search` means (it does), then sends us the appropriate content, which happens to be the search page.

If a computer on the internet doesn't recognize the path we gave it, like if we say `google.com/notarealpath`, it will send us back an error code, one of the most common of which is "404". That error means the computer we're talking to has no idea what we're talking about; it doesn't recognize the *path*, which is `/notarealpath` in this case.

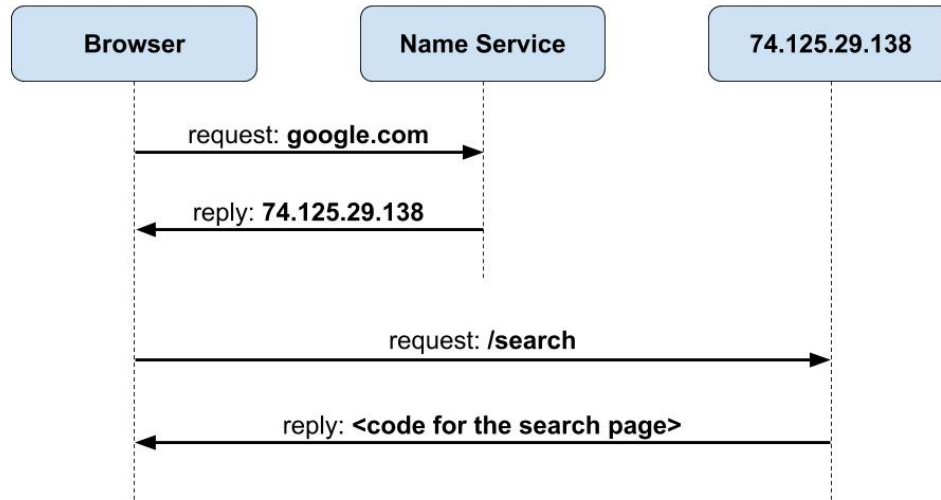


Figure 1: How a browser looks things up.

To summarize, when you type an address into the location bar of your browser, say `example.com/path/to/data`, the browser

1. Contacts one of its **name servers** to look up the address for `example.com`,
2. Receives the address for that name and attempts to contact the computer at that address,
3. Asks for the data indicated by the **path** after the domain name, in this case `/path/to/data`, and
4. Interprets what it receives so that it can present something useful to you.

You can think of this as akin to trying to get a book from a library, when all you have is the name of the library and the name of the book. Let's explore that metaphor a bit and see where it leads us.

Suppose you want to get the book *Computational Fairy Tales* (ISBN 9781477550298, highly recommended reading) from the Baltimore County Public Library. If you were to write that out as a fictional internet location, the parts might look like this:

baltimore-county.library/Computational Fairy Tales

The name part is the place you want to get the book from, and the path part is the book title. In order to get there, though, you first need a location. You do the same things a browser does:

1. Look up the location of the library,
2. Navigate to it in your choice of transportation,
3. Ask for the book at the front desk, and
4. Go home and read it.

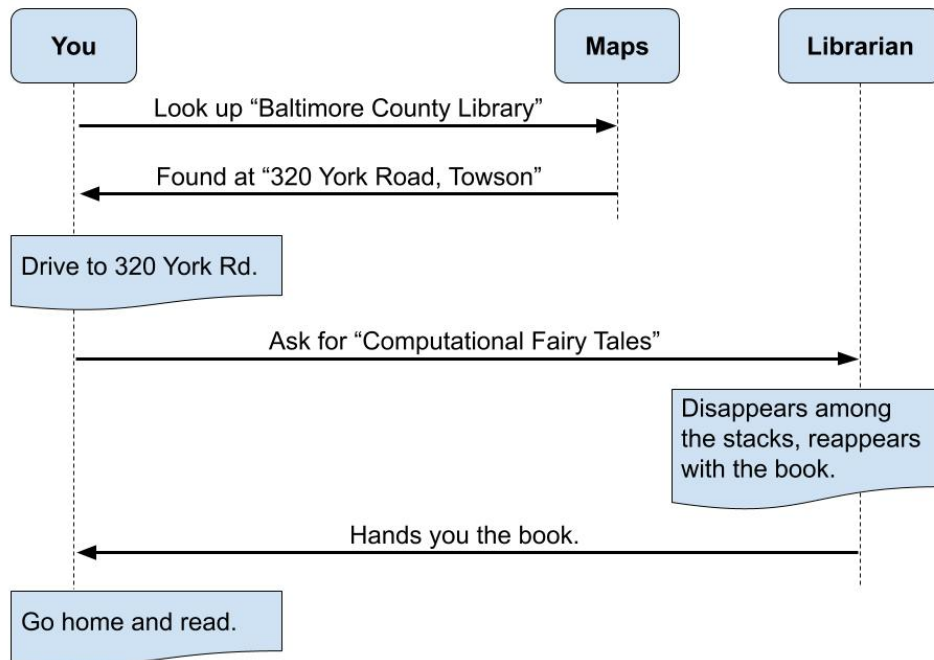


Figure 2: A library analogy for how the browser looks things up.

That’s what a browser does. It finds the address of the domain you ask for (e.g., `google.com`), asks the computer at that address for some content, and then displays the results to you.

If you go to the library and ask for a book that isn’t there, the librarian will give you an “error message”, something like “Sorry, it isn’t here.” Just like a browser, but with more sympathy and less “404 Not Found”.

Browsers Can Get Things From Right Here

Importantly, browsers don’t have to go out onto the internet for everything they display. They can also pull information directly from your computer’s local storage. We will make extensive use of that fact in this course, since we will be writing code, storing it on the computer, then

asking the browser to interpret what we've written. The difference is in the **protocol** part of the address. When we access things online, we typically see a prefix like `https:` (or `http:` for insecure sites). In this course, we will be using the `file:` protocol, which pulls data from local storage.

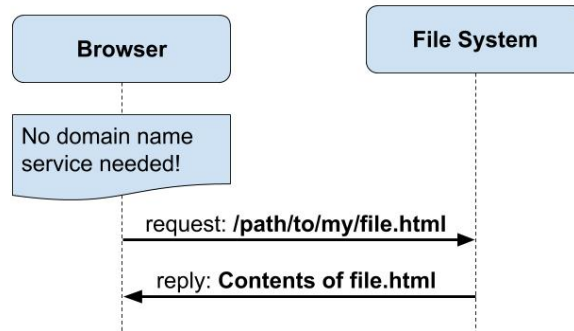


Figure 3: No address lookup for local files.

HTTPS stands for “HyperText Transfer Protocol Secure”. We won’t get further into what the protocol section of a location means in this course, except to say this: when using the `file://` prefix, the browser treats everything after that as a path to a file. Since a file path is already something the computer understands, it doesn’t have to look it up somewhere else first. Thus, for local files there is no separate “lookup the address” step: the address is simply “files right here”.

Many protocols are followed by `//` to separate them from the stuff that follows. For HTTPS, that’s a domain name. For files, it’s a path to a file on your local system, and that path often starts with the “root folder”, called `/`. Thus, with `file:`, we usually see *three* of those forward slashes following it, after which is a familiar path. The Downloads folder on a Mac, for example, might look something like `file:///Users/myname/Downloads` because the path of interest starts with a forward slash: `/Users/myname/Downloads`.

If that didn’t make a lot of sense, don’t fret. It only matters if you ever saw `file:///` in a location bar and were confused as to why. Now you know.

We will be making use of the “get files from this computer” feature of our web browser in this course. We will write code, save it on the computer, then instruct the browser to read it and interpret it. Basically, we will write down the dance moves, store them on disk, then the browser will read them and cut a rug for us.

Browsers Interpret Stuff

We mentioned briefly that browsers do some interpreting of what they receive so that they can present something meaningful to you, a human user. But what does that mean? It means that

what you *see* is not actually what you *get* in the above process: there is an interpreter between you and the internet, and it is making things nicer for you than they really are.

When your browser obeys a request to get something like `mozilla.org/index.html`, what it receives is a bunch of text that isn't usually very nice for people to read in raw form. You can see this text by finding the `View Source` option after loading a page, often available as a "right click" menu item, or under a `View` menu. In fact, you should try that right now. Part of learning to program is learning to lift the covers off of things and peer beneath them to see how they work:

- Open up <https://mozilla.org/index.html>, then
- Right-click and select `View Source` (or find it in the menu somewhere).

If "right-click" doesn't make sense to you, you're not alone. Back in ancient history, but long after I was born, people interacted with their computers using multiple buttons on their pointer devices (whether a mouse, a trackpad, a trackball, etc.). The left button is the one under a right-handed user's strongest finger (the index), so it's the primary button people used for clicking stuff. The right button was used for extra special but not terribly frequent needs, like bringing up a context menu for a particular item.

OK, that's all very boring and doesn't help you much. Here's what you need to know: when I say "right-click", I mean "tap something in the standard *but alternate* way". On computers with left and right buttons, that's obvious (unless you have them switched because you are fortunate enough to be left-handed). Many computers do not have multiple buttons, or even any buttons at all. In those cases, you can try holding `Ctrl` down—or if you have a Mac you can try holding `Command` down—while clicking. Some trackpads let you click using two fingers at once, instead of just one, and that often does what we want here. Finally, if none of that works, sometimes a "long press" triggers the secondary click functionality.

There are many ways to do it, all depending on your specific computer. On Chromebooks and Macs, I have verified that two-finger clicking does the right thing.

Back to looking at `View Source`: what you see there is the "source code" for your page. As anticipated, it's not very nice compared to what you usually see.

The "source" is what the browser uses to assemble what you see and interact with in the browser. It's a bunch of instructions that describe the page. The browser interprets all of this, laying out the page as instructed, but hiding the actual instructions from you. It also runs any JavaScript code it finds. It does that using a built-in **JavaScript interpreter**.

The word "interpreter" is not an accident. You can think of what the browser is doing as speaking a set of languages that you don't (yet) speak—HTML, CSS, and JavaScript, mostly—and translating them into something you can use. Browsers do an awful lot of work behind the scenes just to show us cat videos.

You Fetch and Interpret Stuff

Now it is time to engage the amazing wet computer inside your head. It is time to turn the page, fetch some text, and interpret it so that you can say you have learned something more about programming in JavaScript. Let personal growth and satisfaction begin!

Chapter 1

Programming and Developer Tools

Let's get this browser to dance for us. Not literally, of course: that might be terrifying. Let's make it do something that we tell it to do.


You learned in the introduction that you need a text editor and a modern browser with developer tools. If you don't have those things, or you're not sure whether you do, stop now and go get that taken care of.


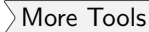
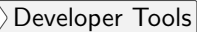




Oh, and you should probably bring your brain along for the ride, too.

Your First JavaScript Expressions

Are you ready to write some code? Excellent. Let's jump in. Cue anticlimactic introduction in 3, 2, 1....

Developer Tools







With an appropriate browser installed, you also have access to some powerful developer tools. These can be accessed by finding them in the appropriate menu. You might need to poke around to find these, including looking in Chrome's "overflow menu". This is usually located to the far right of the location bar, and usually looks like three vertical dots , though it can sometimes look different if your browser has an update ready.

Let's find that now. In Chrome, go to the overflow menu, then follow this path:   . Then select the  tab. You can also use the menu at the top of the screen if there is one (like on a Mac):   .

If you don't see a console item, or can't get to it, you might be using a version of Chrome that doesn't support it, such as on a mobile device like an Android or iOS phone or tablet. Or, your account might have family or school restrictions that disable developer tools; Google's Family




Link, for example, disables the console in the browser. You can do many things in this course without the console, but you may find it somewhat limiting in the very beginning.

If there is no way for you to get access to the console, that's probably okay, but you will need to write more code in text files and load it with additional alert outputs. This will make sense as you continue reading. Don't despair, even the console is optional here, it's just really, really helpful. What is absolutely **not optional**, however, is access to a real text editor. If you are denied that, you will either need a different kind of computer or more privileges on the one you have.

You might notice some symbols near the final menu entry for “Developer Tools”. These are the keys you can press to get to the tools quickly without traversing the menus every single time. On my Mac, for example, the symbols say I should press  +  +  (meaning  +  + ) to turn the tools on and off. If you memorize the keystrokes that get the developer tools open, you'll have a better time, but the menu works fine.

We are all set up to use the developer tools, so let's write some JavaScript!

Even though it is assumed that you will use Chrome for this course, you can use something like Firefox, as well. Here are a couple of useful Firefox tidbits, then you'll be on your own with getting to and using the developer tools (but it's all pretty similar):

- Developer Tools:  >>  >> 
- Console typing area: always at the bottom of the console.
- Programming Editor: you pick. If you're running Firefox, there's a good chance that you can install non-browser-related software. VSCode^a is a solid choice if you are unsure where to begin.

^a<https://code.visualstudio.com/>

The Console

If you don't have access to a console, read along anyway, and skip the exercises until the text editor is introduced.

For now, start by opening the console as described in the previous section (from the overflow menu, or by pressing the indicated key sequence for your machine). The console can do a number of things, but we're just going to focus on one of them right now: you can type short JavaScript expressions and see their results immediately. It is very useful for fiddling around and testing ideas.

We will use the console as a simple calculator at first. Find the spot in the console that lets you type, typically prefixed with `>`. In Chrome, you can type at the first empty space in the console.

Did you find a place where you can type something? Type in the following and hit **Enter**:

:console:

```
> 10 + 10
```

What do you see? If you're using Chrome, you should see something that looks like this:

:console:

```
> 10 + 10
< 20
```

Here is an image of what this looks like on my machine:

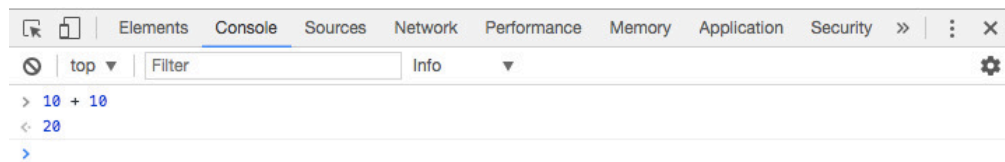


Figure 1.1: The Chrome Developer Console evaluating $10 + 10$

A Simple Calculator

Let's play around with some other numeric operators. If you type the prompts below, you should see the answers given:

:console:

```
> 10 + 10
< 20
> 10 * 10
< 100
> 5 - 2
< 3
> 2 - 6
< -4
> 3 / 2
< 1.5
```

Every time you type something in the prompt and hit “Enter”, the browser’s JavaScript interpreter springs to life and attempts to compute the value of what you typed. In some browser consoles, answers will begin to appear while you are typing, which is fun. When it succeeds at interpreting what you wrote, the value is displayed. If it doesn’t know what to do, it displays an error. Try this in the console:

```
                                     :console:
> 2 @ 2
⊗ Uncaught SyntaxError: Invalid or unexpected token
```

What does it produce? You should see a `syntax error` of some kind (it may be different than the example here), meaning that what you typed doesn’t look much like JavaScript to the browser; it doesn’t follow the prescribed patterns of the language. Those patterns, in natural human languages like English as well as computer languages, are called “syntax”. In this particular case, the JavaScript interpreter in the browser has no idea what it means to apply `@` between two numbers; that’s not an operator that it recognizes, so it’s a syntax error.

There are many other things you can do with numbers, but for now we will stick to the basics: addition (+), subtraction (−), multiplication (*), and division (/). If you are using a modern browser, you can also use `**` to compute powers (exponentiation). Go ahead and play around with those for a bit.

Parentheses also work in the way that you might expect: they force certain operations to go first. Try this:

```
                                     :console:
> 3 * 2 + 3
< 9
> 3 * (2 + 3)
< 15
```

Just like familiar arithmetic expressions, JavaScript respects an order of operations: multiplication and division take precedence over addition and subtraction, except when you force the issue by adding appropriate parentheses, as above.

Congratulations! You have written something in JavaScript! Yes, those arithmetic expressions are actually valid JavaScript, and executing them produces the answers just like a calculator would.

Variables

One of the first things you will need to know is how to work with **variables**. In JavaScript, variables behave a lot like they do in algebra. There are three things that you can do with a variable:

- **Assign:** give it a value.
- **Evaluate:** get its value.

- **Declare:** make a new variable without a value.

The difference between mathematical variables and programming variables is usually rooted in the meaning of `=`. In math, it defines a *relationship* between the things on either side of the `=` sign: they are equal to each other. In programming JavaScript and similar languages, it is an *operation* that stores the value on the right into the variable on the left. In math, this means you can have *anything* on the left, but in programming, you can only have a single variable. The consequences often feel the same for simple expressions, so this difference is not typically a problem for new learners.

In JavaScript the act of **assigning** a variable also **declares** it at the same time, so we will skip the idea of declaring things for now and just focus on assignment.

To **assign** a value to a variable, you name it and set it equal to some expression, like this: `x = 15` or `y = 1 + 2 + 3 + 4`.

:console:

```
> a = 10
< 10
```

As you can see, the result of this assignment is the value assigned. Now you can get that value back any time you want by just mentioning the variable name by itself:

:console:

```
> a
< 10
```

When we “mention” a variable like this somewhere other than the left side of an assignment, we are **evaluating** it. That means we are “getting its value”. In general, the concept of evaluation is exactly that: turning an expression into a value. Anything that can become a value can be on the right side of an assignment, and anything that can be assigned (like a variable) can be on the left.

Note—and this is **extremely important**—that variables and other identifiers in JavaScript are all *case-sensitive*. This means that the variable `a` is not the same as the variable `A`. For example, the function `alert` exists, but the function `Alert` does not: they are completely different words as far as JavaScript is concerned.

With that out of the way, here are a few more examples of variable assignment and evaluation:

:console:

```
> a = 5 * (2 + 1)
< 15
```

Here we assign the value of the expression `5 * (2 + 1)` to `a`. The console helpfully tells us that `a` was assigned to contain `15`. We can be sure about that by just asking it to evaluate `a` afterward:

```
                                     :console:
> a
< 15
```

Yep, that's what we want. But we can also evaluate the variable `a` inside another expression. Check this out:

```
                                     :console:
> b = a + 10
< 25
```

In case you are wondering why the console keeps showing you the value of assignments, here's a forgettable little tidbit: the console only shows the value because this is a *pure assignment* without an accompanying declaration keyword like `var`. When assigning and declaring at the same time, the console shows `undefined`. Don't worry about that right now, though. It's unnecessarily distracting. The most important thing is that your variables take on the values that you assign to them, and *sometimes* those values are also displayed immediately in the console. If they aren't, you can always find a variable's value by typing it in and letting the console evaluate it.

Now `b` contains the number 25. How did it get that? Let's pretend to be the interpreter for a moment, looking at `b = a + 10`:

"I see that I'm assigning the variable `b`.

"That doesn't exist, so I'll create it. What's on the right side of the assignment?

"It looks like I need to get the value of `a` first. That's 15. Adding that to 10 gives me 25.

"I'll assign that to `b` now."

The process is more simply explained here:

- Evaluate `a`, get 15,
- Evaluate `15 + 10` to get 25, and
- Assign `b` to be that value: 25.

All kinds of combinations like this are possible, and they all rest on these twin ideas of **assignment** and **evaluation**.

You might be tempted to think that after this runs, changing `a` to something else will also change the value of `b`, since that's how it would work in algebra (where `=` defines a relationship that must be true all the time). It doesn't work that way in JavaScript: we are **assigning** a value to `b` at a particular moment in time, not defining `b` in terms of `a`. If we change `a` after setting `b`, it doesn't do anything to the value of `b`; those two variables, once set, are completely unrelated. To illustrate this, consider the following (slightly condensed for

clarity) console session:

```
:console:
> a = 10
> b = a + 5
> b
< 15
> a = 15
> b
< 15
```

When `b` is assigned, it captures the current value of any variables mentioned in the right side of the assignment expression and uses those. It does not establish a long-term relationship with those variables. Thus, even though the value of `a` was used to make `b`, changing `a` later does nothing to the value that `b` is assigned.

If you wish that `a` would change whenever `b` does, then what you really want is a **function**, not a variable assignment. We will talk about functions in the not-too-distant future.

Programs in Files

The console is really great for testing out simple bits of code, but we will quickly want to graduate to files that contain more code than is reasonable to type into the console. It's time to start using a text editor. If you don't have one installed, going through the introduction first will get you prepared for the next part.

Starting a new program for this course will typically look something like this:

- Open a file in your text editor,
- Write some stuff in the file and save it, and
- Instruct your browser to open (and interpret) that file.

If you are all set up and already know how to do all of this, then you can skip ahead to “Your First JavaScript Program”; the next sections are just very, very detailed versions of the above bullet points. If any of the above sounds unclear or unfamiliar, read on. All will be explained.

Writing a File

You might want to create a special folder for your programs. That's optional, though, at least for now. As long as you can write files and find them again, you're in good shape.

Speaking of writing files, open up your programming text editor and try these steps:

- Open your text editor (e.g., “VSCode” or “Text”),
- Write some text, like `Hello, world!` into a blank file,
- Save the file, naming it `hello.html` and putting it somewhere that you can easily find it again.

To be sure that you can find it, try closing your text editor and opening it up again. If you can get to your file, that's a good sign.

Note that you need to be very careful to ensure that the *only* characters at the end of your file name are “.html”. Watch out for trailing spaces, for example, as they can cause the browser to become confused. That dot-separated suffix is an important signal to the browser, indicating what kind of file it is reading, and it can be fairly picky about it.

File Managers and Local Files






Every operating system has a file manager of some kind. At a minimum, it will let you find and open files that are already there. For example, in Chrome OS the file manager is an application called “Files”. If you open that, you can navigate around places like your Downloads directory or any external storage (like SD cards) you may have attached.

Windows has “Explorer”, Mac has “Finder”, and Linux distributions all have their own thing. If you have access to a command shell and know how to use it, that will work great, as well.

You next need to find your file using the file manager (not just the text editor) so that we can load it into the browser and see what it does.

Opening In the Browser

Creating and opening text files in a text editor is essential for this course, but so is opening a file in the browser, where all the magic happens. The text editor is an author's tool, and the browser is a viewer's tool: the actor that performs the author's script for the audience. It's time to ask it to read what you've written.

If you are on a Windows/Mac/Linux machine, you should be able to use the   menu in Chrome to find your file and open it. If you are using Chrome OS, there is no  menu. You can open a file using the keyboard shortcut  + , which just brings up the “Files” app. Navigate to the file you created and open it in the usual way (e.g., double-tap). It should open in the browser by default if it ends with `.html`, which it does if you followed the previous steps.

If it doesn't open in the browser from the Files app, try opening a blank tab in your browser, then drag your file from the Files app to that new tab. That usually works.

Note that if you fail to name your file in a way that ends with `.html`, the browser won't interpret it, it will just show you code. Thus, if you drag your file into the browser and it just shows you what you typed, you might not have named it properly. If all you typed was text (like “Hello, World!”), then it won't matter either way.

With the ability to create and edit text files, and to view them in the browser, you now have all of the basic skills that you need to start writing programs. Don't worry if this seems like a lot to remember; we will review later.

If you wanted to do this the “real” way, you would run a small web server on your computer, then access your code using the HTTP protocol. You are of course welcome to try this approach. If you are happy sticking with “everything is in Chrome”, you can download the “Web Server for Chrome” app from the Chrome Store. It’s nice and simple: you tell it where your files will live, and then you can access them via a URL like this:

`http://127.0.0.1:8887/myfile.html`

Here `myfile.html` lives in the directory you told your web server about, `127.0.0.1` means “this computer” (often called `localhost`), and `8887` is the port number the web server is listening on (configurable).

Why mention this here? Because if you do want to try using a web server, it can remove some restrictions on where your files can reside. It would, for example, allow you to store your multi-file programs in Google Drive and serve the files from there, where if you choose not to use a web server, you are limited to single-file programs or programs on local storage (because of strange file path behavior in Google Drive on Chrome OS).

This is not at all critical to learning to program, especially in this course. Everything in this little aside is for those interested in trying it out. It can have benefits, but they are not important to your ability to learn from this book.

Here are steps to get this working, if you would like to try:

- Go to the Chrome store,
- Search for and install “Web Server” under “Apps” (made by “chromebeat.com”),
- Run it.
- Use the “Choose Folder” option to select where you want all of your code to live. This can be in Google Drive, if you want, and then you get your stuff backed up more easily.
- Click the URL that appears (mine defaults to `http://127.0.0.1:8887`: again, `127.0.0.1` is the IPv4 address for `localhost`, meaning “this computer”).

What you will see at that point is a list of all of the files in your chosen directory, right there in your browser. If you start saving your code in there, you will be able to pull it up in your browser pretty easily.

Again, ignore this if it seems like too much, too soon. It’s not terribly important, but it can be terribly interesting. It can also just be terrible; don’t get stuck here.

Your First JavaScript Program

With a basic understanding of variable assignment and evaluation, we can write a program in a file, then open the console to see what the program did. That gets us most of the way through the introductory material, and from there we can fiddle with code and learn as we go

without all of the tutorial baggage weighing us down.

Remember the text editor that you have all ready to go? Now is the time to start using it. As a reminder, here are the steps we are going to take:

1. Start your text editor.
2. Write some code into a file and save it as `firstprogram.html`.
3. Load that file in the browser.

We will go over each of these steps in detail at this early phase. If they all look familiar enough to you that you don't need help, that's great—skip to the Write Some Code section.

Start Your Text Editor

This is the first step (shown in bold) of the process here:

1. **Start your text editor.**
2. Write some code into a file and save it as `firstprogram.html`.
3. Load that file in the browser.

A text editor is just a program that you run, and it allows you to read, edit, and save plain text files in permanent storage. A good text editor, as discussed previously, will also aid you in programming by coloring syntax and automatically indenting your code for you.

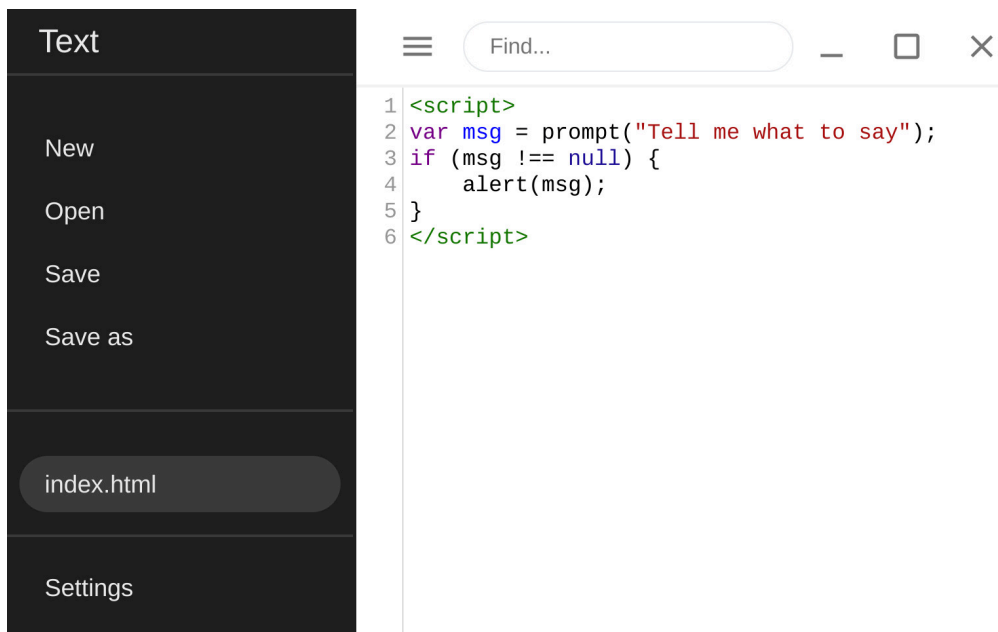


Figure 1.2: Text, a Chrome-based text editor, has highlighting and auto-indent, among other things.

Remember that you really **cannot** use a standard word processor for this. If you try, you will curse its sudden but inevitable betrayal.

If you start your editor and see a blank file, possibly named “Untitled”, that’s perfect.

Write and Save a File

We are now at the second step (in bold) in this process:

1. Start your text editor.
2. **Write some code into a file and save it as `firstprogram.html`.**
3. Load that file in the browser.

If all has gone according to plan, you are looking at your text editor and a blank file within it. Without worrying too much about what it means, put the following into your file:

:html:

```
<script>
a = 10;
b = 20;
c = a + b;
alert(c);
</script>
```

Once you have written that into your file, save it somewhere predictable with the name `firstprogram.html`.

Did it work? If it did, you can check by going to your file manager (the “Files” app in Chrome OS, for example) and seeing if a file called `firstprogram.html` shows up where you thought you put it. You might have to search for it (if you do, try to remember where it went, for next time). If you can find your file in the file manager, you did it! You now have a medium in which to practice your programming arts.

Note that we wrote some JavaScript code, but we did so *inside of HTML* code. HTML stands for “HyperText Markup Language”, and is the native language of web pages. This is how we will proceed in this course most of the time: it is simpler to learn how to program if we just put our JavaScript inside of HTML files in the beginning. This makes it simple for us to use the browser as a programming lab.

The important thing to know for now is that anything between `<script>` and `</script>` is interpreted by the browser to be JavaScript code, and it will be run when the page loads.

You will sometimes see a tag like this `<tagname/>` with the slash at the *end*, which just means that there wasn’t anything inside of the opening and closing tags, so they got smashed together. Approximately, anyway. Some tags do not allow this shorthand, however, and the `script` tag is one of those. So, even though you might sometimes see

things like `
`, you will never see `<script/>` because that just isn't allowed. It's too bad, really, because sometimes it would be nice.

One more thing: we have sneakily introduced the concept of a **function call** here by putting in the code `alert(c)`. We will get comfortable with that later. For now, just note that `alert` causes a little window to pop up with, in this case, the value of the variable `c` displayed. That's an easy way to see if our code is doing anything. A more complete explanation of function calls is coming very soon.

Run Your Program

We finally get to take the final step in bold below, and see our program work.

1. Start your text editor.
2. Write some code into a file and save it as `firstprogram.html`.
3. **Load that file in the browser.**

On most operating systems, opening something in the browser is pretty straightforward: you can go to the browser's `File > Open` menu and find your `firstprogram.html` file right where you saved it. In Chrome OS, you will need to find it in the "Files" app first (you can also press `Ctrl + O` to get started), then open it by double-clicking it or by dragging it onto an empty browser tab.

If you have succeeded, you will see a little alert window with the number 30 in it, and the location bar should show something that looks like a path to your `firstprogram.html` file, starting with the text `file:///`.

In Chrome OS, the real path to your file probably has a bunch of random characters in it, presumably to make it hard to predict in the interests of security. That's one reason we need to use the Files app to find the file; trying to type it in directly is usually not going to work. Other operating environments do this more straightforwardly.

Also, if you opted to run your own web server, as mentioned briefly earlier, you will go to a different place to find your code, typically a URL of the form

`http://127.0.0.1:8887/projectdir/firstprogram.html`

(details will differ, depending on how you set it up).

If you don't see a little alert window pop up with a number in it, either you didn't put the code into a file, didn't save it, or didn't successfully find and open it. If you don't have any ideas on how to fix your programs, you might try going back to the beginning and doing all of the steps again. Eventually this will seem familiar and easy, but at first it can feel like a lot of steps just to do one thing.

The good news is that once you have your program file set up like this, you can just change the file in your text editor, save it, and then **reload** the appropriate tab in your browser: it's

already pointing to your file, it just needs to read it again. If you refresh like that, after saving, you will see the changes. All of this file dragging is only needed at the beginning of a programming session. After that you will just spend time flipping between the editor and the browser, saving in the editor and reloading in the browser every time you change something.

If you are having trouble and your program isn't working, note that the JavaScript interpreter will just run your code from top to bottom. Because of that, a common technique is to add `alert` statements throughout the code to see which line makes them stop appearing. For example, take a look at this buggy program:

:html:

```
<script>
a = 10;
b = 20;
c = a + b;
Alert(c);
</script>
```

This won't ever alert because when it sees `Alert(c)`, that function name is spelled wrong (remember, names are case-sensitive). If you wanted to find out where the problem is, you could sprinkle some alerts in your code like this:

:html:

```
<script>
alert(1);
a = 10;
b = 20;
alert(2);
c = a + b;
alert(3);
Alert(c);
</script>
```

In this case, you'll see an alert saying "1", then after dismissing it, "2", then finally "3". But after you dismiss that one, nothing else will appear. This helps you to narrow down the problem to the one offending line. If you have developer tools, there are even easier ways to go about this, but this is still a tried and true approach to finding your own code problems.

Interpreters Are Loopy

If you have succeeded with the above, congratulations! You have officially begun. There is still a lot to learn about this JavaScript interpreter that the browser puts to work for us, though. What does it do, anyway?

The first thing to really get clear is this: your text editor is not running your code. It is merely letting you type it and save it somewhere. The browser does all of the work of interpreting and running the code you write.

The next important thing to understand is how the browser treats your code once it opens it: when it encounters a `script` section in HTML, everything in there is handed off to the JavaScript interpreter (which is part of the browser).

Loosely speaking, the JavaScript interpreter **reads** each statement from top to bottom, one at a time, and **evaluates** it in order. Sometimes the statement it's working on just sets things up for later use, and sometimes it does something right away. Right now we only have experience with stuff that happens right away, like this:

:html:

```
<script>
a = 10;
b = 20;
alert(a + b);
</script>
```

From top to bottom, the variable `a` is assigned a value (and created if it doesn't already exist), then the variable `b` is assigned, in that order. That's a rather important property to be able to rely on, since the last line `alert(a + b);` needs to use the previous results in order to do its work.

This last example is a little bit different than what we saw before: it contains an **expression** inside of a function call. It's still true that a full explanation of function calls lies ahead of us, but this brings us to another thing about the interpreter: it not only evaluates things from top to bottom, it also evaluates them from the inside out. Basically, it has to evaluate `a + b` before it can call `alert` with that value, so that's the order it works in.

If you, as a human, were to evaluate a statement $f(x + y)$, you would do the same thing: find the value of $x + y$, then evaluate f with the result; inside out.

What the interpreter is doing is called a “Read-Eval Loop”. It reads, it evaluates, and then does it again on the next statement. It's kind of like shampoo bottle instructions: “lather, rinse, repeat”, or “read, eval, loop”. It's the same, really, with less hygiene and more internet.

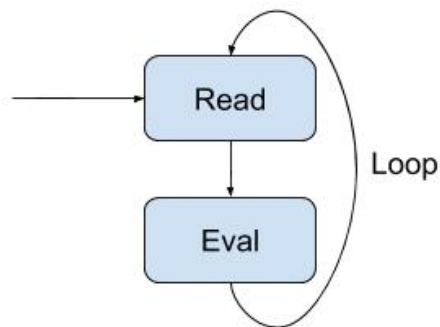


Figure 1.3: The Read-Eval Loop

When you run a file like this, all of that happens *silently*, without showing you anything that it is doing, unless you explicitly tell it to with something like `alert`. When you tell the browser to load your file, the interpreter processes it as fast as it possibly can, without your help.

The console is a bit different than regular JavaScript execution. It has its own special interpreter loop, called a “Read-Eval-*Print* Loop” (commonly but unimaginatively called a “REPL”, and often pronounced “repple”). When you type something into the console, it not only reads and evaluates it, it also prints the result without you asking for it to do so, like this:

:console:

```
> 10 + 20  
< 30
```

It reads `10 + 20`, then evaluates it, then **prints** that value (`30`): read-eval-print. Then, if you type something on the next line, it does it all over again.

That difference is why loading our files into the browser doesn’t show us anything unless we do something like call `alert`: the normal read-eval loop has no “print” stage. That stage is unique to the console, where every single thing we do is right there in our face.

More JavaScript Syntax

We have so far used the JavaScript console as a calculator, and even created a small program that did some computations using variables and showed an alert with the final result. All of the operands and results in those expressions have been numbers. Numbers are nice and all, but we like our programs to be able to communicate with people, so text is also going to be pretty important. There are two different kinds of text in JavaScript programs, and we will talk about both of them here: comments and strings.

But first, an important message from your REPL.

A Semicolonoscopy

A quick note is in order about semicolons, a topic that is almost certainly already your favorite.

The simple version is this: in the same way that you would end a sentence with a period, **end all of your statements and expressions with a semicolon**. An example of a program with semicolons is here:

:javascript:

```
a = 10;  
b = a + 5;  
alert(b);
```

See how every expression is terminated with a semicolon? Just do that and you’ll be fine.

When we work within the console, we can only ever enter a single expression at a time, where an **expression**, which we haven't really formally defined before, is "something that can produce a value" or "something that can be *evaluated*". Once we hit `Enter` in the *console*, our expression is complete and the interpreter tries to evaluate it immediately. When we are writing a program in a file, however, we are giving the interpreter a whole bunch of stuff to do, and some of our expressions will be complex and span more than one line.

The general rule, which we will revisit and revise to be more accurate as we go, is this: **always terminate your expressions with a semicolon**. JavaScript tries to be kind here and inserts invisible semicolons where it thinks you meant to put them, but it **often fails to insert enough of them**, and that can lead to all sorts of confusion. Therefore, in all of the code examples that are in a file, you will see semicolons at the ends of expressions. Follow that example and you'll be far better off.

In the console, we leave them out where we can because the console evaluates things immediately when we press `Enter`, avoiding this issue. The console is special, and again, we won't be focusing on it for the entire course, just the beginning when we are playing around.

The reason a lot of this matters is that JavaScript has some misfeatures, things often referred to as "warts" in programming language parlance. One of those warts is the way that it handles automatic semicolon insertion. When first programming, you won't run into situations very often where it matters. Simple stuff is unambiguous enough that JavaScript gets it right, there are few semicolons, and there is much rejoicing.

But, once your code starts getting more advanced, you start running into this problem all the time, and by then you might have developed an unfortunate bad habit or two. Thus, even though you might not notice the difference when you leave them off right now, always use semicolons anyway. You'll thank me later.

Comments

Sometimes you want to leave notes behind for your future self, or for other people who might look at your code and want to understand it. These notes are things that you don't expect the computer to care about; you want to leave behind some hints for *humans*. After all, code is often easy to follow while you are in the process of writing it, but can become confusing after some time has passed and the full picture of what you were doing has long since fallen out of your brain. Notes like these are called "comments", and knowing when and how much to write in them is both an art and a skill; you will get better at it by finding good examples to emulate, and by practicing.

JavaScript has two forms of comments, **line comments** and **block comments**. They look like this:

:javascript:

```
// Line comments stop here
// Always going to the end
// Never further on.

/* Block comments go on
   Without regard for endings
   Except for this one: */
```

If you were to type those into the console, nothing would happen.

It might tell you that the expressions have the value `undefined`, which is confusing but technically true—comments can be thought of as do-nothing expressions that evaluate to nothing, and “nothing” in JavaScript is often spelled `undefined`.

Frequently you will see people format their block comments like this:

:javascript:

```
/* Block comments are seen
 * With some leading asterisks
 * To make them tidy.
 */
```

The important thing to remember about comments is this: the JavaScript interpreter ignores them. They don’t do anything, they just sit there in your code, ready for a human to read later when trying to fix the bugs you left behind. The interpreter just skips right over them and continues on its way.

This is actually a little bit of a lie, though it is true enough for our purposes. JavaScript actually lets you get at all kinds of things, including the actual text of your functions (and therefore your comments within them), from other parts of your program. That means some programs have comments that the computer cares about. But you shouldn’t worry about that; those circumstances are extremely rare and usually either ill advised or carefully documented.

We will have comments in the example code throughout this course, so you will get familiar with them in a hurry.

Strings

Programming is not much fun if our programs can’t communicate with people. That means processing text that, unlike comments, is used by the interpreter. Bits of text are called “strings” because they are a representation of an ordered “string of characters”.

As is often the case with computer science ideas, particularly where they intersect with human endeavors like language, the concept of a “character” is surprisingly deep and interesting,

much more so than even many seasoned programmers realize. For our purposes, though, it will be sufficient to describe a string of characters as “what you are reading right now.” The ideas on this page are expressed one sentence at a time, each sentence being formed from words and punctuation, each word, space, and punctuation mark being formed from **characters**. These include letters, numbers, spaces, and all kinds of symbols in all kinds of languages, even emoji.

To express strings in JavaScript, we **delimit** them on either side with quotation marks, either single or double; it doesn’t matter so long as we use the same **delimiter** on both ends. For example, a program that sets the variable `s` to hold the string “Hello” might look like this, using double quotes to delimit the string:

:javascript:

```
s = "Hello";
```

As mentioned, you can also use single quotes, like this:

:javascript:

```
s = 'Hello';
```

Use whatever you like, as long as it matches on both sides.

To “delimit” something is to “set it apart from” other things. Thus, the double and single quote characters are called “string delimiters”; they set the string contents apart from the rest of the code.

Strings can be a little tricky to understand at first, because they are always surrounded by quote marks, but those quote marks are not part of the string. Rather, they are a signal to the interpreter that it should treat the characters between quote marks as a string. To hopefully make this extra clear, let’s talk for just a moment about what the interpreter does when it sees the very short program above. First, it sees that we are assigning something to `s`, so it then tries to evaluate `'Hello'`;

Looking at `'Hello'` from left to right, what does it see first? It sees a single quote `'`. That signals to it that “what follows is a string”. It also tells it that “the string ends when you find another single quote.” Therefore, it starts reading, one character at a time, keeping track of the string’s contents, until it finds a quote just like the first one, then it knows it has the full string. In this case, that string consists of the characters ‘H’, ‘e’, ‘l’, ‘l’, and ‘o’, in that order. The quotes just tell it when to start making a string, and when to stop.

If you fire up the console, you can play with strings there. Let’s see how they look in that tool.

:console:

```
> alert("Hello");
```

If you run that in the console, you will see a pop-up window with `Hello` written in it. Again,

the quotes are not part of the string, they just delimit the string.

Since the interpreter reads characters one at a time until it finds the ending quote mark, how can we include a literal quote mark as part of the string itself? After all, a quote mark is a character. What if we just try the following?

:console:

```
> // This won't work:  
> alert("this " is a double quote");
```

That won't work. The interpreter sees the first `"`, then starts reading characters one at a time, looking for another `"` to tell it when to quit. It sees that second quote mark right after the word `"this"`, and then it thinks it is done with the string. That's a problem, because there's more coming.

One way to handle this is to use single quotes for delimiters around strings that contain double quotes, like this:

:console:

```
> // This works.  
> alert('this " is a double quote');
```

The interpreter sees that first single quote, then starts reading characters until it finds another single quote, which it only finds at the very end, right where we want it to stop. That works! The same thing works in reverse, as well: strings containing single quotes can be delimited with double quotes.

Now, what if you can't do that? What if you want both kinds of quotes inside of your string? To do that you need a special character that signals to the interpreter, "Pay attention, because the next thing you see is not a delimiter, it's part of the string." We call that an **escape character**, and in JavaScript strings it is the backslash character `\`. You can use it in your program like this:

:console:

```
> alert("Sometimes you \"want\" to have 'both' kinds.");
```

Shiny! You can tell the interpreter that you want to include a quote mark in your string by typing `\"` or `\'` instead of just the bare quotation marks. But what if you want to include a backslash character? How would you do that? As it happens, the backslash is its own escape character. Check out this console statement:

:console:

```
> alert("Escape \" ALL \' the \\ things!");
```

If you were to type this into the console, it would display `Escape " ALL ' the \ things!` in a window. See how the first backslash is not there? It signaled to the interpreter

that the next character is literally to be part of the string, no matter how special it normally is, and that next character was a backslash.

By the way, for the love of all that is right in the world, please know the difference between the forward **slash** / and **backslash** \ characters. One leans forward (for languages that read left-to-right, anyway, which is how most programming languages also work), the other leans back. The forward slash / is often just called “slash”. It is *never* appropriate to call it “backslash”. That distinction is reserved entirely for the \ character. We’re in programming land, now, where precise communication is crucial.

This public service announcement is a backslash backlash. Know your slashes, avoid backlashes.

The rule for string escaping is this: read characters from left to right. Any time you see a backslash \, it is the beginning of an **escape sequence**, a sequence of characters that means something special to the interpreter. Escape sequences start with \ and can contain one or more things afterward. You have just been introduced to escaped **character literals**, where the backslash precedes exactly one “normal” character: the thing you really want to express but can’t because of string rules. There are other escapes¹, as well, but we’re going to ignore them for now.

A **literal** is something that you type into the interpreter that is just itself. Something like 10 is a literal, as is "hello". They are literally what they appear to be, not part of the language syntax: just values. The first is the number 10 and the second is the string "hello".

Strings are pretty useful. We will be seeing them a lot. In fact, we will be seeing them a lot more than we want to, because JavaScript has a habit of trying to turn everything, including numbers, into strings (note: it also tries to do the opposite when it thinks that’s warranted). That can make for some interesting and unfortunate accidents if we’re not careful, because strings can be added to each other:

:console:

```
> "Hello, " + "world!"  
< "Hello, world!"
```

See how we just added two strings together to make a new one? The addition operator, when it has a string on the left, **concatenates** it with the string on the right. It jams them together to make a new string.

Here’s a tidbit that we will run into later (so feel free to appreciate it and then forget it). Check this out:

:console:

¹https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String

```
> "5" + 10  
< "510"
```

Oops. We have a string literal on the left and a numeric literal on the right (no quote marks around the right-hand number). JavaScript silently converts the number `10` into a string `"10"` for us and then does string concatenation. This particular example is silly, but sometimes you don't know whether the thing on the left is a string: for example, if it's a value within a variable. It's therefore relatively common practice to use the unary `+` operator to ensure that strings are converted to numbers before computation is done on them, like this:

:console:

```
> a = "5"  
< "5"  
> a + 10  
< "510"  
> +a + 10  
< 15
```

That's better. The unary `+` operator is not seen much in mathematics because it is sort of redundant, but it exists in a lot of programming languages. It's like adding the number to 0, just like the unary `-` is like subtracting the number from 0. In this case, it unambiguously instructs the interpreter that the next thing should be a number, even if it starts out as a string.

Putting It All Together

With that, we can now create what is arguably the most important program that anyone ever writes. Open your editor, type the following into it, save it as an HTML file of some kind (ending in `.html`), and open it in your browser:

:html:

```
<script>  
alert("Hello, World!");  
</script>
```

Congratulations! You are now officially an amateur programmer. There is still much to learn, but you are armed with a possibly useful textbook. Don't stop now!

Exercises

Exercise 1-1: Create a new programming session

Solution on page 363

Practice starting a new programming session from scratch. Close all of your program tabs, then

- Start your text editor.
- Write “Testing my sanity” or some other recognizable and pithy comment, and save it as an HTML file.
- Load that file in the browser.

Exercise 1-2: Practice loading the console

Solution on page [364](#)

Once you have successfully loaded your new program (with your new statement inside of it), open the Developer Tools and find the console.

Exercise 1-3: Find the source viewer

Solution on page [364](#)

When you write the text of a program, you are writing what is called “source code”. This means your code is the “source” of all of the instructions that the computer will execute on your behalf. Find the developer tool that lets you view the source of the file that the browser has currently loaded.

Note that this is not the “View Source” menu item that you might be familiar with. Rather, this is the *developer tool* that shows you the source and gives you tools to work with it.

Exercise 1-4: Faster developer tools access

Solution on page [365](#)

Developer tools are incredibly important and useful, particularly for this course, so we are going to want to be able to get to them quickly. Find the combination of key strokes that opens (and closes) the console and practice them. Memorize them.

Exercise 1-5: The console as a calculator

Solution on page [365](#)

Use the console to find the value of this expression: $1 + 2 - 3 + 4 - 5 + 6 - 7 + 8 - 9 + 10$.

Exercise 1-6: Definitions

Solution on page [366](#)

Define the following:

- syntax highlighting
- JavaScript console
- text editor
- variable
- expression
- evaluate

- assign
- delimit
- escape character
- literal
- concatenate

Exercise 1-7: Assignment practice

Solution on page [367](#)

Show code that causes the variable `x` to contain the value 42.

Exercise 1-8: Evaluation

Solution on page [367](#)

Write the result of evaluating $(x - 12) * 3 + 9$ where x is 42 without using a computer - use your brain to figure out what the answer will be. Show your work.

Exercise 1-9: Using variables

Solution on page [367](#)

Use a variable to compute the value of the polynomial $x^2 - 2x + 1$ where $x = 3$. Hint: you can use `x*x` to mean “x squared”.

Exercise 1-10: Approximate Euler’s constant

Solution on page [368](#)

If you are using Chrome or Firefox (as they support exponentiation using the `**` operator), approximate the value of e by computing $(1 + 1/1000)^{1000}$. What value do you get?

Exercise 1-11: A non-console program

Solution on page [368](#)

Write a program (in your text editor, not the console) that uses string concatenation to show “Hello!” using only strings that each contain a single character.

Exercise 1-12: Escapes in strings

Solution on page [369](#)

Write a program to create a string that shows this emoticon `\o/` (backslash, ‘o’, slash) and alert it.

Chapter 2

Function Calls and the Debugger

So far we've had occasion to draw on one concept from algebra: **variables**. There is another concept from algebra that is absolutely critical for programming: **functions**. Functions are so central to computer science that, combined with what we know about variables, you could actually write complete programs using nothing else (thanks to Alonzo Church¹ and Alan Turing²). Functions are very important, and we can't do much in JavaScript without knowing about them.

This part is a bit more detailed than what we have done before, so be prepared to do some thinking. Also, we will take a break from programming for a bit to do some review, as this concept is so fundamental. Let's dive in.

Functions in Algebra

In algebra, you have probably seen function notation, like in this example:

$$f(x) = x^2 + 2x + 3.$$

You have probably even plotted something like it, which in this case gives you a parabola, as depicted below.

This might seem really familiar, but it might also be deeper than you think: how exactly did we get from $f(x)$ to that graph? A very typical approach is to just evaluate $f(x)$ at a bunch of values of x and plot those points, joining them afterward with curvy lines. That approach is certainly intuitive (and is basically what the computer did to make that picture), but what does it mean to “evaluate $f(x)$ ”?

¹https://en.wikipedia.org/wiki/Lambda_calculus

²https://en.wikipedia.org/wiki/Universal_Turing_machine

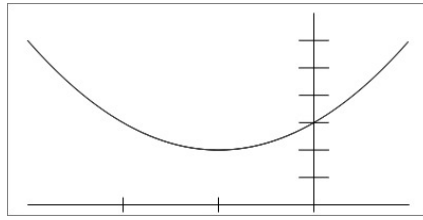


Figure 2.1: The function $f(x) = x^2 + 2x + 3$.

This is actually a very deep question, and it has a lot of answers depending on the context. This text is not the place to discuss most of them, but you should know that functions are anything but boring and simple: their nature and what they can express are still an active area of research.

This whole notion of “evaluating” something is central to mathematics and to programming. We have touched very briefly on it before, saying it is the process of getting something’s value. But how do we do that? As an example, we evaluate expressions like $3 - 2$ by simplifying them into a single result: 1. To **evaluate** is essentially to **produce a value**. If we have an expression like $x^2 + 1$ and we try to evaluate it, the first question is “What is the value of x ?” If we know that value, we can “plug it in” for x and simplify the whole expression to a number. That’s evaluation: plug and chug.

There are a couple of ways to view algebraic functions, so you should know that the explanation here is biased toward what we need to know to be successful at programming, not toward mathematical completeness. As long as we are comfortable with that kind of pragmatism, we can create a narrow but useful working definition of a function:

A **function** is a specification of

- **steps** to take
- to obtain a **value**
- given adequate **settings**
- for any **parameters**.

The **parameters** of a function are the variables mentioned within the parentheses, like x . Their full name is **formal parameters**, but we will refer to them as just “parameters”.

When viewed in this way, you can think of the function as a little machine. You put parameter settings in, and the function’s definition tells you how to get a value out. In our parabola example, $f(x)$ has a single parameter: x . When we set x to some known value, like in the expression $f(4)$, that value is called an **argument** to the function: the parameter x has been set to 4, so 4 is the argument corresponding to the parameter x . At that point all parameters have been specified (all necessary arguments are supplied) and we can get a value out of the function by evaluating its definition in terms of those arguments.

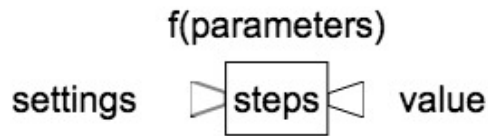
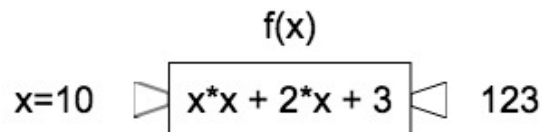


Figure 2.2: A basic function box.

That was a lot of words to describe what is actually a simple concept: stick stuff into the variables in f , do what its definition tells you, and you will get a value out. That's how you evaluate a function.

This summarizes the two contexts in which we see notation like $f(x)$:

- **Definition**, such as $f(x) = x^2 + 2x + 3$, and
- **Evaluation**, such as $f(4)$.

Figure 2.3: A function box for $f(x) = x^2 + 2x + 3$.

With a definition like $f(x) = x^2 + 2x + 3$, we can easily answer the question “What is $f(10)$?” If we try that, we are evaluating f at $x = 10$. If this feels natural already, don't overthink it, you've got the idea. We'll go into some additional detail below just in case it's a little fuzzy.

To find $f(10)$, we start with the definition, then do some substitution and simplification:

$$\begin{aligned}
 f(x) &= x^2 + 2x + 3 \\
 f(10) &= 10^2 + 2(10) + 3 \\
 &= 100 + 20 + 3 \\
 &= 123
 \end{aligned}$$

In other words, we

- **Expand** the function definition ($f(x) = x^2 + 2x + 3$), then
- **Substitute** parameters (x) with corresponding arguments (10), and
- **Evaluate** what remains ($10^2 + 2(10) + 3 = 123$).

With care taken to ensure that variables in different scopes are properly distinct from one another, this works for all functions: **expand**, **substitute**, and **evaluate**.

The annotated process is shown here:

$$\begin{aligned}
 f(x) &= x^2 + 2x + 3 \\
 f(10) &= \overbrace{10^2}^{x^2} + \overbrace{2(10)}^{2x} + 3 \\
 &= \overbrace{100}^{10^2} + \overbrace{20}^{2(10)} + 3 \\
 &= \overbrace{100+20+3}^{100+20+3} \\
 &= \overbrace{123}
 \end{aligned}$$

We will frequently use these kinds of annotations for evaluation (less so for substitution, since that's usually pretty straightforward), to describe what is going on with evaluations. They signal where each term comes from, hopefully allowing you to see more intuitively how each step is performed, without getting lost.

If we only show one step per line, we can use over- and under-brackets without annotations to show what is happening, like this:

$$\begin{aligned}
 f(x) &= x^2 + 2x + 3 \\
 f(10) &= \underbrace{10^2 + 2(10)}_{\overbrace{100}^{10^2} + \overbrace{20}^{2(10)}} + 3 \\
 &= \underbrace{100 + 20}_{\overbrace{100}^{10^2} + \overbrace{20}^{2(10)}} + 3 \\
 &= \underbrace{100 + 20}_{\overbrace{100}^{10^2} + \overbrace{20}^{2(10)}} + 3 \\
 &= \underbrace{120}_{\overbrace{100}^{10^2} + \overbrace{20}^{2(10)}} + 3 \\
 &= \overbrace{123}
 \end{aligned}$$

Do you see how the brace beneath one line matches with the brace above the next? We will use that when we want to do one step at a time in our evaluations, just like a computer does. Take a few moments to be sure that these annotations make sense before proceeding.

With that out of the way, we now know that $f(10) = 123$. That means that $f(10)$ *has a value*. It kind of *is* a value, really. You can replace $f(10)$ with 123 everywhere you see it.

If $f(10)$ is a value, what is $f(x)$? Because x is unknown, $f(x)$ is not a value, it's just instructions for producing a value once we do know x .

In other words we can think of functions as defining a process, a set of steps. They are like recipes. If you follow the instructions (expand the definition) with the right ingredients (arguments), you get to eat (find the value).

And, just like with cooking, progressing from a function or expression to a value requires work. We will be touching on that idea later on when we start writing some code.

Some expressions fundamentally require more work to evaluate than others, and it is an important field of study in computer science: the study of **complexity**, essentially how many resources are required (in both time and space) to get certain answers.

Let's sum up a bit.

- Functions like $f(x) = x - 1$ are definitions. They have variable parameters, not values, and they define a recipe for getting a value when we are fortunate enough to know the arguments.
- Function expressions written like $f(10)$ become values in their own right. You can do expansion, variable substitution, and evaluation to replace them with actual values.
- Evaluating functions requires work involving actual space, actual time, and actual energy, just like following a recipe.

Recursion

There is a really interesting and crucially important quirk of function definitions in algebra that can be a little bit mind-bending at first, so we are going to spend some time with it here. That concept is **recursion**. Recursion is basically a function that is partly defined in terms of itself.

Nested Function Expansion

Before we jump into that, though, let's remind ourselves about functions that are defined in terms of other functions:

$$\begin{aligned}f(x) &= x + g(2x) \\g(y) &= y - 1\end{aligned}$$

What if we want to evaluate $f(5)$? Let's go through the steps: expand, substitute, evaluate:

$$\begin{aligned}
 f(5) &= 5 + g(\underbrace{2 \cdot 5}_{10}) \\
 &= 5 + g(\underbrace{10}_{10}) \\
 &= 5 + \underbrace{(10 - 1)}_{9} \\
 &= \underbrace{5 + 9}_{14} \\
 &= 14
 \end{aligned}$$

In order to evaluate $f(5)$, we first had to pause to evaluate $g(10)$. So, we did that by expanding, substituting, and evaluating $g(10)$, then we could finish what was left in our evaluation of f 's expansion. This is a very important principle: in order to evaluate an expression, you must first evaluate any functions that it contains. Basically, you cannot finish evaluating an expression until you have evaluated all of its parts.

That is why evaluation is done inside out: the outside depends on the result of the inside, so the inside is computed first.

Recursive Evaluation

It turns out that you can do the same thing with the function f —it can appear in its own definition.

While that might sound a bit weird, it actually appears fairly early on in algebra. Let's talk a moment about the factorial operator for integers $k \geq 0$. Note that we will call it $F(k)$ instead of the usual $k!$ to make it a little easier to see the recursion:

$$F(k) = k(k-1)(k-2)\dots(1).$$

What does this mean? Well, if $k = 4$ then $F(k) = 4 \cdot 3 \cdot 2 \cdot 1$. If you squint at it, you can see that pattern in the formula above: multiply k by the value one less than it, then the next one down, and the next, all the way until you get to 1. What we have not yet said is that there is a special case for $k = 0$: $F(0) = 1$. This will help us in a moment. The ellipses (...) just mean "continue as you were already going" until you get to the end, where you have 1.

This is useful enough as it is, but let's see if we can make the definition a little easier to write down, and notice some patterns in it while we're at it. Look at $F(k)$ and $F(k-1)$ together for a moment:

$$\begin{aligned}
 F(k) &= k(k-1)(k-2)(k-3)\dots(1) \\
 F(k-1) &= (k-1)(k-2)(k-3)\dots(1)
 \end{aligned}$$

To make it concrete, have a look at what happens if $k = 5$ and $k = 4$:

$$F(5) = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$

$$F(4) = 4 \cdot 3 \cdot 2 \cdot 1$$

Well, that's interesting! It looks like $F(4)$ gets us most of the way to $F(5)$. Taking advantage of that gives us $F(5) = 5 \cdot F(4)$. That seems like something we can use, but to really be useful, it needs to be written in terms of not just 4 and 5, but an arbitrary integer k :

$$F(k) = \begin{cases} 1 & \text{for } k = 0 \\ kF(k-1) & \text{for } k > 0. \end{cases}$$

Note that in the above notation there are two ways to compute $F(k)$: if $k = 0$, then the factorial $F(k)$ is defined to simply be 1. If, however, $k > 0$, then you have to do a little more work, because if $k > 0$, then $F(k) = kF(k-1)$.

This is a **recurrence relation**: it defines how a sequence can be built using recursion. So, how does it work? How do we use it in a practical way, for example, to evaluate $F(3)$? This particular one is simple, so looking at it, you would probably just turn it into a sequence of multiplication: $F(3) = 3 \cdot 2 \cdot 1$, which is 6. But if it were longer, you would need a more principled strategy.

The complete strategy is to expand and substitute, then evaluate, one step at a time. Exactly as when we introduced g into the definition for f above, our evaluation will not be complete right away: it will include another function evaluation in it. Let's take a look at the process, starting with our two definitions, then moving on to evaluation of $F(3)$:

$$F(0) = 1$$

$$F(k) = kF(k-1)$$

$$\begin{aligned}
 F(3) &= 3 \underbrace{F(2)} \\
 &= 3 \underbrace{(2 \underbrace{F(1)})} \\
 &= 3 \underbrace{(2 (1 \underbrace{F(0)}))} \\
 &= 3 \underbrace{(2 (1 \hat{1}))} \\
 &= 3 \underbrace{(2 \hat{1})} \\
 &= \underbrace{3 \hat{2}} \\
 &= \hat{6}
 \end{aligned}$$

We start with our definition, which now takes up two lines because we have the **base case** for 0 and every other case for values larger than 0. Then we substitute $k = 3$ in our expansion and do *everything we can*. It's okay that we can't evaluate the whole thing, yet. Our goal is just to make progress.

That leaves us with another function to evaluate, so we do that over and over again until all we have left are values. Note that eventually we evaluate $F(0)$, which we have defined to be 1, so we finally get a function expansion that does not result in another function. After that it is just finishing all of the partially-completed computations that we left behind, and we arrive at the number 6.

If you look at the pattern carefully, you can see a sort of diving and surfacing (or expanding and contracting) effect. We continue diving into evaluation until there are no more functions in our expansion, and then we start to surface by completing each computation that was paused while we did those expansions, evaluating the innermost expression first (just like we would normally do with bracketed expressions).

Many, many things in mathematics and computer science are defined in terms of themselves like this. A function that is defined in terms of itself is often referred to as a **recursive function** (because it “recurs”). As mentioned previously, you can define computation itself in terms of variables and functions, and recursion is a big part of what makes that possible. You can bet that this will come up again, and when it does, you will be ready to understand it.

For some reason, many programmers seem to think that evaluating a function inside its own definition means it “recurses”. But it never *was* cursing, and it certainly isn't doing it again. What it *is* doing is **recurring**. The function *recurs*, it does **not** “recurse”.

Bring this up, hold your ground, and people might even recurse at you!

Calling Functions in JavaScript

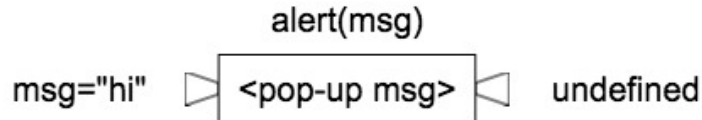


Figure 2.4: The alert function.

When we evaluate a function in a programming language like JavaScript, we say that we **call** that function with certain arguments. We treat it something like a robotic courier: we *call* it, have an *argument* with it about what kind of pizza we want, and expect it to *return* (hopefully quickly) to us without any errant toppings.

Our first encounter with this idea was the `alert` function, which is different than familiar algebraic functions because it only does work (displays its argument), and doesn't produce a value (output is `undefined`). Nevertheless, we call it with a message argument, and it does work before returning control (but not a value in this case) to the caller.

Let's start a brand new program file so we can play around with this concept more concretely. It's time to get out the computer and do stuff!

This is still new, so here's a quick reminder of what to do to write a new program:

1. Open your text editor,
2. Write some code and save it as `prompt.html` or something like that (names are hard, enjoy!).
3. Load that file in your browser.

If you have forgotten how to do any of those steps, go back to the first chapter and refresh your memory with the details there. We will be doing this for every new program.

Do you have your text editor open? Good. Let's put a familiar-looking program into it:

:html:

```
<script>
a = 10;
b = 20;
alert(a + b);
</script>
```

Now, this program does not do anything new at all, but we are about to change that. First, let's remind ourselves of what the interpreter does:

- Reads the next statement,
- Evaluates it, and
- Repeats that process until the end of the program.

In this program that means it displays 30 in a popup window.



Figure 2.5: The alert window showing the value of a+b.

Let's get a bit more interesting with functions that actually return a value, now.

Functions That Produce Values

What if, instead of just displaying stuff we typed into a program earlier, we want to ask the user for input that we can use in our program? Well, it turns out that there is also a function called `prompt`³. It accepts a message, shows that message in a pop-up window with an empty text entry field, and it returns whatever you type in that field. Let's look at some simple code using `prompt`. Try replacing your code with this:

```
<script>
msg = prompt("Hello! Type something exciting.");
alert(msg);
</script>
```

:html:

To be completely honest, the `prompt` and `alert` functions are probably not the best examples we could have started with. What would have been a bit more comfortable, especially coming from a pre-algebra background, would be functions that just compute

³<https://developer.mozilla.org/en-US/docs/Web/API/Window/prompt>

something and return the value of the computation. You know, functions like the square root, which does what it sounds like: takes a value and returns its square root. So why start with things like `prompt` and `alert`, which interact with the real world instead of doing simple computations?

First, programs are pretty hard to understand if they aren't doing something with the world. We need insight into what is happening, and `alert` is really handy for that. Second, it's good to get started off with the understanding that functions are not just a *description of how to combine values to make more values*, but are also *things that do useful work*.

Truthfully, there is a third—purely practical—reason: JavaScript doesn't have many interesting functions that are not tied to an *object*, and we haven't talked about those, yet. Foreshadowing is fun, but so is understanding what you are reading.

Let's pick this apart so there is no chance of confusion. What is happening here?

That first statement actually has a lot going on! The interpreter, in its read-eval loop, reads the statement and tries to evaluate it. It does so left-to-right, inside-out. You can imagine it carrying out an internal dialog like this one:

- Hey, that looks like a variable assignment. OK.
- There is no `msg` variable, so I will create one. Done.
- Now, what are we assigning it? Hmm, that is a function call.
- I cannot assign anything until that thing returns. Call `prompt` with that string.
- (prompt shows a window and waits for you to finish—everything pauses until you click a button)
- Yup, that finished, and it gave me something. Assign `msg` to be what the user typed into the window.

That's just the first line! The next line is more familiar: it just alerts the thing you typed in.

If you load this program, it will immediately ask you for input. Once you are done, it will display what you typed.

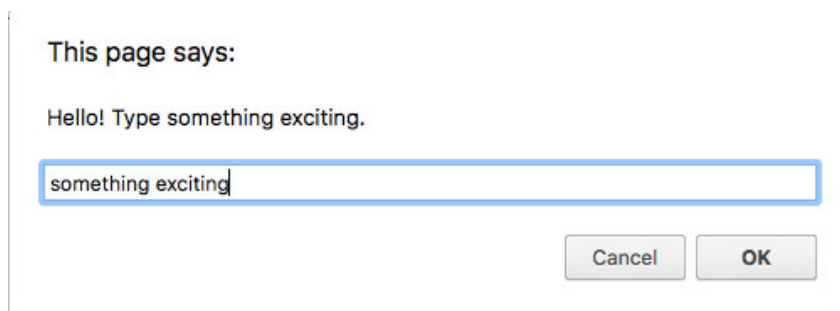


Figure 2.6: The prompt asking for something exciting, and our cheekily compliant response.

What happens if you hit “Cancel” instead of “OK”? That displays `null`. The `null` value is not a number, or a string, or a function, but a sort of featureless empty thing. Unlike

`undefined`, it is more sensible to treat it as an actual value. In the case of `prompt`, that value can be interpreted to mean “Cancel was pressed”. Otherwise you get a string message.

If you look back at the algebraic discussion of functions, you will see that `prompt` is acting pretty much like you might expect. You give it an **argument** (the message to display), and it **returns** a value (what you type in the pop-up window) accordingly. This is a bit different from algebraic functions in that it relies on what is happening in the world to produce its value, but the fundamentals are otherwise the same. Calling a function always involves doing work; in this case, that work is waiting for you to type something and then turning your keystrokes into a string of characters. The `alert` function similarly does work in the real world: it pops up a window that shows its argument, and waits for you to dismiss it before the code continues from that point.

A Debugging Interlude

What is “debugging”, and why do we call it that? Because legend has it that Grace Hopper⁴ fixed a computer once by removing a moth from one of its relays. Ever since then our program errors are called “bugs” and fixing them is the process of “debugging”. Grace Hopper was awesome, by the way, and a very important pioneer in computer science. It is well worth your time to learn about her life and contributions.

When we run our programs, we are just waiting to see what they do. We do not have a lot of insight into what is happening under the hood, as it were, unless we use our brain as a sort of biological JavaScript interpreter to predict what is happening. That is a very valuable thing for us to do and to get practice with, but the computer can also help us to see what is going on, step by step. To do that, we use the **debugger**, a tool that grants us a view of the inner workings of our programs.

Note that, if you are using a locked-down student laptop, or trying to go through this course on a mobile device, you may not have access to the debugger, or even the console. It is definitely possible to get through the course without it, but you will need to do some reading without hands on keyboard in a few places, including below. A debugger is a very useful tool for stepping through code and understanding what the computer is doing, but it isn’t required. Find another way of accessing a computer with proper tools if you can, but forge ahead and just learn about them otherwise; debuggers are helpful but optional.

If you have no access to the debugger, skim from here until the exercises, and skip those that pertain to the debugger.

Debugging Our Programs

Let’s go back to our little program from the previous chapter. Remember that one?

⁴https://en.wikipedia.org/wiki/Grace_Hopper

:html:

```
<script>
a = 10;
b = 20;
c = a + b;
alert(c);
</script>
```

At the risk of being a bit repetitive, let's talk about what this does, one line at a time. The read-eval loop, after all, is a core concept, so it won't hurt us to see it again:

- Assign 10 to the variable `a`.
- Assign 20 to the variable `b`.
- Assign 30 to the variable `c` by first evaluating and adding `a` and `b` together.
- Alert the value of `c`.

The first three steps are invisible. When you load this file in the browser, you just see the alert box with the number 30 displayed inside of it. Otherwise, all of the computations beforehand are not shown anywhere. Let's see if we can peek inside and see a little more of what happened.

Peeking Inside

Our program created and assigned three variables: `a`, `b`, and `c`. That means we should be able to see them in the console after our program is finished (they stick around). Try it!

Our variables are in what we call the **global scope**, meaning they are visible from *anywhere in our program*. We will come back to this, but one of the features of the global scope is that you can also see those variables in the console.

In the same tab where you loaded your program, dismiss the alert if you haven't already, and open the console. Type `a` to evaluate it. What do you see?

Note: if you really don't have access to the console at all. For example, if you are *unable* to use the console or the debugger, maybe because your account is limited or you are using a mobile device, you can just `alert` things when you want to see what they are. For example, in your program above, you could add `alert('a=' + a + '\n');` to see the value of `a` in an alert window. Making the alert from adding a string to the value helps you to see what you are looking at. Also, adding `'\n'` to the end of it allows you to alert multiple things on separate lines, like this:

:javascript:

```
alert('a=' + a + '\n' +
      'b=' + b + '\n' +
      'c=' + c + '\n');
```

That `'\n'` addition sends you to the next line (it stands for “newline”). You are really outputting a single string, it just happens to have control characters in it that break it up across lines and make it easier to see what is happening. Again, if you have no console, this can be an effective replacement for `console.log` and other console tricks used to see what is happening.

Let’s try evaluating each of our variables in the console:

:console:

```
> a
< 10
> b
< 20
> c
< 30
```

If that is what you get, you did it! Your program loaded, and the three variables were set just the way you would expect. Congratulations!

The console, when used like this, can be a powerful debugging tool. It can help us to see what our program has done, even though we have not necessarily gone out of our way to display anything. If you find that you are flying blind and are not sure what’s happening, try assigning a special variable and looking at its value in the console. Often things become clear after that.

The Debugger Tool

The console is just one of several different tools you can use to help with programming. We will be using it quite a bit, but mostly for viewing results, not quite as much for executing code like we do when using it as a calculator. When you have the console open, there are several tabs at the top of that area of your browser that have other tools. In Chrome they are listed as things like “Sources” and “Network”.

In Firefox you can find the debugger under a symbol that says “JavaScript Debugger” when you hover over it with the mouse pointer. Much of what we discuss for Chrome will apply to Firefox with some minor modifications that should be pretty clear if you are looking at the tools. They are all called basically the same things.

With our JavaScript code in an actual file, we are able to consider using **breakpoints**, **steps**, and **watches** to see what our code is doing. These are powerful debugging techniques that we are going to cover briefly right now. In Chrome, this is found under the “Sources” tab in the developer tools, which is why we needed to write our code into a file. Without that, we would not have any sources to look at in the first place.

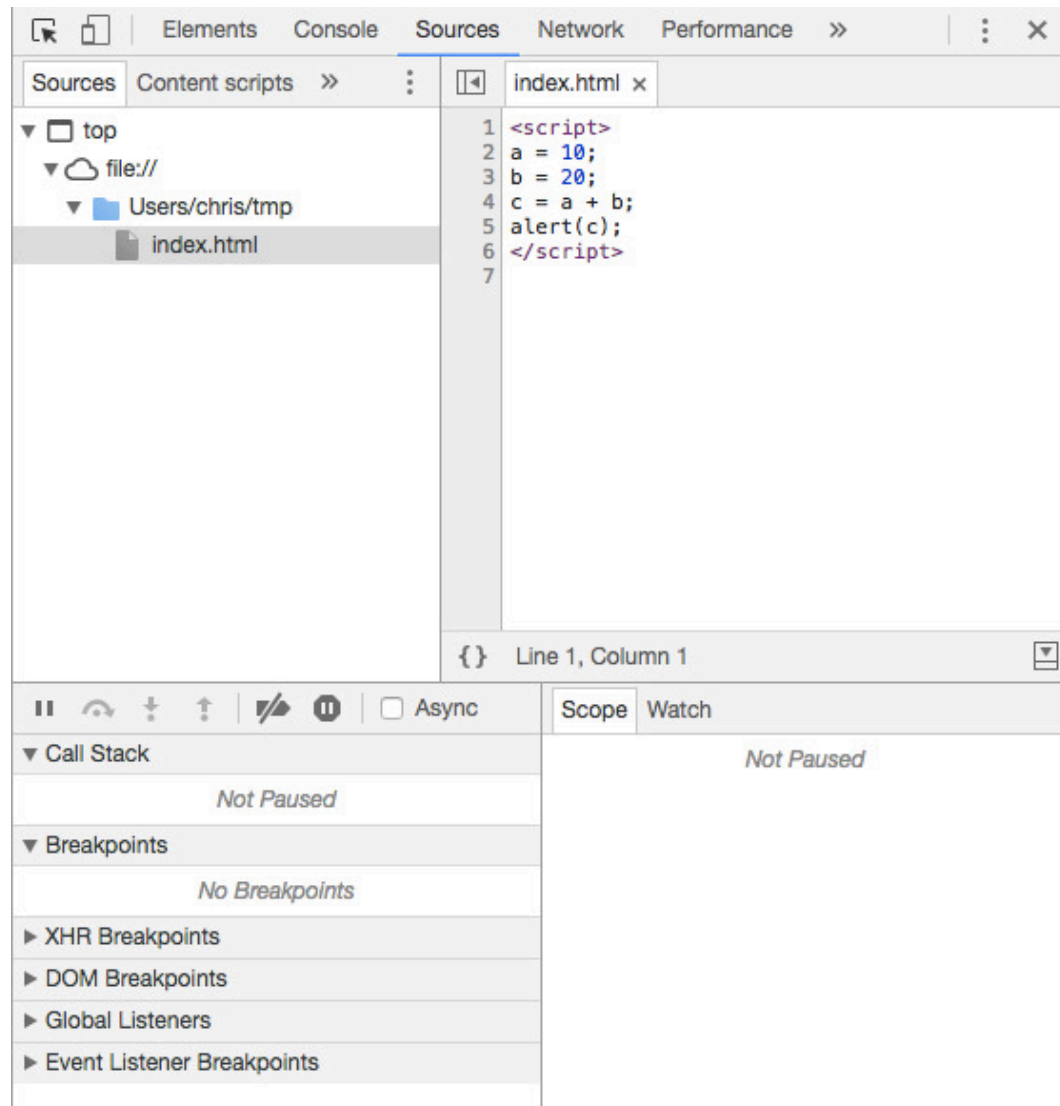


Figure 2.7: Chrome's Developer Tools with the Sources tab showing, and `index.html` selected. Your layout might look a bit different.

Find Your Code

Here are the basic steps for seeing your code in the debugger. This is the first thing we have to do before we can use the tools; we need to see the code in the right place. These steps will generally work with any set of developer tools, but details will be given for both Chrome and Firefox. In general, to get to where you can see the code you want to debug, do this:

1. Load the program in the browser.
2. Open the debugger in developer tools.
3. Select your program file.

For Chrome, the above steps look specifically like this:

1. Load the program (using Files or your file manager on a non-Chrome-OS machine).
2. Open the “Sources” tab in the developer tools.
3. Select your program file from the listing on the left.

You should now see your code in a tab named after your program (like `firstprogram.html`).

For Firefox, the above steps look specifically like this:

1. Load the program
2. Open the developer tools and find the tab that says “Debugger” (hover if you only see icons). Go there.
3. If, under “Sources”, your file is not listed, *reload the page* with the developer tools open. It should show up then.
4. Select your file. Note that if it is already selected, but you do not see your code, try tapping it again.

You should now see your code in the developer tools window.

Break, Watch, and Step

If you reload the page with your code in it, it will run really fast and you won’t see what is going on. What we need to do next is instruct the interpreter to stop time while we take a peek at it. We do this by setting a **breakpoint** in our code.

Next to your code, there should be a bunch of line numbers, starting with “1”. If you click on a line number, it will get highlighted with a pointer, indicating that execution will pause there and await your instructions. Find the line where `a` is first assigned, and click on its line number. It should be highlighted. If you click it again, it will stop being highlighted—that clears the breakpoint.

Now, we *reload the page* with that breakpoint turned on. When we do, we will start seeing some new things. Somewhere you should see something that says “Watch” (in Chrome) or “Add watch expression” (in Firefox). A “watch expression” is something that you want to see

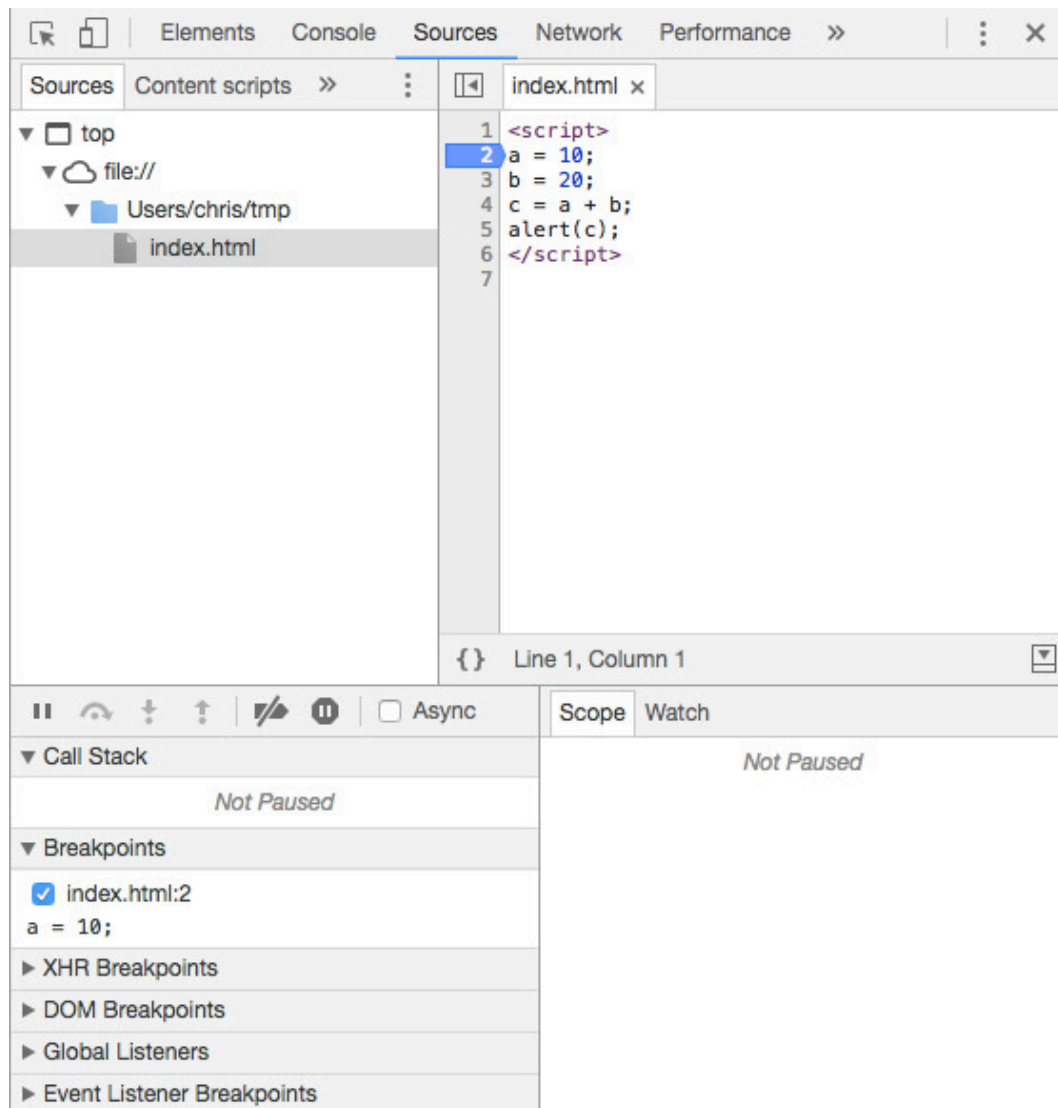


Figure 2.8: Chrome Debugger with a breakpoint on line 2.

the result of. In our case, we just want to see the values stored in our variables, so we will add `a` as a watch expression, `b` as another expression to watch, and finally `c` as the last thing we want to look at.

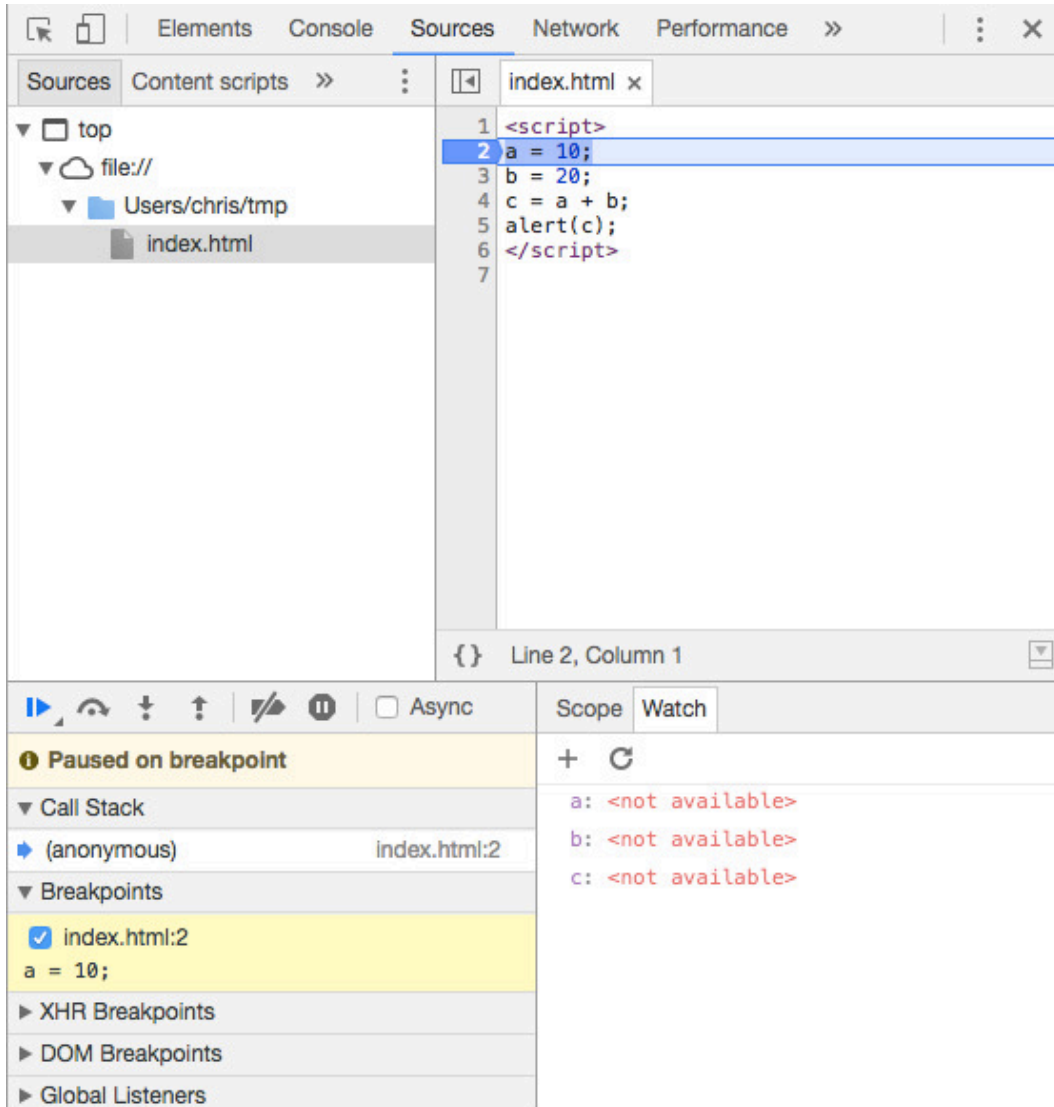


Figure 2.9: Chrome Debugger stopped on a line 2 breakpoint, with three variables added to the Watch section

In Chrome, this is all done from the little “Watch” section (you might need to click on it first), which has a big plus sign you can click to add expressions. In Firefox it is much the same: just “Add watch expression” as many times as you want, with different variables.

When we get into functions, we will actually be able to watch variables without saying so first. The debugger doesn't show us all of the global variables by default because there are a ton of them, so we have to tell it which ones we want, like `a`, `b`, and `c`. When we get into debugging functions, their **local variables** will show up automatically.

At first, the variables will not contain anything, and may say they are unavailable. That's fine: they don't contain anything because execution is paused before anything has actually happened. None of our code has run yet—it's stuck on a breakpoint. What we want to do now is **step** through our code and see what happens to the variables we are watching.

The symbol we want to use for stepping through code looks like an arrow jumping over a dot. It is called "Step Over" and it means "step over this line". Try pressing it.

What happened? You should have seen the active line of code move down one, and the watch expression for `a` change to contain `10`. Well, that's cool, that's exactly what we wanted to see, since we just assigned `a` to be `10`. Let's do it again!

The next step sets `b` to `20`, and that should show up in our watch list if we have added `b` as a watch expression.

When we execute the final line, it adds `a` to `b`, so we should see that `c` gets `30`. Take that last step and make sure that happened.

There's one other button that is useful here: the little play symbol (to the left of the step button in the Chrome figures). Pressing that button causes the program to continue until the end, getting us out of stepping mode.

There is actually a lot more that you can do with debugging, but this is as far as we take it in this course. The debugger is a great way to get a feel for what is happening in your code by watching how the interpreter handles it one step at a time, and can sometimes be used to find out what is going wrong when something unexpected happens. Sometimes we just want to log things to the console, or to a global variable, but other times we might want to step through our code to get a good mental model of what is happening. The debugger can be a useful way to do that.

Exercises

Exercise 2-1: Define terms

Solution on page 370

Define the terms

- formal parameter
- argument
- substitute

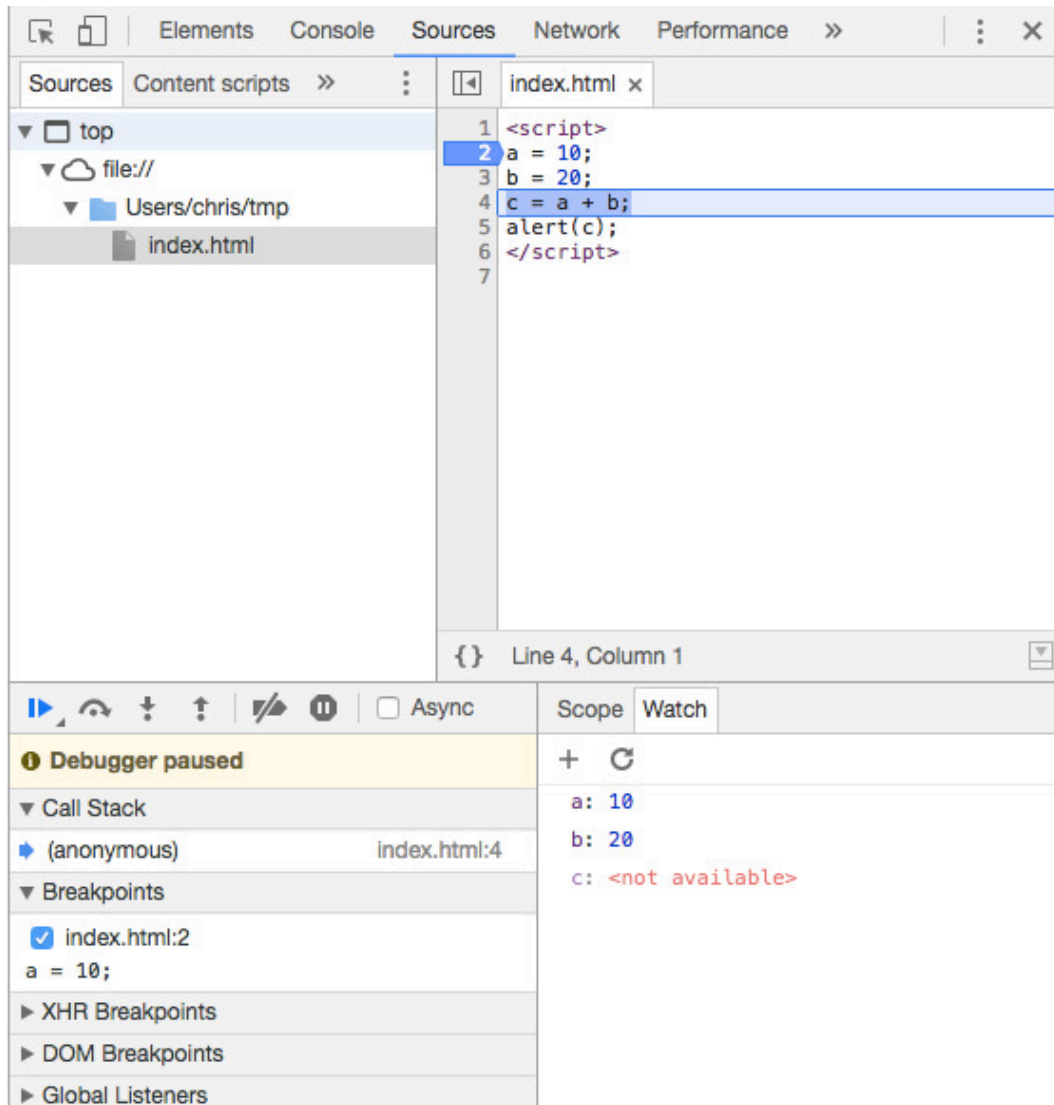


Figure 2.10: Chrome Debugger after stepping twice. Note the values in the Watch window and the indicator showing what will execute next.

- evaluate
- recurrence relation
- recursive function

Exercise 2-2: Writing functions

Solution on page 371

Using pencil and paper, write a function using algebra notation that represents *all of the steps* of the following computation:

1. Square x ,
2. Multiply y by 6,
3. Add them together,
4. Subtract 7 from that, and
5. Divide the whole thing by 2.

Exercise 2-3: Defining functions

Solution on page 371

Write a function f that accepts a *non-negative integer* x (in other words, you do not need to worry about negative values in your answer) and gives instructions to

1. Produce the value 1 if $x < 2$
2. Otherwise add $f(x - 1)$ to $f(x - 2)$.

Exercise 2-4: Recursive evaluation, basic

Solution on page 372

Given the function $h(x) = x + h(x - 1)$ where $h(0) = 0$, evaluate $h(5)$ showing all steps. Do not completely evaluate any subexpressions until all of their parts are available. Note that as a result of this, you will have multiple levels of brackets during parts of the computation.

Exercise 2-5: Recursive evaluation, more advanced

Solution on page 373

Using pencil and paper, evaluate $f(6)$ using the definition below (this is the Fibonacci Sequence). Show all steps, and do not do any work early (do not take advantage of the associativity of addition, for example, just systematically complete the innermost expressions first).

$$f(x) = \begin{cases} 1 & \text{for } x < 2 \\ f(x - 1) + f(x - 2) & \text{otherwise.} \end{cases}$$

Exercise 2-6: Match terms

Solution on page 376

Match the following terms

- substitution
- evaluation
- function
- call
- recursion
- break point
- watch
- step

with their definitions:

- A definition of a process of computation.
- The process of replacing a variable with its value.
- The process of performing a computation to produce a value.
- Supplying a function with an argument and getting a result.
- A computation that is defined in terms of itself.
- In a debugger, execute one statement of code and then stop.
- In a debugger, show the current value of a variable.
- In a debugger, where a program should pause for inspection.

Exercise 2-7: Pencil and Paper Debugging

Solution on page 376

Given the program below, pretend that you have entered it into the computer, set a debug watch on `x` and `y`, and are stepping through it one line at a time, starting at the top. What are the values of `x` and `y` at each step? Show them in a table like this:

x	y
—	—
—	—
—	—
—	—
—	—

Hint: `y` will be undefined until it has been assigned something.

:javascript:

```
x = 10;
x = x + 1;
x = x * 3;
y = 10 + 3 * x;
x = x - 3 + y;
```

Chapter 3

Writing Functions and Handling Events

We've spent some time learning about how to both define and evaluate algebraic functions, and in the last chapter we learned how to **evaluate** (call) JavaScript functions, but not how to **define** them. Now it's time for us to learn how to define them.

This chapter will cover some of the most new and foreign ideas that you will see at this early stage. These ideas are also probably the most foundational in the language, so they can be tricky to tackle at first. This is the point at which a lot of students think, "This is super confusing, I'm obviously not cut out for this." Instead, what's really true is that this might well be the hardest chapter in the book. Because of that, we take our time and go over things thoroughly. That said, if you get to the end of the chapter and think "Wow, that made no sense at all," skim through it one or two more times, then leave it behind. It's likely that it will make more sense when we get into writing actual games, and there's plenty of review later on in that context.

If this chapter takes a little more effort or time than others, that's okay. Get to the top of this summit and enjoy the view! The next hill is much gentler.

Writing Functions

To really understand what is going on with functions, we need to learn how to write our own. JavaScript, particularly in the browser, has a truckload of built-in abilities that you can access by calling functions, and we will only cover a few of them in this course, learning as we go. Right now, let's dig in and try to understand how functions are structured and what they are all about.

A Quick Refresher

To begin with, we will do something basic, like prompting for a number and returning its square. To do that, we will use only things that we have learned thus far. Here is a small program that does this if you type in a number:

:html:

```
<script>
num = prompt("Enter a number:");
sq = num * num;
msg = "The square of " + num + " is " + sq + ".";
alert(msg);
</script>
```

Let's review every part of this script, just to be sure it's clear.

First, we are in a file whose name ends in `.html`. That's very important. This is an HTML document, written in a text editor, then opened in your browser. If you name your file differently, the browser might just show you your program's source code instead of actually running it.

Next, the program appears inside of HTML `<script>` and `</script>` tags (opening and closing). What's between (inside of) those two tags is our JavaScript program.

Now, looking at the program itself, we see a variable assignment. That's the `num =` line. It assigns the result of the thing on the right to the variable `num`. What's on the right? A call to the `prompt` function. When you see the prompt and type in a number, this line sets the value of `num` to be that number.

The next line, `sq = num * num`, multiplies the number you entered by itself to produce its square. It then assigns that result to `sq`, another variable.

Our final variable assignment is to the `msg` variable, and it gets a string. What does that string look like? Here we have used **concatenation** to create a single string from a lot of little pieces. Reading from left to right, we assemble the string by concatenating

- "The square of "
- num
- " is "
- sq
- "."

There are a couple of things going on here that need to be mentioned. Remember how we talked about **types** earlier? Something like "hello" is a **string** of five characters (the quote marks delimit the string, but they are not part of it). Something like 10 is a **number** that can be divided, multiplied, added, subtracted, etc. Here are a couple of quick quiz questions, with their answers:

Question: When you type something into the prompt window, and it gets assigned to `num`,

what type is `num`?

Answer: It's a string. Everything you type into a prompt window is always a string. You type the digits one at a time, and their corresponding characters appear. Those characters are returned as a single result string from a call to `prompt`.

OK, so `num` holds a string. Here's another question:

Question: How does `num * num` work if `num` is really a string? Strings can't be multiplied!

Answer: The JavaScript interpreter, when it sees what you are trying to do with a variable, might also try to convert it to a new type for you first. Imagine, for a moment, that there is a `stringToNumber` function built into the language that, given a string, returns the number that it represents, so `stringToNumber("10")` would return the actual number 10. When the interpreter sees `num * num`, it knows that `num` must be a number for that to work. If `num` is really a string, the interpreter silently rewrites your program so that it now reads `stringToNumber(num) * stringToNumber(num)`. In other words, it converts it to a number if it can.

This is called **implicit type coercion**, and while very convenient at times, it can also be a source of hidden trouble. What happens, for example, if you have `twoNums = num + num`? Well, `num` is a string, and it just so happens that `+` works on strings: it concatenates them. Therefore, the interpreter doesn't see a need to convert anything for you. That means if `num` is `"10"`, then instead of getting 20 from the addition, you will get `"1010"` (the string "10" concatenated with itself), which is not what you wanted!

We have seen how to force the interpreter to give us a number from a string, when we introduced the **unary plus** operator. To review, unary plus is in the same position as a negative sign would be, as in `+1`. That's legal in JavaScript, and in most cases doesn't do anything. But, if you apply it to a string, it will force it to convert to a number. Thus `+"10"` evaluates to the number 10.

Whenever you accept input from somewhere and want to be sure it's treated as a number, you can use that trick. If we were to apply it to our program above, we would write

:javascript:

```
num = +prompt("Enter a number:");
```

That little plus sign is pretty helpful. We will be seeing it again.

Note that type coercion works the other way, too. If the interpreter sees `"My favorite number is " + 7`, it converts 7 to the string `"7"` before performing concatenation, which would produce `"My favorite number is 7"`.

That's the only mildly tricky part about the `msg` assignment above: string concatenation (jamming all of the pieces together), results in an implicit conversion of `sq` to a string.

With all of that review out of the way, the final line displays our concatenation-constructed string in an alert window by passing it to the built-in `alert` function.

Try running the program to see what happens. Remember, stuff insides of quote marks is all important (including spaces) because they are the literal characters of the string. Outside of quote marks, spaces are less important.

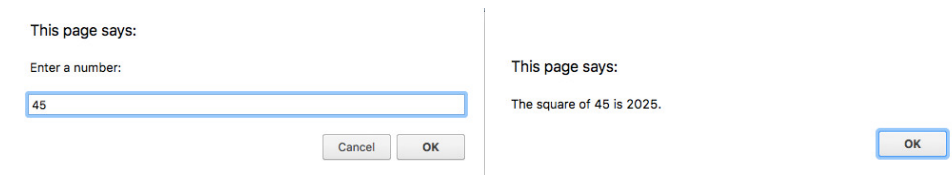


Figure 3.1: Prompt and result.

Our First Function

We have thus far been using built-in functions. Let's learn to write a brand new function of our own! We can use it to compute something from a number we enter via `prompt`.

To define a function in JavaScript, we use the `function` keyword. We give our function three essential things: a **name**, a **parameter list**, and a **body**, like this:

```
:javascript:
```

```
function doStuff(p, q, r) {
    return p + q + r;
}
```

The name (`doStuff`) and parameters (`p`, `q`, `r`) should look familiar from algebra. Parameters are just variables inside the function body. That body, which in this example is the statement `return p + q + r`, is the definition of what the function actually computes. It's like a miniature program that can produce a value. This particular function merely adds its three arguments together and returns that result, but functions can do much more than that. Note that we will talk about the **return** keyword in greater detail in just a moment.

You might have noticed that there is no semicolon after the function definition's final closing brace. It isn't needed here. This is true for a few other constructs we will learn about in later chapters, as well, such as loops, conditionals, and class definitions.

With syntax out of the way, let's define a function we can use in our program.

```
:html:
```

```
<script>
// This is your function definition.
// It is setting things up for later use.
function sq(num) {
    return num * num;
```



```

}

// This is your program. This is where you
// actually call (use) your function definition.
n = +prompt("Enter a number:");
msg = "The square of " + n + " is " + sq(n) + ".";
alert(msg);
</script>

```

This should do exactly what it did before, but this time we are using our brand new function to compute the square. It works just like you would expect: the value of `sq(n)` is computed by running the statements inside the body of `function sq(num)`. That body has exactly one statement in it: `return num * num;`.

You can think of this algebraic version:

$$\text{sq}(\text{num}) = \text{num} \cdot \text{num}$$

as being basically equivalent to this JavaScript version

:javascript:

```

function sq(num) {
    return num * num;
}

```

Really they're just different notation, different syntax for the same idea.

We glossed over the `return` statement, though. What does it do? It sets the value of the function call, when it is finished.

The `return` statement actually does one more thing. It also *always ends the function computation immediately*, no matter what. If the interpreter encounters a `return` in a function, that function exits right away with the specified value, no matter what might come later inside the function body. We will see some examples of that as we progress in this course.

Review of Function Evaluation

If you got through the section on algebraic functions, this should feel pretty natural at this point. If you would like a very thorough review in this new context of the JavaScript function definition, read on, otherwise skip to the next section on variable scopes.

Let's assume that `n` is 12 (because that's what we typed when prompted). If we substitute that everywhere in our program, we end up trying to compute something like this (simplified a bit, since we know `n` is 12 already):

:javascript:

```
msg = "The square of 12 is " + sq(12) + ".";
```

Listening in on the interpreter's imaginary internal dialog, we might hear something like this:

- It looks like this is a variable assignment because `msg = something`. Let's make space for "msg" as a variable and see what we get for it.
- Oh, that "something" is a string. It says "The square of 12 is", but that's not the whole thing. It appears that we need to add it to something.
- That "something" is a function call. We can't do anything until we have a value for that.

Here the interpreter pauses a moment to remember what it has already figured out, and what it has not, so that it can come back to this very spot when the function call completes. Having made a note of that, it then calls the function. This means it jumps to the function's body with (in this case) `num` set to the argument 12 that we passed into it. At this point, the interpreter starts executing the body, which is just the `return num * num;` statement.

Thus, the interpreter now has the value of `sq(12)`: it's 144, giving it this (again, in its imaginary internal dialog), after concatenation:

:javascript:

```
msg = "The square of 12 is 144.";
```

And that should be what we see in the alert window that pops up after we enter 12 in the prompt. Give it a try!

Putting it all together in one place, here is how the interpreter computes the final result, one step at a time, right up until it has to call the function:

- `msg = "The square of " + n + " is " + sq(n) + ".";`
- `msg = "The square of " + 12 + " is " + sq(n) + ".";`
- `msg = "The square of 12" + " is " + sq(n) + ".";`
- `msg = "The square of 12 is " + sq(n) + ".";`
- `msg = "The square of 12 is " + sq(12) + ".";`

Now it has to call `sq(12)` to get a result. That function returns `num * num` (tasty), which is `12 * 12` (gross), and the interpreter continues computing the sentence:

- `msg = "The square of 12 is " + (12 * 12) + ".";`
- `msg = "The square of 12 is " + 144 + ".";`
- `msg = "The square of 12 is 144" + ".";`
- `msg = "The square of 12 is 144.";`

Congratulations, you just wrote your very own function and walked through its evaluation in an expression! Granted, it's small and doesn't do a whole lot, but this is such a critical concept that it is worth covering with some trivial examples first. If you have understood all of this,

you're in good shape. If any part of it is still uncomfortable, now is the time to go back and deepen your understanding. Everything else in this chapter hinges on these concepts.

Variable Scopes

Now that we have a program with a function in it, we are ready to talk more about **scopes**. Let's look at our function in a new program:

:javascript:

```
// Here we define our function.
function sq(num) {
    return num * num;
}

// Here we use it in our program.
a = sq(10);

// ** YOU ARE HERE **.
```

There is a lovely little comment in there that says `** YOU ARE HERE **`. At that point in the program, what is the value of `a`?

Now for a more interesting question, since you got that one right the first time: right there where it says `** YOU ARE HERE **`, what is the value of `num`?

That is kind of a trick question. The answer is “The variable `num` is not defined right there because it only exists inside of the function `sq`.” Did you get tricked?

In other words, the variable `num` is “not in scope” at that point in the program. It is only “in scope” inside the body of the `sq` function. It doesn't exist outside.

Parameters Are Local

The variable `num` is what we call **local** to the function `sq`'s body. It is in the body's **local scope** and is thus only valid during a call to that function. When we call `sq(10)`, the variable `num` inside of `sq` has the value `10`, but only while that function's body is being executed that one time.

If that all makes perfect sense, then great! You're all done with the important parts of this chapter. You can skip to the section on global variables.

If not, or you would like to see more examples of what the debugger can do, read on.

Debugger Scopes

If you can, fire up the debugger again and see what's going on here. Remember, this is your program, and you will need to type it into a file, save it, and load it in the browser before

debugging can work:

:html:

```
<script>
function sq(num) {
    return num * num;
}

n = prompt("Enter a number:");
msg = "The square of " + n + " is " + sq(n) + ".";
alert(msg);
</script>
```

As a reminder, to get to the debugger, you need to open your developer tools and find the debugger within them. In whatever browser you are using, you will need to see your source code so you can set a breakpoint and step through that code. In Chrome the debugger is under “Sources” and in Firefox it is under “JavaScript Debugger”. As a quick refresher, these are the steps you want to follow for this exercise:

- Load your program in a browser tab.
- In that same tab, make sure the developer tools are up.
- Find your source code in those tools (the text of your program).
- Set a breakpoint where the `msg` variable is assigned (click on the number in the left margin of your code window—it should be highlighted now).
- Reload the page and enter a number when prompted.

At this point, the debugger should look like it is waiting on the line where your breakpoint is. In the past, we used the “Step Over” button to get to the next statement. This basically says “treat the current line as one operation and just do it all at once.” What we really want to do now, though, is see what happens when we step *into* that `sq(n)` function call. Before we do, let’s use the console to find out the status of things.

If I run this program and enter the number 9 when prompted, then it stops at the `msg =` line because of a breakpoint, I can enter `n` and `msg` as watch variables (or I can type them into the console) to see what they contain. If I do that, I will see that `n` equals “9” and `msg` does not have a value yet.

Now comes the magic. Use the “Step Into” debugger button (it is usually an arrow pointing at a dot instead of jumping over a dot). This is called “Step Into” because, when there is a function call on the current line, it will take you inside that function so you can see what is happening in there. Press that button. You should see the highlighted line jump up to the interior of the `sq` function. You will also see something like “Local” or “Function Scope” in the debugger window if you click on “Scope”, and it should have your `num` variable in it, with the value “9”!

This actually brings up an important point: note that the value of `num` is “9” (a string) instead of 9 (a number)! Everything will still work, because JavaScript is overzealous in its desire to

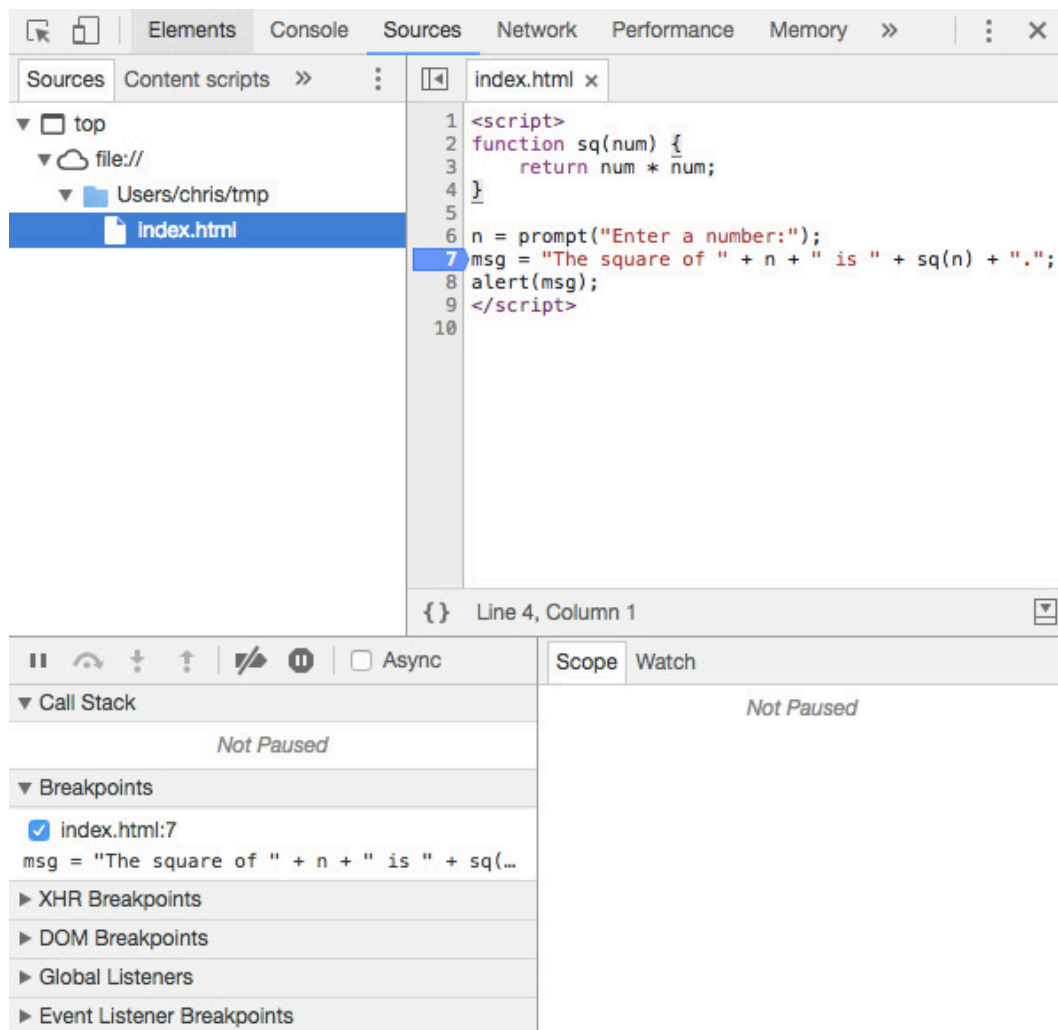


Figure 3.2: The Chrome Debugger with our program loaded and a breakpoint on the msg line.

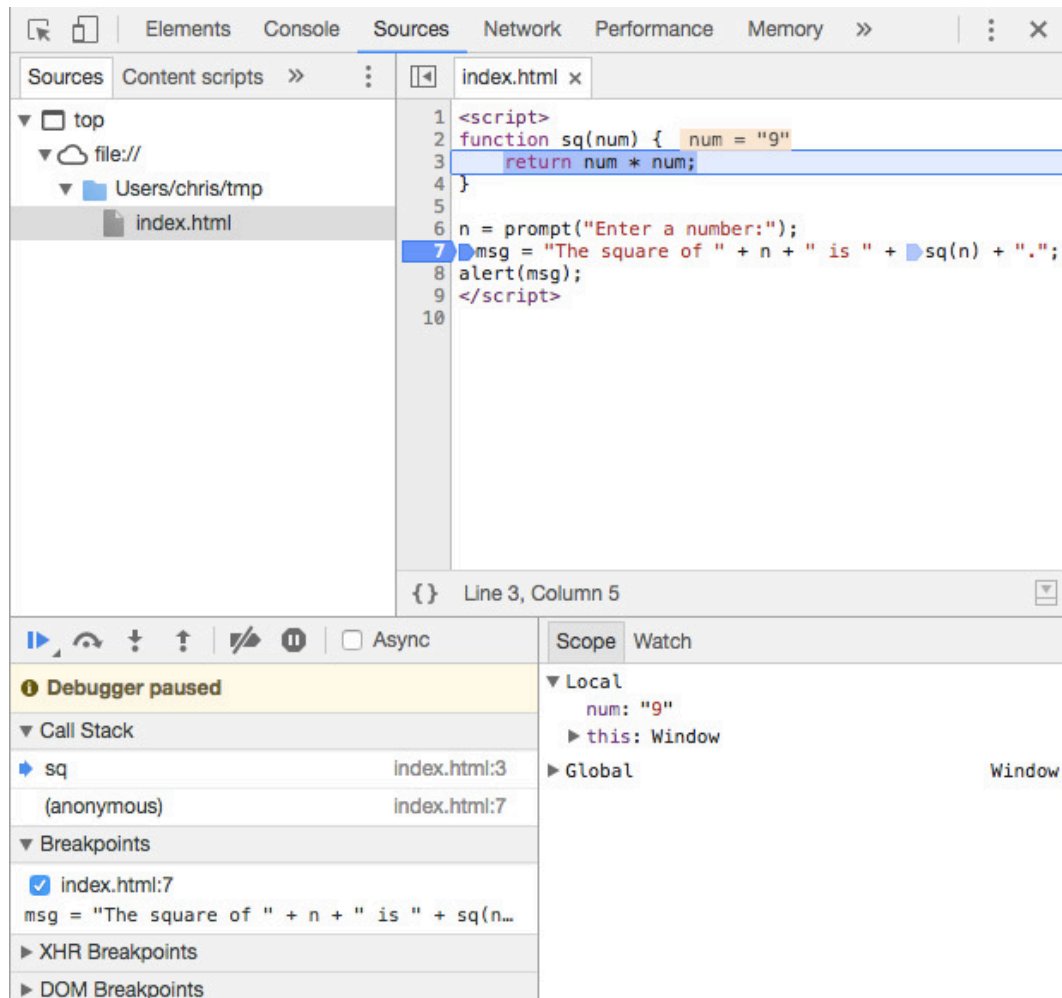


Figure 3.3: The Chrome Debugger, just after using “step into” to get to the body of “sq”.

help us out, but running this in the debugger helps us to see that we might have a mistake waiting to happen. We ought to be calling `sq(+n)` to ensure that we get a number in there. We won't bother fixing it right now, but it's useful to note how the debugger showed us something that might not have been obvious without it.

That's pretty handy! Now, we will use the "Step Out" button (an arrow pointing *away from* a dot) to run the program just until this function has returned. We could also use "Step Over" until we finish with it, but "Step Out" is a good thing to learn about, so we will use it this time. Play around with these things. It's kind of addictive watching your code execute one step at a time and learning how to navigate around in it, and it is incredibly useful for gaining intuition about what the computer is doing with your masterpiece.

When we step out and continue stepping enough to finish the assignment of `msg`, we can see what it contains by adding a watch for it. It should have a message like "The square of 9 is 81.", which it will then alert to us.

By the way, if you get sick of stepping around, you can always hit the "Play" button to run to completion or to the next breakpoint, whichever comes first.

Global Variables

Once again, the variable `num` inside of the function `sq` is a **local variable**. It springs into existence when our function is called, is only visible inside the function body, and disappears when the function returns (either with the `return` keyword, or by falling off the end, which happens to be the same thing in our tiny function).

In contrast, many of the variables we have created before now have been **global variables**. This means that once we defined them, we could basically access them from anywhere in our program. Our programs have been pretty short, and they will be short for a little while yet, but even very long programs can use global variables from anywhere. They can be seen from inside functions (you can see `n` from inside of `sq` as well, since it has been assigned before the call) and from inside code loaded from other files or internet sites. They are really, truly global to anything in the current browser window.

The global scope in the browser is actually the `window` object. You might have seen that in the debugger. If you expand it, you will see a huge number of things in there, including your very own variables. We will get into objects in the next chapter, so don't worry about what that means right now if it doesn't already make sense to you.

This high level of visibility makes global variables both powerful and dangerous. You can leak sensitive information using global variables that other code used by your website, but not written by you, can read. You can stomp on important global values by accidentally assigning them, and thus make some functionality disappear or act weirdly. In general, global variables are easy to understand but hard to use safely. Local variables are much safer and easier to

reason about.

Non-Parameter Local Variables

Remember how we mentioned that there are three things you can do with variables, but we only talked about two of them? Here are the three things:

- Assignment
- Evaluation
- Declaration

We talked about assignment and evaluation, or setting and getting values, but we deferred the discussion of declaration until later. Well, later is now.

We can **declare** a variable with the keyword `var`¹. When we use that inside of a function, it creates a new variable local to that function's scope, like this:

:javascript:

```
function poly(x) {  
  var y = x * x;  
  return 2 * y + x;  
}
```

The function body is a small program. That program sets `y` to be the square of `x`, then uses it in the final computation that is returned.

You will almost always see the `var` keyword in the context of a variable assignment, as with `var y = x * x;`. It can also appear by itself, though, like this:

:javascript:

```
var y;
```

In this case it only declares the variable to be local without giving it a value at all. That is only rarely useful, though; it is basically the same as saying `var y = undefined;`.

Since browsers support ECMAScript 2015 pretty uniformly by the time this is published, the `let` and `const` keywords are better choices than `var`. They have nicer properties generally, but we are sticking with `var` for now because it is still by far the most common thing you will encounter in the wild. Times are changing fast, though. If you want to get up with those times, you can use `const` instead of `var` for variables that are only assigned when created, and `let` for variables that are assigned more than once. Feel free to use them - they (mostly) work just like `var`, but are truly block-scoped and work better in loops with closures.

Now, what would happen if we left off the `var` keyword inside our function? Our variable `y` would be **global**. The default scope, unless otherwise specified, is the global scope. Because

¹<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/var>

of that, from now on we will always declare our variables with `var`. That way they will be local when inside of functions.

A quick note in order: you can declare and assign multiple variables at a time, separating their declarations (and assignments) with commas:

:javascript:

```
function foo() {  
    var x = 10, y = x * 2;  
    // other stuff here.  
}
```

Multiple assignments with a single `var` keyword are quite common. They can be made a little easier to read if separate lines are used, thus:

:javascript:

```
var x = 10,  
    y = x * 2;
```

and that is basically the same as this:

:javascript:

```
var x = 10;  
var y = x * 2;
```

Either way works. Use whatever is comfortable.

The name `foo` is part of a long tradition of temporary names in programming tutorials. It sits alongside its friends `bar` and `baz` (or `spam` and `eggs` if we're talking about Python tutorials). It is, admittedly, not the most useful name, but then again, in tutorials we aren't always making the most useful functions. You really shouldn't see it in "real" code, because if the code is doing something real, "foo" is a terrible, meaningless name. It works here, though! We're in temporary tutorial territory.

Functions Are Values

We have so far covered two very important things that you can do with functions:

1. You can **define** them, and
2. You can **call** them.

We define a function using the `function` keyword, as we have seen many times in this chapter. We call it by mentioning its name, followed by arguments in parentheses.

If we look at functions like little machines with inputs and outputs, as we have been drawing them, they look something like the function box containing a body of code.

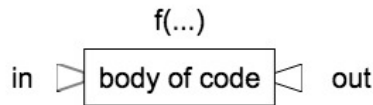


Figure 3.4: A function box with code.

Functions, then, are kind of like little machines that contain a bundle of code.

Variables Can Hold Functions

If functions are like machines, let's see how far this analogy takes us. You can build a machine, you can operate a machine, and you can move a machine from one place to another, not unlike a toaster.

Functions are like that, too. You can define (build) functions, you can call (operate) functions, and you can move functions around just like strings or numbers or any other value in JavaScript. This means, among other things, that you can assign them to variables and pass them into other functions, not entirely unlike tossing your toaster into an industrial shredder to make tiny metal pieces: one machine operating on another.

This all comes down to a simple fact: functions are also values! Crucially, that means variables can not only hold numbers and strings, *they can also hold functions*.

Values (things you can stash into variables) that you have encountered thus far are things like

- `10`: a “number”
- `"Hi there"`: a “string”
- `prompt`: a ... wait, what is that anyway? It's a function!

To drive this home a bit more, let's remind ourselves of what happens when we define a variable in the console:

```

:console:
> x = 10
< 10
> x
< 10
  
```

We assign `x`, then (second-to-last line), we ask the console to show us its value. It dutifully replies `10`. Now try this with `prompt`, which is already defined:

```

:console:
  
```

```
> prompt
< function prompt() { [native code] }
```

Note that we didn't include the parentheses when we typed `prompt` above. If we wanted to call it, we would have done something like `prompt('hello')`: putting parentheses after a function triggers a call. But just mentioning it without those parentheses treats it a lot like a variable: it tells us what it refers to.

You could also have tried this:

```
> alert(prompt)
```

:console:

You are basically saying “`prompt` is a thing - please alert what it is”. If you tried that, you would see an alert window pop up showing you something about the `prompt` function (because you passed it as an argument). Here we again see the difference between calling a function and just mentioning it like a variable: we're calling `alert` (it has parentheses after it), but just mentioning `prompt` (no parentheses) as we pass it to `alert`.

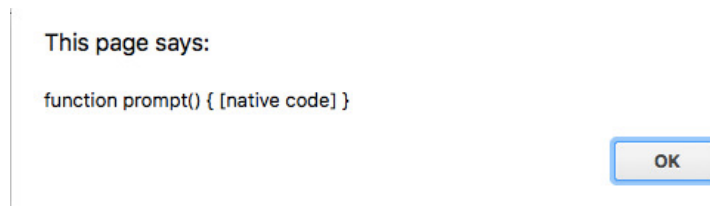


Figure 3.5: Alerting the prompt function.

Okay, then: the word `prompt` acts an awful lot like a variable, because if you mention it, it is evaluated. It just so happens that its value is a function. That's interesting! Can we assign variables to functions in general? Yes, we can. If you assign `f` to be whatever `alert` is, then call `f()`, you get an alert window, like this:

```
> var f = alert
> f("hi");
```

:console:

And the alert window pops up!

Basically, `f` and `alert` both contain the same function, now. You can call it either by its original name `alert` or by the variable that holds it, `f`. They both “point to” the same little machine we call a function.

Not all function names behave exactly like variables, but every variable can hold a function. Some function names are considered very special and cannot be reassigned, because that would mean that programmers could overwrite essential system code and

mess things up for other programmers. Most web sites contain code written by numerous people who did not cooperate with each other to write their little piece. The cooperation they *do* have is implicit: it assumes that essential functionality can be relied upon to be there under common names. Some stuff cannot be mucked about with in this way, so it does not *quite* act like a variable. But most of the time a function name is exactly like a variable.

On the other hand, some critical system functions *can* be overwritten in JavaScript, reminding us once again that every language has warts, and sometimes it is far too late to freeze them off.

Anonymous Functions

Since we can assign a variable to refer to a function, that means that the function does not really need to have a name. We can get it by going through the variable that holds it. Let's consider a new simple `poly` function that computes $2x^2 + x$:

:javascript:

```
function poly(x) {  
    return 2 * x * x + x;  
}
```

This is pretty straightforward. We accept one parameter, which becomes the variable `x` when the code inside is running. Then we basically compute $2x^2 + x$ and return it. We had to give this thing a name, so we did. But, it is also possible to create a function *in place* without giving it a name. This will become important in just a moment, but first let's see a different way of writing this:

:javascript:

```
var poly = function(x) {  
    return 2 * x * x + x;  
};
```

The difference is perhaps a bit subtle, but very important: the function itself has no name. What we have here is a **variable assignment** with an anonymous (no name) **function expression**. Remember how expressions are “things that have a value”? A nameless function expression has a value: it is an **anonymous function**. You can set variables equal to any expression, including one of those.

It turns out that *all* function definitions are expressions in JavaScript, including those with names. It is valid to assign a variable to a named function definition, and it is even sometimes useful to do so. We will not make use of it here, at least not yet.

That means that after the above code runs, `poly` points to a function that we defined right there, in place. This has some far-reaching implications when you think about all of the things you can do with values. So far, for example, we have seen that not only can you assign

variables to them, you can also pass them in as function arguments. The next section will show a common and very useful example of that, where it is likely to make more sense in context.

Functions That Take Functions

Here's a fun little bit of code to study, but not worry too much about understanding fully, at least not yet:

:javascript:

```
// This function accepts another function as a parameter, then calls it.
function callThisThing(f) {
    f();
}

// This is the function we want callThisThing to call.
function myFunction() {
    alert("You called!");
}

// This is our program: it tells callThisThing to call myFunction.
callThisThing(myFunction);
```

What is this doing? We define a function that accepts another function as a parameter. We have called this function `callThisThing` because that's what it does: it calls the function we pass to it. That's kind of different, right?

We then define a function for it to call: `myFunction`. If someone, anyone, calls this function, it alerts, "You called!" Finally, we pass `myFunction` (a function that alerts) into `callThisThing`, which immediately calls our function and causes an alert.

This is a silly example, but there is nothing in here that you have not seen before. It is all a bit new and probably not settled in your mind, yet, but it's worth looking at carefully to understand all of the things that are happening. However, if it's just a bit too weird for now, move on and stay tuned. We will soon see this in context, and that will likely help clarify things.

Why are we talking about this, though, when you could just call `myFunction` directly, instead of asking someone else to call it for you? The reason is that this is how events work in JavaScript, and we are about to talk about those. As a bit of foreshadowing, there are functions that already exist in JavaScript that act a lot like `callThisThing` does above, and one of those functions is called `setTimeout`. Let's see how events work, and how `setTimeout` works in particular.

Events

So far our programs have been fairly linear in nature: the browser loads our code and starts executing it in order (temporarily jumping back to function definitions when they're called). It

starts at the top, does things until it reaches the bottom, and quits. It does this as fast as it can.

Computers need to interact with the real world, however, and the real world has its own schedule. Time marches on, regardless of how long our program takes to run. People bang on keys on a keyboard or touch spots on a screen at moments that our programs can't predict.

When things happen outside of our program, and we want to react to them, they are called **events**. We will get much more deeply into events in coming chapters, but for now we are going to talk about one particular event to illustrate the principle: the passage of a predetermined amount of time.

Let's suppose we wanted to make a little timer. After we load the page, we wait ten seconds, and after that time an alert pops up saying "Done!" How would we do that? How do we wait for time to pass, then trigger an action? We use the built-in `setTimeout` function, which instructs the browser to call our function after a time delay that we specify.

To experience `setTimeout`, create a new program that looks something like this:

```
                                :html:

<script>
// This is how to "call us back".
function done() {
    alert("Done!");
}

// This asks the browser to call back after 10 seconds.
setTimeout(done, 10000);
</script>
```

This is an example of how you can tell other processes to call you back: you create a function, and you give that function to them so they can call it later. Here we ask the browser to call `done` after 10,000 milliseconds (10 seconds).

Let's draw our attention to that call again: `setTimeout(done, 10000)`. The word `done` in there refers to our function. But because there are no parentheses after it, it is not called, it is merely passed into `setTimeout` as an argument.

Until now, functions have been sort of interesting as a concept, but we haven't really seen how to use them to organize programs because our programs have been tiny: we can just inline everything. This little bit of code, though, is something you cannot do without functions: we are setting aside a bundle of code so that it can be run later by some other process. For that, functions are required.

Granted, the `setTimeout` function also accepts a string containing JavaScript that it will interpret when the time comes, but don't do that. It's a wart. Practically speaking, it boils down to basically the same thing, but it is much less secure. Many other event registration functions do *not* accept strings. Nor should they. Just pass functions to get

callback behavior.

If you run the program, the browser will look like it isn't doing anything, but 10 seconds after loading the page, the "done" alert window will appear!

In this example, we named our function `done` and passed it into `setTimeout`, referring to it by name. In practice, you will see anonymous functions, too. The above program, for example, can be rewritten like this:

:javascript:

```
// Anonymous function assigned to the variable "done".
var done = function() {
  alert("Done!");
};

setTimeout(done, 10000);
```

That's not really very interesting, we just use a variable to hold an anonymous function instead of naming it directly. What if, instead, we take the next step and don't even create the variable? What if we define the function and pass it into `setTimeout` at the same time? Here is an example of that idea:

:javascript:

```
setTimeout(function() {
  alert("Done!");
}, 10000);
```

Do you remember when we created a message in a variable called `msg` and then called `alert(msg)`? And then later, we just put the message right inside the call, like `alert("hello")`? What we were doing there was changing from passing a variable to passing a literal (a string literal, in this case).

The code above is using the same idea. Earlier, we created a `done` variable and passed it into `setTimeout`, but there's no reason we can't put a **function literal** directly into `setTimeout`, and that's what this most recent example does.

It looks a little bit harder to understand at first blush, but it's very common in JavaScript, as it removes the need to think of a useful name for your function and makes it clear exactly what that function's purpose is. We will make extensive use of this idea going forward.

It can, however, look a little bit confusing, especially at first. If you run into something like this and aren't sure what to make of it, how can you figure it out? It may help you to think of things in terms of layers. When you run into something tricky, take a deep breath and start peeling layers away one at a time. Try to act like you are the interpreter. Run your own mental Read/Eval/Loop machine. If you were to do that here, it might go like this:

- `setTimeout` is a function because it is being called (it has parentheses after its name).

- Everything inside of its parentheses is an argument.
- The basic structure therefore looks like `setTimeout(...)`.

At this point, you can peel off `setTimeout(...)` and set it aside in your mind. Now focus on what's inside:

- Function arguments are separated by commas. Are there any commas?
- Yes, between a longish function thing and the number `10000`.
- That longish function thing is the first argument, and `10000` is the second.

And now you are basically there. Now you can see that the anonymous function will call `alert("Done!")` at some point in the future. You know that because you are calling `setTimeout` and handing it two arguments: a function that will `alert("Done")`, and a time of 10,000 milliseconds.

This deconstruction process applies everywhere. If you run up against something that seems too complex, start peeling off layers one at a time and you will get there. It can even be helpful to take a confusing piece of code like this and rewrite it to a form without anonymous functions, to see if it becomes clear when written differently. That can help to detangle the ideas. After a while, though, constructs like the above start to look pretty familiar and transparent. It's just getting over this initial hump that takes some effort. The key is to remember that it is something you already know how to do, and you are capable of understanding it, given a little study and time.

Of course, if you already know that `setTimeout` expects a function as its first parameter, then the process is much simpler: you see `setTimeout` and immediately know that the first parameter is a function and the second is a delay in milliseconds. That is obviously a faster way of understanding what is going on, and it comes with experience as you build up your own personal function vocabulary.

With the concepts of values, variables, and functions in your repertoire, you are prepared to understand pretty much all of the rest of the language. As with learning a natural language, there is a phase where you spend a lot of time with the grammar, and then there is the rest of it where you learn a lot of vocabulary and idioms, and get practice putting things together in sensible ways. This was probably one of the hardest grammar lessons, so congratulations! There are a few more bits of grammar to learn that are highly useful, but none of them are as deep as function definitions, function calls, anonymous functions, and events.

We are not done with functions yet, as we have not talked about closures and prototypes, but those come much later. What we have learned here will get us quite far.

Summary

That was a lot to take in, and this was probably one of the most foundational chapters in the entire curriculum. Please make sure that you feel reasonably comfortable with it before moving

on. If you can understand how functions work, how they are called, and what locally-scoped variables are, then you are well on your way to being able to understand all of the rest, even if some of it is still a little bit fuzzy. Fuzziness is fine: these concepts will come up again in the context of real programs, which will help make things clear.

These concepts are not only some of the most critical to understand, they are also some of the most common stumbling blocks, and many students give up right here, at this point, right when things are starting to get interesting! You can move forward from here. You have basically got it made, so congratulations! This was the biggest conceptual hump. There is more climbing to do, but getting through this part of the climb has made you stronger. You can do the rest.

Exercises

Exercise 3-1: Parameters and Arguments

Solution on page 378

What is the difference between a “parameter” and an “argument” when talking about functions?

Exercise 3-2: Function Anatomy

Solution on page 378

In a function definition like this one,

:javascript:

```
function A(B) {  
  // C  
}
```

which parts of the function (A, B, and C) correspond to

- the name
- the parameter(s)
- the body

Exercise 3-3: Return

Solution on page 378

What does the `return` keyword do in a function?

Exercise 3-4: Function Definition

Solution on page 379

Define (and write down below) a function called `mid` with two formal parameters `low` and `high` that returns the number right in the middle of the two. Hint: this number will be the *average* of `low` and `high`.

When finished, test your function in a real program by passing it the following values and alerting the results. Then fill in the answers:

low	high	mid(low, high)
4	6	
2	10	
-5	2	
17	37	

Exercise 3-5: Scopes

Solution on page 381

In the following program snippet, fill in the table with the scope of all variables. These will be either “global” or “local”. Recall that global variables can be seen both outside *and* inside of functions, and local variables are only visible inside and during a call.

Also, remember what `var` does. It’s kind of important.

:javascript:

```
x = 10;

function f(a, b, c) {
  z = 3 * c;
  var d = a + b - z;
  return d * d;
}

y = 15;

alert(f(x, y, 5));
```

variable	scope
x	
y	
z	
a	
b	
c	
d	

Exercise 3-6: Anonymous Functions and Timers

Solution on page 381

Write code to set a timer to `alert("Ding!")` after 5.5 seconds using `setTimeout`. Use two different methods, one with a named function, and one with an anonymous function. Recall that `setTimeout` is called like this: `setTimeout(functionToCall, delayMilliseconds)`.

Exercise 3-7: A Tiny Game

Solution on page 382

Write a program that, each time you reload the page, prompts for a number and alerts its signed distance from the number 5. See if you can get someone else to play it and figure out what it is doing without telling them.

Hint: the signed distance of the number x from 5 is $x - 5$.

Chapter 4

Objects

The heart and soul of a language is not really the syntax, though we do have to spend quite a lot of time on that at first. Without syntax, we can't really express ourselves in the language. It is a critical thing to build on, but it isn't the only important thing.

Programming languages are a lot like natural languages in some respects. There is what we call the "language specification", consisting of the way we write things like variable assignments, how function definitions are structured, what special key words like `return` mean and do, etc. But that can really only get you so far. After that, we need what we call a "standard library", which is analagous to a language's basic vocabulary. We need *words* for this language we are learning!

We are definitely not done learning the core language yet, though! In fact, we have to learn another core concept just to *use* the standard library, and then we can make use of it while learning the rest of the core language in later chapters, which makes things a lot more fun and interesting. The next subject that opens the door to all others is the concept of **objects**.

Objects Are Containers

If you want to get excessively technical and precise, pretty much every value in JavaScript is an **object**. While this is true, it is not terribly helpful right now. Let's pretend for a moment that objects are special things and that they are different from other things, and then maybe we can talk about how it is essentially objects all the way down.

An **object** in JavaScript can be usefully thought of as a container for other things. Each item within it has a unique **key** and a **value** to go with it. It is much easier to explain this using an example, so let's do that by opening up the console and playing around.

In the console (or in a file, of course), we can create a new object using curly braces, like this:

```
:console:
```

```
> {}  
< ► {}
```

The curly braces `{ }` delimit an object. Here we have created an “empty” object, as is evident by the fact that there is nothing between the braces. The console, after we did that, helpfully told us we had created an object. Good so far.

If you expand that object, you might see some interesting (but not terribly transparent or currently useful) things. Do feel free to poke around, but don’t worry about understanding what is there. Just know that even empty objects are not really all that empty; they just start out empty of things that we want to put in them. Let’s make an object with something of our own creation inside of it and assign the whole thing to a variable:

```
                                     :console:  
> var obj = { "cheer": "huzzah!" }  
> obj  
< ► {cheer: "huzzah!"}
```

Can you see what happened here? We put one thing into our object. We set up a key/value pair: the key is `"cheer"` and the value is `"huzzah!"`. They are separated by a colon character.



Figure 4.1: Our object `obj`.

A key and its value are called a **property** of the object, but that isn’t really important unless you are looking up documentation on the language.

Now, what can we do with this object? For one thing, we can get the value for a particular key:

```
                                     :console:  
> obj["cheer"]  
< "huzzah!"
```

We can thus access properties¹ inside the object by using square brackets `[]` around the key. That is called **bracket notation**, and it is quite useful. There is another way to access properties, too, if they happen to be valid identifiers² (meaning you could use them as variable

¹https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Property_Accessors

²https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Grammar_and_types#Variables

names—no spaces or dashes, cannot start with a number, and a few additional rules). In that case you can also access values by key using **dot notation**³:

:console:

```
> obj.cheer  
< "huzzah!"
```

When you see periods separating variable names like this, that means you are “drilling down” into an object, getting something from inside of it, by key name.

What else can we do with objects? We can set additional key/value pairs in them, as in this example (omitting some intermediate console values):

:console:

```
> obj.boo = "bad form!"  
> obj["10 little monkeys"] = "high hospital bills"  
> obj  
< ▶ {cheer: "huzzah!",  
...   boo: "bad form!",  
...   10 little monkeys: "high hospital bills"}
```

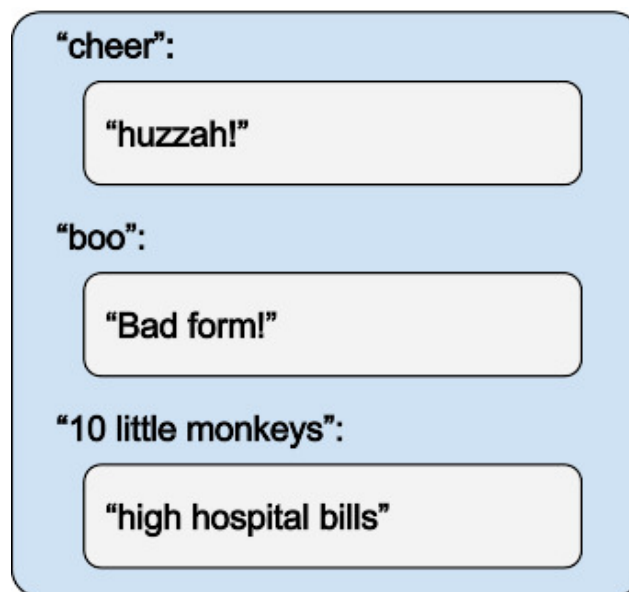


Figure 4.2: Our object `obj`.

Here we have used both dot notation and bracket notation to assign new values inside of the object. This looks an awful lot like variable assignment, right? It turns out that it works in exactly the same way. If the name you are referencing does not yet exist in the object, it will be created. If it does exist, its value will be overwritten with the new value you are specifying.

³https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Property_Accessors

But, because these names are all defined inside of an object, there is no need (indeed it is not possible) to declare them using `var`. Only the outermost object is a variable in your program; everything else here is a member of the object.

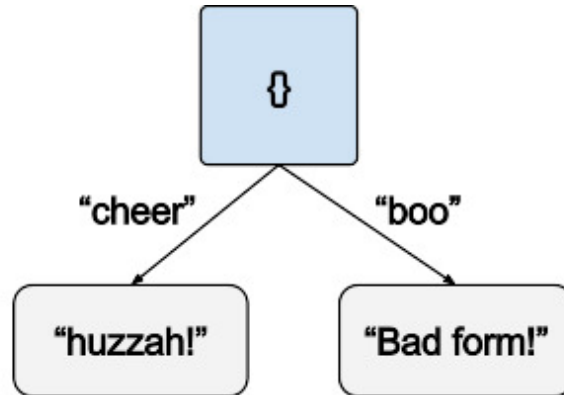


Figure 4.3: An object as a tree.

As before, we can get values from their keys using either dot notation (when possible) or bracket notation:

:console:

```
> obj["cheer"]
< "huzzah!"
> obj["10 little monkeys"]
< "high hospital bills"
> obj.boo
< "bad form!"
> obj.whatAmI
< undefined
```

If, as in the final example, we try to reference a key that doesn't exist, we get the special JavaScript value `undefined`, like we have seen with functions that do not return anything, or expressions that have no value. We will continue seeing that value a lot as we proceed through this JavaScript jungle. Just watch out for monkeys. They throw unsavory things.

We have now seen how to create objects with initial values, add values to objects, and retrieve them by key. What if we want to delete them?

It might be tempting to just try setting them to the special `undefined` value, but that's not quite what we want:

:console:

```
> obj["10 little monkeys"] = undefined
> obj
< ▶ {cheer: "huzzah!",
...   boo: "bad form!",
...   10 little monkeys: undefined}
```


Interesting. That left the “10 little monkeys” key in there, taking up space with the value `undefined`. To actually remove this, we need to use the `delete` statement⁴:

:console:

```
> delete obj["10 little monkeys"]
< true
> obj
< ▶ {cheer: "huzzah!", boo: "bad form!"}
```

Did you see how the value of `delete` was `true`? We are going to get into the boolean values `true` and `false` shortly, but for now, just know that `delete` returns `true` if it succeeds, and `false` if it does not. Note that it will always succeed when using it on your own objects and keys, even if the key isn't there, which makes it a fairly useless return value for our purposes.

The Console

Now that we have seen dot notation, we can start to explore a lot more of JavaScript's vocabulary: its **standard library**. Let's begin with something that all modern browsers have (except on mobile platforms like Android and iOS): a **console**.

Up to this point, we have used the console as a way of typing statements and seeing what happens. That is a very powerful feature of the console. We might have also seen some errors in our previous programs when loading them. The console is a place where error messages go so that if we want to see them, we can. A rather entertaining (and perhaps a little scary) thing to do is to load popular web sites with the console open and see how many errors show up. It turns out that our browsers are surprisingly robust in the face of some common kinds of errors. It also turns out that a lot of web sites are just broken and the authors do not appear to be aware of that fact. The console gives you the super power of awareness.

It also gives you something else: a place to dump information that you might want as a programmer, but that a user of your program might not care about. Errors are not the only things that can be logged to the console. You can log anything you like! Let's find out how. We will do this using the built-in `console` object.

The `console` object is part of the global scope in JavaScript programs running in the browser and environments like NodeJS⁵. It contains a number of things, one of which is the `console.log` function, which we can call to produce console output:

:console:

```
> console.log("hello there, console!")
hello there, console!
< undefined
```

⁴<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/delete>

⁵<http://nodejs.org>

Well, that's new! We have not one, but two lines coming out of our one console expression. Why is that? Let's remember that when we see something like this:

```

:console:
< undefined

```

That little symbol at the left means “this is the value of the expression you typed”. The expression we typed was a function call, and that function does not return anything. It is a lot like `alert` in that way.

A line without a symbol is just something written to the console. So, when we see this,

```

:console:
> console.log("hello there, console!")
  hello there, console!
< undefined

```

what we are seeing is

- `console.log(...)`: the command we typed,
- `hello there, console!`: the thing we told it to log, and
- `undefined`: the value of the call to `console.log`, which does not return a value.

This is going to be very useful to us. The `alert` function is handy and all, but we are very shortly going to want to see a lot more than just one piece of information at a time, and it would be nice to not have to tap “OK” all the time to dismiss our little windows. Now we can just open the console and see whatever we have logged there.

If you want to see more of what the console object contains, you can try just looking at it in (you'll never guess...) the console!

```

:console:
> console
< ▶ {}

```

That object will be expandable (click the arrow), so you can look at some of the things it contains by poking around. We are pretty much going to stick with `log` throughout this course, but you should always feel free to play, or to read up on the documentation for your environment's console, such as the Chrome⁶ or Firefox⁷ documentation (they are the same in the ways most important for this course).

⁶<https://developers.google.com/web/tools/chrome-devtools/console/>

⁷https://developer.mozilla.org/en-US/docs/Tools/Web_Console

The Standard Library

Everything in the standard library is organized inside of objects, which is why we needed to cover them here. Now we can walk through and see what we can actually do with this language. If you head over to the Mozilla Developer Network JavaScript Documentation⁸, you will see a huge amount of stuff. If you start poking around there (start with `Math` or `String` if you're not sure what to look at first), you might notice that there is an awful lot to know. It can be overwhelming. How will you ever remember it all?

One of the dirty little secrets of programming is that nobody actually does it the way it's depicted in movies. In movies you have a hero or villain staring at possibly several monitors all full of incomprehensible text, tapping away madly without once consulting anything other than their own brain to perform their on-screen magic. Quite often they are not even consulting the monitors.

While it is true that having more monitors is great for programming, and sometimes a programmer will get into the zone and work for a while without interruption or help, most of the time the (very, very smart) programmers I know have multiple tabs of documentation open in their browser and spend a great deal of time looking at that documentation and at code examples from other programmers to jog their memories and get inspiration on what to do next. They look at documentation even if they have seen it before, even if it was just yesterday.

Not only is there no shame in needing to consult documentation, there is generally no success without it. There is a way to do it without consulting email and cat pictures throughout the day, too, but that might be a bridge too far.

The point is that good programmers are not only good at typing commands into the computer, they are also good at finding what they need to know in documentation, support forums, and code examples. There are so many things to know about the libraries available for even one programming language that it is practically impossible to accurately remember even a useful fraction of it without assistance. Therefore, programmers get very good at looking at documentation and remembering roughly how to find things. If we had to remember every command and every facility in the language, we would simply fail to do anything.

A skill you will do well to cultivate as a programmer is this: recognize when it is likely that someone else has needed to do this before. If what you are about to do, or do not know how to do, is likely to have been done before, go find it. The internet is an amazing resource for programmers (stackoverflow.com⁹, for example, is an essential tool). Documentation is a crucial resource for programmers. Get good at searching and guessing keywords; your next program may depend on it.

All that said, if you are on a restricted network, or no network at all, you can absolutely finish **this course** without going online. There is plenty left to learn right here, and we will cover all

⁸<https://developer.mozilla.org/docs/Web/JavaScript>

⁹<https://stackoverflow.com>

of the facilities needed to successfully create small games.

Not that there is anything particularly special about games as such; they just happen to be great pedagogical tools because they exercise a wide swath of any programming language. There are lots of extremely useful programs that we could write, but it would be hard to find something else that is simultaneously small, interesting, and requires such broad use of a programming language.

With that out of the way, let's look at some useful standard library objects.

The Math Object

We have pointed out that you can add, subtract, multiply, and divide numbers. That's nice, but there is a whole lot more that you might want to do with them, like take their square root. For all but the most basic operations, you want the built-in `Math` object¹⁰.

There are a lot of functions available on the `Math` object, most of which you can view by evaluating `Math` in the console. Let's look at just a few of them:

:console:

```
> Math.abs(-20)
< 20
> Math.floor(5.62)
< 5
> Math.floor(-5.62)
< -6
> Math.ceil(3.1)
< 4
> Math.sqrt(144)
< 12
```

Here we have—in order—taken the *absolute value* (removed a negative sign if it exists), the *floor* (go down to the closest integer), the *ceiling* (go up to the closest integer), and the *square root* (no explanation required).

There are a lot of useful functions in here, but we will focus just on what we need in this course. Another very useful one is `Math.random()`, which takes no arguments and produces a random number between 0 and 1. We will cover that in greater detail later on, but it can be fun to play with if you are so inclined.

The String Object (and Prototypes)

The `String` object¹¹ is kind of like the `Math` object in that it provides some functions you can use, but it is also very different, in the sense that many of its functions are **prototype**

¹⁰https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math

¹¹https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String

functions. Let's take a look at what we mean by that.

If you look at the documentation for `String`, you might see in the margin that it has **properties** and **methods** like these:

- `String.length`
- `String.fromCharCode()`
- `String.fromCodePoint()`
- `String.prototype.toUpperCase()`
- `String.prototype.trimRight()`
- etc.

What is that `length` thing? Well, it turns out that every time you create a string, you are creating an object that has all of the stuff in it that the `String` object has, but *specific to your string*. In other words, when you type `"hello"` you are creating a `String` object with the characters "hello" in it, and with a `length` property equal to 5: the number of characters represented between, but of course not including, the delimiters (quote marks). Other strings will have other values specific to them. To get a feel for this, take a look at this console session:

:console:

```
> var s = "hi there"
> s.length
< 8
> "all on one line".length
< 15
```

See how we can access the `length` property on any string we choose? This can be done whether that string is pulled from a variable like `s` or from a literal like `"all on one line"`.

There are other things we can do to strings, as well. If you are able to look at the documentation for the `String` object online, you will see a very long list of methods and properties.

Since we have gained the superpower of logging things to the console without actually typing in the console itself, it's time to create a program. In fact, from now on, we will only rarely refer to the console for examples, because our examples are going to get a lot more involved. Typing a lot of involved stuff in the console is a pain. Use it for playing, use it for basic exploration, but once things get a little bit busy, use a friendly text editor to write a file, then load it. You can always log things using `console.log`.

If you can't, or don't want to use the browser console, a neat trick is to create your own logging function. It's a little bit advanced because it requires some things we haven't yet covered, but if you're willing to try it, here are some steps.

Step 1: Copy the following code into a file called `fakeconsole.js`:

:javascript:

```
document.addEventListener('DOMContentLoaded',
function() {
  var div = document.createElement('div');
  div.id = 'log';
  div.style.border = '1px solid black';
  div.style.position = 'absolute';
  div.style.bottom = '0px';
  div.appendChild(document.createTextNode('Log:'));
  div.appendChild(document.createElement('br'));

  document.body.appendChild(div);

  this.console = {
    log: function log() {
      var v = Array.from(arguments).join(' ');
      div.appendChild(document.createTextNode(v));
      div.appendChild(document.createElement('br'));
    },
  };
});
```

Step 2: In all of your HTML files throughout the rest of this course, add the following line above all other scripts, and close with `</body>`:

:html:

```
<body>
<script src="fakeconsole.js"></script>

<!-- Other stuff goes here. -->
</body>
```

Make sure that `fakeconsole.js` is in the same folder as your program file, and it should be able to find it.

Note that `<!-- -->` is a comment tag in HTML, in case you were wondering.

The body tags are important! Otherwise you won't be able to create the div and add it to the body, because it won't have been created, yet. It's kind of advanced, but if you save `fakeconsole.js` and then remember to add body tags around your programs, in addition to using the `<script src="fakeconsolejs"></script>` line, it will create a box at the bottom of the page and print log entries to it, even if you don't have console access.

If you haven't already, now might be a good time to create a special programming project that you always just use for examples. Then it will be ready to go whenever you want to type something in like this:

:html:

```
<script>
var s = "hi there";
console.log(s);
console.log("length of s:", s.length);
console.log(s.toUpperCase());
</script>
```

Without looking below just yet, what do you think this will output in the console when you load it?

Note that `console.log` (and our little fake replacement, if you don't have access to the console) can take any number of arguments, which can be useful if we want to know which log entry we are looking at without figuring out file line numbers. It outputs all of its arguments separated by spaces.

It is also worth reiterating that identifiers in JavaScript are *case-sensitive*. That means capitalization matters a great deal. The identifier `toLowerCase` is *not the same* as `toUpperCase`. To spell any identifier correctly, you have to get the cases correct.

Finally, note that `toUpperCase` is a function, which is why we call it to get a new value. The value `length`, on the other hand, is just a numeric value, not a function on a string. It acts something like a variable.

First of all, we are just outputting `s`, which is our “hi there” string. Then we output a line that tells us how long it is, which should be 8. But then we call a function using dot notation on `s` and output its result. This should mean we see the following output in the console:

:console:

```
hi there
length of s: 8
HI THERE
```

We have seen function calls before, but `s.toUpperCase()` is a little bit different. This function takes no arguments but still produces a value based on `s`. How does it do that? It can do that because it is a **prototype** function. When you see the word “prototype” in an object’s documentation, that means that the function knows about what is to the left of the dot when you call it. The value `s`, in this case, is an **implicit argument** to the `toUpperCase` function.

This is a gross oversimplification, and other functions (closures) can be made to act this way even if they are not defined on the prototype, but there is really no need to get technical about it right now. The simple explanation is going to be sufficient for a number of chapters.

Much of the JavaScript standard library is designed in this way, but the first and most obvious place we run into this is with `String`.

Let’s try adding a few more examples to our program:

:javascript:

```
console.log("    hey".trimLeft());  
console.log("WHAT?!".toLowerCase());
```

Here we are calling those methods on a brand new string using dot notation. This is just the same as doing it on a variable. The interpreter has to evaluate the thing on the left before it can find out what is inside of it, so all of the standard evaluation rules apply: get a value before doing anything else with it. In the case of our `trimLeft` call above, the interpreter follows these steps:

- Evaluate `console`, get an object
- Evaluate `log` in that object, get a function
- Cannot call it yet, need the value inside, so
- Evaluate `" hey"`, get a string
- Evaluate `trimLeft` in that string, get a function
- Call that function with no arguments
- Call `console.log` with the result

Remember, when things get a little muddy, as they might have done again in our most recent example, you can always peel them off a layer at a time by going through this exercise. Left to right, inside to outside, you can poke at the things you do understand, and then after not much time at all, you will realize that you understand the whole thing.

JavaScript, as is true with many programming languages, is fairly “regular”. This means that things are somewhat predictable once you understand the basic rules of evaluation. They apply everywhere (almost) equally, even when evaluations are buried inside of other evaluations like a nesting doll. In fact, the concept of evaluations containing other evaluations (or functions calling other functions, or statements containing other statements, or objects containing other objects, etc.) is called **nesting** because of exactly that analogy.

Drawing Pictures

It is finally time to do something other than pop up alert windows and log stuff to the console. It's time to draw some pictures!

The world of the browser is pretty rich with functionality. Browsers can display text and images, of course, but they can also play videos, do animations, run games, process audio, and many other interesting things.

The capability we are going to focus on for much of the rest of this course is the `canvas`. A canvas is basically what it sounds like: something we can draw pictures on. To get access to a canvas, we will need to create one using the current **document**.

Document

When you open a page in your browser, you are looking at a **document**. A document contains everything you can see and everything you can do with that page. It contains all of the text, all of the buttons, form elements, images, videos, and code that you are running. It is usually specified in HTML. You have actually been creating HTML documents this whole time.

The document that contains your code is referred to by the global variable `document` inside of that code. This `document` object contains other objects that contain other objects, etc. It is a nested structure, approximately a tree. At the root of the tree is this `document` object. The tree itself is called the “Document Object Model” (**DOM**). As you might imagine, the tree can get very big for complex documents, with lots of nested objects and long lists of text and formatting. Each object represents something that you can manipulate or get information from in your program.

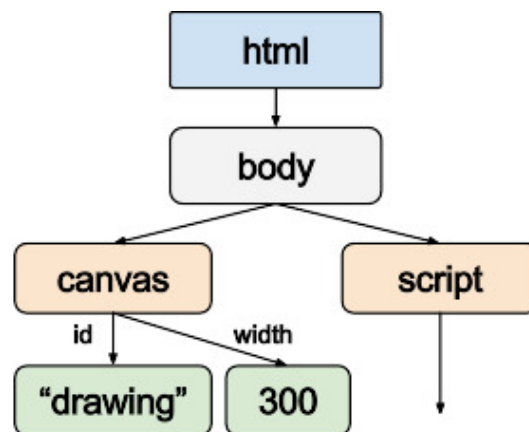


Figure 4.4: A document as a tree.

Because these trees can get really big (we say “deep” because we usually draw the root on top in computer science), the `document` object has a number of functions that you can use to find and manipulate various parts of the document, called DOM **elements** or **nodes**. This may all sound unfamiliar, but you have actually already run into a node. Consider a document containing typical “Hello, world!” program that logs to the console:

:html:

```

<script>
console.log("Hello, world!");
</script>

```

Or, if you don’t have console access and need to use our fake, above:

:html:

```

<body>
<script src="fakeconsole.js"></script>

```

```
<script>
console.log("Hello, world!");
</script>
</body>
```

This is a document. It contains a few hidden elements that we will talk about later, but right away you can see one of the elements: the `script` node. The script tags and the code within them are represented in an object in the DOM, and we can get at it using document methods (functions) like `document.getElementsByTagName('script')` (this returns a list of all of the script nodes in the document). There are similar functions for finding elements by other special attributes like “id” or “class”.

Canvas

Right now we want a document that contains a canvas element. We can just add one in our HTML. We will add it before our program code so that it exists before our code tries to find it:

:html:

```
<canvas id="drawing" width="400" height="300"
        style="border: 1px solid gray"></canvas>
<script>
var canvas = document.getElementById('drawing');
// Our other code goes here.
</script>
```

The width and height of the canvas are specified in pixels.

In case you don't know this already, the word “pixel” is short for “picture element”. Each pixel is a tiny rectangle of color, the smallest unit of drawing on a computer display, and the basis of our canvas's default coordinate system.

Also, note that spacing is not terribly important inside of a tag. Our `canvas` tag is split across multiple lines to make it look good in book format, but you could have done it on one line or many more lines.

Note that when we do this, the browser is filling in some things for us to be nice, but your browser may or may not be as easygoing as mine. If you find that things are not really showing up the way you expect them to, try putting in all of the missing HTML tags, like this (we will omit them in most examples for the sake of brevity, so if you have to add them here to make things look right, you will need to add them everywhere even when the examples do not say so directly):

:html:

```
<html>
<body>
<canvas id="drawing" width="400" height="300"
```

```
        style="border: 1px solid gray"></canvas>
<script>
var canvas = document.getElementById('drawing');
// Our other code goes here.
</script>
</body>
</html>
```

The browser is filling in these missing `html` and `body` tags for us because it found a `canvas` tag and knows that it needs to be in the body, but again, it might not work that way for you. If you have any doubt about this at all, just add in those missing tags and see if that gets you fixed up.

Also note: if you are using the fake `console.log` implementation described earlier, that goes immediately after the opening `<body>` tag. The first `<canvas>` tag would come after loading `fakeconsole.js`.

In our canvas, we are also giving it a 1-pixel solid gray border so that we can see that it is there. Please don't worry too much about the `style` attribute: we are not going to be getting into Cascading Style Sheets (CSS) in this course, so even though I dislike mystery in any serious curriculum, in this case it's a good idea to treat this as a magic incantation and just **copy the canvas tags when you need them elsewhere**. CSS is a language all by itself, and it is well beyond the scope of this book. If you want to get into it, please do! Find a good book on HTML and CSS and go nuts. Or get driven bananas by CSS rules. The latter is actually pretty likely.

First, let's talk briefly about the one line of code (not counting comments) found within:

:javascript:

```
var canvas = document.getElementById('drawing');
```

If you look closely at the canvas tag above, you will see we gave it some **attributes**. It has, for example, an attribute named `id` with the value `"drawing"`. The value can really be anything you like, but if you want to find it again, you need to use the same value in the `getElementById` function, like we did here.

Our JavaScript code is asking the document to find an element, any old element, with an attribute `id` equal to the string `"drawing"`. Since there is an object with that identity, the `getElementById` function finds it and the object representing that element is given back to us (our canvas!). We then keep track of it in the `canvas` variable. Again, there's nothing special about the variable name `canvas`. We could just as easily have called it `x` or `c`. It's our variable, after all, and we can name it whatever we want.

If you load your program right now, you will see the canvas with its border. Now it's time to draw something onto it.

To draw on a canvas, we need a **context**¹². This is where we set colors and line widths, and is also what we use to draw on the canvas. You can think of it as your brush and paint set. We aren't getting into 3D modeling (WebGL) in this course, so we ask for a 2-dimensional canvas using `getContext('2d')`. Then we set some colors and draw a rectangle with those colors. Here's the new code, so copy it into your HTML program file and save it (though you can skip the instructional comments if you want to):

:html:

```
<canvas id="drawing" width="300" height="200"
        style="border: 1px solid gray"></canvas>
<script>
// Find the canvas element.
var canvas = document.getElementById('drawing');

// Get the context so we can draw.
var context = canvas.getContext('2d');

// Set the current fill color.
context.fillStyle = 'red';

// Draw a rectangle with the current color.
context.fillRect(10, 10, 30, 30);
</script>
```



Figure 4.5: Program output: one red rectangle.

If you run this and it doesn't work, don't forget to open the console to look for error messages!

¹²<https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D>

There is quite a bit of new stuff going on in there, so let's do what we usually do: take things one step at a time and make sure that we understand before moving on.

First, we get a canvas object from the document by searching for its ID. That part is familiar. The ID in `getElementById` matches the one we set in the canvas tag, so it finds the canvas successfully. Then, we ask the canvas for a context by calling `getContext('2d')`.

Remember, case matters, so we have to spell the function precisely as `getElementById`, no matter how much it might offend our sensibilities.

After all, we're not searching for the element by invoking its Freudian "id". We're not savages.

As mentioned earlier, the context¹³ holds the current status of your drawing tools. If you set `fillStyle` to a particular color, that is like dipping your brush in that color. Then, when you actually draw something with it, that color is what ends up on the canvas. It can do more than that, like keeping track of how much your canvas is stretched or rotated, but we will stick with colors and line styles here. You can find out about everything you can do with a canvas context by looking at the documentation online¹⁴.

Since we have not applied any stretching or rotating transformations, the coordinate system of the canvas is pretty straightforward: the upper left corner is pixel (0, 0) and the lower right corner is (width, height), or (400, 300) in our case. That's why the red rectangle shows up near the upper left corner: it has a corner at (10, 10) and extends 30 pixels to the right (the rectangle width) and 30 pixels down (the rectangle height) from there.

As an aside, note that the coordinates of the canvas might not be exactly what you expect. An x coordinate of 0 means "exactly on the left edge of the canvas", and the "width" coordinate is "exactly on the right edge of the canvas". Similar logic applies to the extremes in the y direction.

So, that means that integer coordinates are *in between* pixels; the actual pixels are the grid squares, and integers occur at the intersections. If, therefore, you try to draw a horizontal line 1 pixel wide from, say, (1, 1) to (5, 1), the line will actually straddle pixel boundaries as shown in the figure. By the way, fractional coordinates like (1.5, 1.5) are allowed; those would point to the center of a pixel instead of the boundary.

For integer coordinates this means that lines drawn at the extreme edges are only *half shown*, and every other line looks fuzzy (the computer tries to make it look like it is only lighting up half a pixel by fiddling with the color). It is therefore not uncommon to shift the whole canvas by half a pixel in each direction, putting the integer coordinates at the center of the pixels instead of on their boundaries, making lines look crisper. We won't worry about that until a bit later;

¹³<https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D>

¹⁴<https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D>

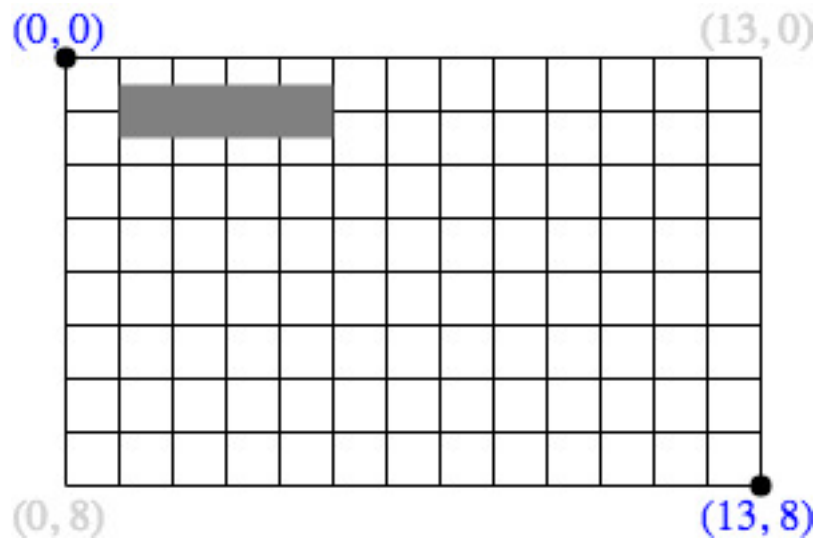


Figure 4.6: A line straddles pixels by default.

the point was to know how the canvas coordinate system works: the coordinates are on the *boundaries* of the pixels, and the pixels are the cells, and each cell can only be exactly one color.

With that out of the way, let's try adding some more rectangles with different colors and positions, just for fun, just inside our script tags (remember, there should be a canvas tag first, then script tags containing this code, it is just omitted for the example here):

:html:

```
<canvas id="drawing" width="300" height="200"
  style="border: 1px solid gray"></canvas>
<script>
var canvas = document.getElementById('drawing');
var context = canvas.getContext('2d');

context.fillStyle = 'red';
context.fillRect(10, 10, 30, 30);

context.fillStyle = 'blue';
context.fillRect(50, 20, 50, 40);

context.fillStyle = 'green';
context.fillRect(240, 150, 10, 20);

// Still green.
context.fillRect(65, 15, 14, 16);
</script>
```

Did you see how we told the context what the current color was, then told it where to put a

rectangle? It used the most recent color each time. If you add more rectangles without changing the `fillStyle` property, it will continue with the last color used.

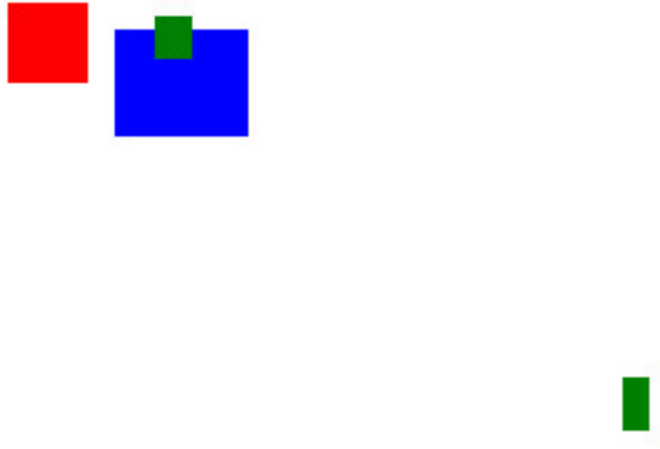


Figure 4.7: Program output: four colored rectangles.

That's the basic principle of the context: you set things up the way you want them, then you draw stuff under those conditions. We have seen color (well, `fillStyle`, really, that happens to be a solid color because that's what we told it to use) as one type of condition, and there are others, but the principle remains the same. Whatever you draw in a context uses the current setup in that context, and the setup only changes when you tell it to. That's how the canvas works.

Congratulations! You have just painted on a canvas that you found by ID, and you changed colors and learned about its coordinate system in the process. You can play around with all of it and see what happens.

Listing

A complete listing of our final program is here. All of it goes into a file like `myprogram.html`, and the `<html>` and `<body>` tags are explicitly shown (though many browsers do not need them for small programs shaped like this one):

:html:

```
<html>
<body>
<canvas id="drawing" width="300" height="200"
        style="border: 1px solid gray"></canvas>
<script>
var canvas = document.getElementById('drawing');
```

```
var context = canvas.getContext('2d');

context.fillStyle = 'red';
context.fillRect(10, 10, 30, 30);

context.fillStyle = 'blue';
context.fillRect(50, 20, 50, 40);

context.fillStyle = 'green';
context.fillRect(240, 250, 10, 20);

// Still green.
context.fillRect(65, 15, 4, 8);
</script>
</body>
</html>
```

Exercises

Exercise 4-1: Object Creation

Solution on page 383

Show how to create **objects** with the following characteristics:

1. Empty
2. One property with key "name" and the value being your own name (or you can make the value just be the string "me").
3. One property with a key that is a number, like 5, and a value of anything you like.
4. The following properties, with string values that correspond to your favorite book:
 - "Title"
 - "Author"
 - "Year Published"

Exercise 4-2: Object Property Retrieval (and Setting)

Solution on page 384

Show how to get (or set) values in an object stored in the variable `obj` for properties with the following names:

1. "title"
2. 10
3. "something amazing"

Do any of these allow you to use dot notation? Why can you not use dot notation on the others?

Exercise 4-3: Non-existent Object Keys

Solution on page 384

What happens if you try to get a property out of an object that does not exist? What value do you get instead?

Exercise 4-4: Deleting Object Properties

Solution on page 385

How do you delete a property from an object? Hint: you cannot just assign `undefined` to it.

Exercise 4-5: Logging

Solution on page 385

How do you log something to the console from code in a file (as opposed to just entering it directly in the console)? How do you log *multiple* things at once?

Exercise 4-6: Standard Library

Solution on page 386

What is a language's "standard library"?

Exercise 4-7: Math Practice

Solution on page 386

Without the aid of the computer, write down how you would compute a few things using the `Math` library:

1. 2^{12}
2. $\sqrt{3}$
3. $|-56|$
4. $\lfloor 6.8 \rfloor$
5. $\lceil 2.3 \rceil$

Note that $|x|$ indicates the "absolute value" (positive value of x), $\lfloor x \rfloor$ indicates the "floor" (nearest integer at or below x), and $\lceil x \rceil$ indicates the "ceiling" (nearest integer at or above x).

Exercise 4-8: String Practice

Solution on page 387

Strings have a bunch of standard methods built in, as well as the standard `length` property. Solve this without the aid of a computer: given the string `s = " This string is mine. "`, what will the following methods produce? Hint: the spaces at the ends of it are part of the string because they are within the quote delimiters.

1. `s.length`
2. `s.toUpperCase()`

3. `s.toLowerCase()`
4. `s.trim()`
5. Bonus: `s.toLowerCase().trim()`

Exercise 4-9: Canvas Practice

Solution on page 388

Write a program that draws three squares (using `fillRect`) on a 300-by-300-pixel canvas. The squares will have the following properties:

- Each square is 100 pixels on a side,
- Each is a different color, one of red, yellow, and green.
- The first square is in the upper left corner, the second is in the middle, and the third is in the lower right corner.

Chapter 5

Reacting Logically

With the ability to draw on a canvas and respond to events like the passage of time, learning more of the language just got a lot more interesting. In this chapter we will be covering two things that are deceptively simple and secretly awesome: the values `true` and `false`.

And we will do it by drawing pictures. But first, math!

Boolean Logic

This section is pretty useful as background reading, and there will be some exercises on the basic concepts in here, but you shouldn't worry too much if it gets a bit deeper than you are ready for. Programmers should be good at logic, and you should at least be exposed to the ideas and techniques in this section as you get started, but you shouldn't expect complete mastery right away. It is fascinating, useful stuff, and we will be practicing it as needed throughout the course. It's not a fire-and-forget kind of thing. Consider this a first pass, if you aren't familiar with the material. It will come up again, with review!

One of the most fundamental things you can learn when working with computers is Boolean logic and algebra: the mathematics of **truth values**. You can think of it as a number system with only two digits: 1 and 0, or `true` and `false`. It is not exaggerating to say that the logic of truth values lies at the foundation of all modern digital computing. That is why you might often hear, "It's all just ones and zeros": it really is. Everything a computer does is ultimately about manipulating those two things.

Well, everything a *classical* computer does is about manipulating discrete values like that. Quantum computers do something else, entirely. We will definitely not be talking about how those work in this course, but it is a really interesting topic with what turns out to be surprisingly accessible mathematics behind it. We're hearing more and more about them lately, and that's exciting, but for this course we'll stick with classical computing.

The number system consisting of only the digits 0 and 1 is called the **binary** number system. While we will indeed talk more about the binary number system, for now we are going to stick to the two values `false` and `true`, and we will avoid thinking of them as being equivalent in some way to 0 and 1. Stay tuned, though, because we will eventually get back to binary.

George Boole

George Boole¹ published *The Laws of Thought* in 1854, a book that contains “Boolean Algebra”, and one that I want on my bookshelf, though it is not necessarily light reading. His ground-breaking mathematical approach to logic underlies how we reason about computation and forms the basis for how most computers work internally. Thus, when we speak of “Boolean Logic” or “Boolean Algebra”, we are speaking of the work that George Boole did in the 1800s. The work is important enough that, even though the concept it refers to is as generic as the term “truth”, “indicator”, or even “bit” (short for “binary digit”), we see some form of his name all over the place in computer programs as a “boolean” type, sometimes shortened to “bool”.

Boolean Types and Truth Values

The values `true` and `false` are fairly special. They aren’t numbers, exactly, though in working with digital computers we often think of them as 1 and 0, respectively. They aren’t names or places. They aren’t strings or images. They are just the values `true` and `false`, and nothing else. In computer science, we say that these “truth values” are of *type* “boolean”. That is, “boolean” is the kind of thing that they are, just like “number” is the kind of thing that 10 is and “string” is the kind of thing that “is” is.

You might say that they are *isomorphic* to the numbers 1 and 0 in certain contexts, so it’s valid to think of them as being basically the same thing.

Also, since we are introducing a new **type** here, let’s talk a bit more about types in general.

Whenever you encounter a new **type** in computer science, you can think of it as describing a *set of things*. For the “boolean” type, for example, you can think of it as the set containing `true` and `false`, and nothing else.

Sometimes the number of things in a type’s set is infinite (like “all integers”), but that’s only in theory, not in the reality of the machine sitting in front of you. In practice there is *always* a finite limit, however large, of the number of things of a certain type that can be represented on a real system. The type “boolean” is a nice place to start, though, because it can only have two values. What are they, again? Oh yes, `true` and `false`.

You know, just in case you forgot.

¹https://en.wikipedia.org/wiki/George_Boole

Booleans in Familiar Places

Other than saying `true` or `false` directly, how can we get a boolean value, and what can we do with it once we have it? It turns out that with a background in basic algebra, you have already had frequent encounters with truth values, but might not have thought about them in quite this way.

To really see this, let's look for a moment at a simple equation:

$$y = -x$$

If you know x , you can find y , and that is probably the most common thing to do with an equation like this in pre-algebra. But what if you already know both x and y ? Suppose you know, for example, that $x = 4$ and $y = -4$. What can you then say about $y = -x$?

We can say that it is `true`, because $-4 = -4$.

Suppose instead that $x = 4$ and $y = -3$. Now what? In this case, the equation is `false` because $-3 = -4$ is not a true statement (not an equation).

And there we have it. The **relational operators** in algebra, such as $=$, give us **truth values**. Equations are true or false depending on how you set the variables. If you only set some of the variables, sometimes your task is to find values for the rest that make the equation true, but ultimately it's all about whether the equation is a true or false statement.

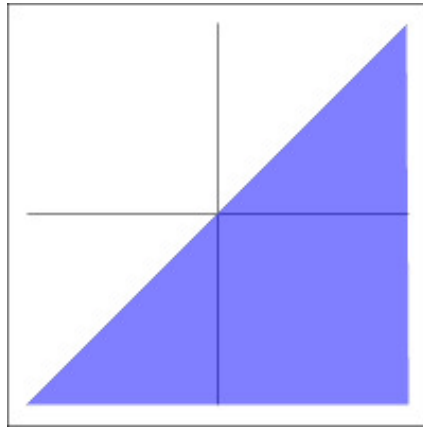
Let's try another relational operator to go over this concept again:

$$y \leq x$$

This is something we usually think about intuitively: if I say that $x = 3$, then you know that y can be any value less than or equal to 3. Why do you know that? Because if I told you $x = 3$ and $y = 4$, then $y \leq x$ would no longer be true. For what values, exactly, is the expression true? One way to find out is to plot a graph of it. To do that, you look at every possible value of y and x and color in the parts that make $y \leq x$ a true expression, as shown in the figure.

Remember, every point on the graph represents a unique pair of values for x and y . Everywhere that is colored in on the graph represents values for x and y that make $y \leq x$ a true statement. Everywhere else, the statement is false.

Of course, we can't graph *every* value, so we usually just pick values of x that show either a representative trend, or a particularly interesting part of the function. In the case

Figure 5.1: $y \leq x$

of a line we often just need to show its slope and intercept, so that's usually the part that gets graphed.

That's pretty neat, really! We just took a simple algebraic inequality and turned it into a mathematical question whose answer is either true or false. In other words, we started with some numbers and ended up with a boolean value! Hopefully when you look at things this way, where every expression has a value, including inequalities and equations, you begin to realize that you have been working with booleans for a long time already.

Are we good so far? If not, this is a good time to take a breather and re-read the previous section, because this next part is where we take a critical leap. Meanwhile, here is the basic summary:

- Boolean values are just `true` and `false`, sometimes written down as 1 and 0, depending on the context.
- Whenever we compute a mathematical **relation** like “less than” or “equal”, the result of that relation is either `true` or `false`.
- Graphing things is basically the process of marking the areas where variable settings make an expression `true`.

Truth Values Are Just Values

Boolean values are like numbers in many ways, and we are going to explore some of those in this section. You can use them in equations, you can compare them, and there are operators that let you combine them. Here is an example of how you might use a boolean value a in an equation:

$$a = (5 < 7)$$

This might look strange to you. Hopefully you are familiar with things like $5 < 7$ and $a = b$, but combinations of those concepts, as shown here, are not often part of an early algebra curriculum. They turn up everywhere in computer programming, though.

One way to get comfortable with a new expression like this is to start picking it apart and figuring out what all of the types are. We will work from the inside out, just like you do in algebra. If we do that, we end up tackling things in roughly this order:

- 5: number
- 7: number
- $(5 < 7)$: boolean expression (true, in this case)
- a : free variable

Now we can have some hope of understanding this equation. It is saying that a is equal to the boolean value corresponding to $5 < 7$, which we know is true. Therefore a must be equal to the value true, otherwise the equation itself would be wrong (false).

This last point is important, too: if you already have $a = \text{false}$, then the entire thing is false. That's interesting, even an equation like $a = (5 < 7)$ evaluates to a boolean. In other words, when $a = \text{false}$, you might read the equation as "It is false that $a = (5 < 7)$," or, after simplifying, "It is false that false = true.

Equations are just statements about how things are related, and it's possible to set things up so that they satisfy that relationship (make it true) or so that they don't (they make it false).

In this form of algebra, there really is no such thing as variable "assignment" (crucially, we are talking about *algebra* and not *programming*, here). There are only declarations about what is true. When we say $x = 5$, that does not somehow create a slot in the universe named x and stick the number 5 into it like it would in one of our programs. Instead, it makes a statement about what is true and what is false. It is true that $x = 5$ and it is false that x equals something else.

As a teaser for what comes next, consider the following statement: $y < x$ AND $y > -x$. How would you shade that graph? Hint: "AND" is an operator that you apply to boolean values, and its result is only true when both sides are true at the same time. Let that sink in for a bit, and then we will jump into the next section.

Boolean Operations and Truth Tables

What can we do with truth values? It turns out that we can do quite a bit! In fact, there is an entire algebraic system developed around these values. Most folks learn a little bit about these rules of logic as they learn to process written language, but the conclusions drawn tend to be error-prone and even subjective, both of which are problems when programming because computers are dumb and require precise explanations of what you want. It is far easier to get these things right when you can write them down and manipulate them algebraically than when they are a jumble of words. Consider this statement, for example:

It is not true that I am not wearing short sleeves and it is also not raining.

That's sort of tricky, right? I mean, what we really want to say is more straightforward, like

It is either raining or I am wearing short sleeves (or both).

Did you know that those two statements are equivalent? They are! But it's hard to tell for two reasons:

- The first sentence is ambiguous: does the “it is not true” apply to the first half or the whole thing? In our case it applies to the whole thing, but the structure does not instill confidence.
- The language is generally unwieldy and imprecise.

This is a teaser, so try not to get hung up on the notation, but here is an example of how to mechanically morph the above statements into each other using Boolean Algebra concepts of double negatives and De Morgan's Laws:

$$\begin{aligned}
 \text{statement} &= \neg(\neg\text{short sleeves} \wedge \neg\text{raining}) \\
 &= \neg\neg\text{short sleeves} \vee \neg\neg\text{raining} \\
 &= \text{short sleeves} \vee \text{raining}
 \end{aligned}$$

Mathematical notation has its perks. Let's learn more about that notation now.

We are already familiar with common numeric operators like these:

Symbol	Formal Name
·	Multiplication
/	Division
+	Addition
−	Subtraction

It turns out that boolean values have their own similar operators:

Symbol	Common Name	Formal Name
¬	NOT	Negation
∧	AND	Conjunction
∨	OR	Disjunction

There are others, like the “implication arrow” \implies , but we will be sticking with just these. Implication can be expressed in terms of these, anyway.

Note that \wedge looks like a capital A with the middle line missing. That can be a good way to remember that \wedge is just AND.

While we can describe what these do in human language, and we most certainly will, it is clearest to express them as truth tables. A **truth table** is just a listing of all the possible operands (“inputs”) to their corresponding expression values (“outputs”).

The simplest of the operators is the unary negation operator \neg . This basically inverts the truth value given to it:

$$\begin{aligned}\neg \text{false} &= \text{true} \\ \neg \text{true} &= \text{false}\end{aligned}$$

That’s the entire table. Since negation only takes one value (much like negation in numeric arithmetic), there are only two possibilities: either it is negating the value false or it is negating the value true. As you might expect from an operator called “not”, $\neg A$ is true if A is not true, and $\neg A$ is false if A is not false. In other words, $\neg A$ is read as “NOT A”.

A **truth table** is really a completely enumerated function definition, consisting of the entire domain-to-range mapping. The connections of computer science, boolean algebra, and even number systems to functions is extremely deep, and Lambda Calculus forms a powerful underlying framework for understanding and expressing these connections. If, you know, you like that sort of thing. I know I do.

The other two operators, \wedge and \vee , are binary operators analogous to multiplication and addition, so they have larger tables. Each entry in the table represents one of the possible situations in which they are used. Let’s start with conjunction (“AND”, written \wedge):

$$\begin{aligned}\text{false} \wedge \text{false} &= \text{false} \\ \text{false} \wedge \text{true} &= \text{false} \\ \text{true} \wedge \text{false} &= \text{false} \\ \text{true} \wedge \text{true} &= \text{true}\end{aligned}$$

Here we have four entries, because there are four possible situations where you can see \wedge : both sides false, both sides true, and one entry each for the sides being different. As you might expect from an operator known as “AND”, a conjunction expression is only true when

both the left **and** the right side is true, and is false in all other cases. We therefore read $A \wedge B$ as “A AND B”, meaning that the statement is true if both A and B are true.

Let’s see what happens with disjunction (“OR”, written \vee):

$$\text{false} \vee \text{false} = \text{false}$$

$$\text{false} \vee \text{true} = \text{true}$$

$$\text{true} \vee \text{false} = \text{true}$$

$$\text{true} \vee \text{true} = \text{true}$$

In this case, as you might well expect from an operator called “OR”, the disjunction expression is true if either the left **or** the right side is true, and is only false when both sides are false. It is important to remember that it is also true when both sides are true, which is different than the way we sometimes say “or” to mean “only one or the other” in English. Disjunction $A \vee B$ is therefore read as “A OR B”, and is true if at least one of A or B is true. The only time $A \vee B$ is false is if both values are false.

Truth tables are extremely useful as shown, but they are often written in shorthand, more like a table than what we have seen so far. Let’s look at the more compact form for the three operators above, and we will shorten true to T and false to F at the same time:

A	$\neg A$
F	T
T	F

A	B	$A \wedge B$
F	F	F
F	T	F
T	F	F
T	T	T

A	B	$A \vee B$
F	F	F
F	T	T
T	F	T
T	T	T

Because the expressions can get more complex than just $A \wedge B$, this format shows the expression once in the column header, with the values below it. This is the more common approach to truth tables, and we will be using it as we go on.

Be aware that common mathematical notation for true is \top (top), while the notation for false is \perp (bottom). We will stick to the letter versions of these because they are more familiar, but it is good to be aware of the standard notation in case you see it elsewhere. We won't be using this for programming.

A	B	$A \vee B$
\perp	\perp	\perp
\perp	\top	\top
\top	\perp	\top
\top	\top	\top

Truth Table Construction and Binary Numbers

To make sure you cover all possible cases in a truth table, you basically count in binary, using one bit per variable. If that doesn't sound familiar, great! We will go over it in detail here. If it does sound familiar, feel free to skip to the next section.

To count in binary, we start with a single row of values for our variables, where they are all initially false. Then we go through a simple procedure to create the row beneath it (and repeat the procedure to create all the rest of the rows).

If you already have a row, then you can create the next row in the sequence as follows:

- Copy the bottom row. Congratulations, you now have a new bottom row.
- Flip the *rightmost variable* in the new row (F goes to T , and T goes to F).
- Stop if you just changed it to T .

There is one last little rule:

- If you ever change anything to F , then the one to the left of it must also be flipped.

When you end up with a row where all the values are T , you are done.

Here is an illustration of how that works with three variables. We begin with them all false, as described, and we will show every changed variable in **bold**:

A	B	C
F	F	F

Then, we copy this row and flip the new row's rightmost variable, giving us this:

<i>A</i>	<i>B</i>	<i>C</i>
F	F	F
F	F	T

Nothing went from true to false, so we are done with that row.

To create the next row, just do the whole thing again: copy and flip. Note, though, that this time the rightmost variable will go from true to false, so we need to flip the one to its left:

<i>A</i>	<i>B</i>	<i>C</i>
F	F	F
F	F	T
F	T	F

So far so good. Now we create the next row by doing it again, flipping the one on the right and checking to see if one left of it needs to flip. Not this time, it turns out (nothing is switching from true to false):

<i>A</i>	<i>B</i>	<i>C</i>
F	F	F
F	F	T
F	T	F
F	T	T

Now we get to a row that is really interesting. We will flip the rightmost variable like we always do, but because it is going from true to false, the one to the left of it has to flip as well. But that one is also switching to false, so the next one over must switch, as well:

<i>A</i>	<i>B</i>	<i>C</i>
F	F	F
F	F	T
F	T	F
F	T	T
T	F	F

And now, if you continue the pattern, you will see pretty quickly how you can get to this complete table, starting with all values false, and ending with all values true:

<i>A</i>	<i>B</i>	<i>C</i>
F	F	F
F	F	T
F	T	F
F	T	T
T	F	F
T	F	T
T	T	F
T	T	T

That is how you create the inputs for a truth table. Because you are an astute student, you might have noticed some other patterns in the completed table, like the fact that the left side has 2 blocks of similar values (4 false and 4 true, in that order), the middle alternates twice as often, and the right side alternates twice as often as the middle (it alternates every time, in fact).

There are other things to notice, as well. For example, if you have N variables, you will always have 2^N rows. That is why we have 8 rows for 3 variables: $2^3 = 8$. And since we know we have 8 rows, we could have just written 4 Fs and 4 Ts on the left, then groups of 2, then groups of 1. It is kind of cool once you start seeing how it works.

That is how counting in binary works, by the way. If you use 1 instead of T and 0 instead of F , you get this:

000
001
010
011
100
101
110
111

Those are the first 8 binary integers, starting at zero!

It is worth noting that this is exactly the same procedure that we use to count in decimal (base 10), we just have fewer digits to work with in binary. Try it in base 10: start with all zeros, count up the rightmost digit, then when it flips from nine back to zero, increment (add 1 to) the one to the left. That's how we count in decimal, and it's also how we count in binary.

There is a relationship here between AND and multiplication, and OR and addition. They obviously aren't perfectly the same: if you draw the AND and OR truth tables using multiplication and addition, you'll get some strangeness with OR (in binary, "10" means

“two”):

A	B	$A \vee B$	$A + B$
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	10

It's sort of interesting, but *more importantly*, AND and OR behave similarly to multiplication and addition *when considering the distributive property*. So, if you are trying to distribute AND over OR, it works like distributing multiplication over addition. Almost. We'll get to that with De Morgan's Laws.

Basic Boolean Algebra

As is the case with familiar numerical operators, the boolean operators can be combined in all sorts of ways to form complex expressions. The first thing to note is that parentheses work exactly the same way as they do with familiar expressions: if something is in brackets or parentheses, it needs to be considered a single expression. Always evaluate what's in parentheses first.

Operator Precedence

In the absence of parentheses, there are certain rules for which things are evaluated first; the operations performed first have **precedence**. This should feel familiar for numbers, where we know, for example, that multiplication takes precedence over addition. It is also true of booleans. When you are calculating something, operations with higher precedence come first. For example, if we just list a few common numerical operators, we might put them in this order:

- brackets
- exponentiation (and roots)
- negation
- multiplication (and division)
- addition (and subtraction)

To illustrate, suppose you saw something like this:

$$y = 3x + 2$$

You would know that x is multiplied by 3 before 2 is added to obtain y . That is true (and it affects the way we write things down) because multiplication has higher precedence than addition. But brackets beat everything, so this is a different equation entirely:

$$y = 3(x + 2)$$

Here 2 is added to x before multiplying all of it by 3 to obtain y .

Feeling comfortable again? Good. Boolean logic has precedence rules similar to those you are used to:

- brackets
- negation (NOT)
- conjunction (AND)
- disjunction (OR)

So, if you saw this

$$Z = A \wedge B \vee C$$

you would know that it is the same as this

$$Z = (A \wedge B) \vee C$$

and different from this

$$Z = A \wedge (B \vee C)$$

because conjunction has higher precedence than disjunction: “AND” comes before “OR” just like multiplication comes before addition.

One thing that is important to know is this: **numeric comparators have precedence over boolean operators**. That means, for example, that $3 < 5 \wedge 4 > 3$ is the same as $(3 < 5) \wedge (4 > 3)$.

Commutativity, Associativity, Distributivity, and De Morgan’s Laws

Conjunction and disjunction are both **commutative**: it doesn’t matter if you flip the arguments around, so

$$A \wedge B = B \wedge A$$

and

$$A \vee B = B \vee A .$$

That seems pretty obvious if you look at their truth tables. They are defined by how many of their arguments are true or false, not by which side they are on.

Similarly, they are both **associative**. This means that

$$(A \wedge B) \wedge C = A \wedge (B \wedge C)$$

and the same is true for disjunction. In practice this means it doesn't matter which things you combine first, so long as the operators are the same.

That is also true of addition and multiplication:

$$2 + (4 + 6) = (2 + 4) + 6$$

so you can just write

$$2 + 4 + 6$$

without getting confused. The same applies to conjunction and disjunction: you can omit the brackets in associative cases like

$$A \wedge B \wedge C$$

or

$$A \vee B \vee C .$$

And just like multiplication distributes over addition, conjunction (AND) distributes over disjunction (OR). Consider this:

$$Z = A \wedge (B \vee C)$$

This has a conjunction (AND) outside, and a disjunction (OR) inside. You can distribute in the same way you would if this were multiplication and addition, giving you this:

$$\begin{aligned} Z &= A \wedge (B \vee C) \\ &= (A \wedge B) \vee (A \wedge C) \end{aligned}$$

Something odd happens when you try to distribute a negation, though. It causes everything inside to be negated, but it also causes disjunctions and conjunctions to trade places; they get flipped upside-down. This is called De Morgan's Law, and it's very useful. Sometimes complex truth expressions can be greatly simplified and clarified in your programs if you know this rule:

$$\begin{aligned} \neg(A \wedge B) &= \neg A \vee \neg B \\ \neg(A \vee B) &= \neg A \wedge \neg B \end{aligned}$$

See how the operators flipped over when distributing negation? You can see that this actually works by looking at the truth tables carefully:

<i>A</i>	<i>B</i>	$\neg(A \vee B)$	$\neg A \wedge \neg B$
F	F	T	T
F	T	F	F
T	F	F	F
T	T	F	F

<i>A</i>	<i>B</i>	$\neg(A \wedge B)$	$\neg A \vee \neg B$
F	F	T	T
F	T	T	T
T	F	T	T
T	T	F	F

You can also talk your way through it. For example, if it is not both raining and windy right now, that means it is either not rainy or not windy. See if you can come up with some other

examples of your own. It always works.

In fact, we kind of got a glimpse of this when talking about what AND and OR really do: AND produces true only when both inputs are true, but OR produces false only when both inputs are false. That is an interesting parallel! And it is very suggestive of De Morgan's Laws.

Conditional Expressions

With this basic introduction to boolean logic, we should be more than ready to start putting these ideas to work in a program. Get a blank project file ready to go, load it into your browser, and let's begin.

Equal Isn't Equal

In JavaScript, as we have already seen, the symbol `=` means **assignment**. The thing on the left has to be something assignable like a variable or an object property. The thing on the right has to be an evaluable expression. Thus, when we see

:javascript:

```
var x = 5;
```

that means “create a slot in our programming universe, name it `x`, and stick the value 5 into it.” Given that `=` already has this meaning, how do we compare two things to check for equality?

For that, we use `===`. That is right, three `=` symbols in a row produces `true` if both sides are equal, and `false` if they are not. Just like there is a symbol for “equal”, there is also a symbol for “not equal”, which is `!==`:

Symbol	Meaning
<code>===</code>	Are both sides equal?
<code>!==</code>	Are the two sides different?

JavaScript, sometimes you hurt those you love. Seriously. The operators `==` and `!=` could have been perfectly reasonable as in many other programming languages, but they just do too many surprising things for me to recommend their use here. For example, `5 == "5"` is `true`. Why? Because, dear JavaScript, `==` does not compare types, only values, and you try very hard to make them the same type before doing the comparison. We know you mean well. We know you want to do what we mean and not what we say, to make our lives easier. It just turns out to be so error-prone! At least you give us the `===` operator, which compares both types and values, and is therefore almost always what we

want.

Try them out in the console:

:console:

```
> 5 === 5
< true
> 5 === "5"
< false
> "hi" !== "hello"
< true
> 5 === 4
< false
> 2 === "hi"
< false
```

Load that up in your browser and see what happens when you look at the console. You should see several truth values in the right order. Play around with it a bit. You can try testing equality for all sorts of things.

Remember, once you have the file loaded in your browser, you can just keep editing it, saving it, and then reloading the page. You don't need to go find it every time because the browser already knows where it is (the location is in the URL bar).

The operators `===` and `!==` are useful for pretty much any type. For numbers, though, we also have the standard comparison operators, shown here with their mathematical counterparts:

Algebra	JavaScript
$<$	<code><</code>
\leq	<code><=</code>
$>$	<code>></code>
\geq	<code>>=</code>

Strings are also comparable using the above operators. When applied to strings, they do a **lexicographical comparison**, loosely meaning that `<` tells you whether the string on the left comes before the string on the right, if you were to look them up in an alphabetized dictionary. Beware, however, that alphabetization may not work the way you think. It has more to do with the numeric value of the characters involved than with any particular notion you have about order. For example, using the English alphabet represented in ASCII or any of the UTF variants (JavaScript uses UTF-16 internally), capitals always come before lower case, which is not likely the way that you are used to thinking of things.

Boolean Operators for Typists

Modern programming languages have their roots in a time where only a limited character set was available to computers, called ASCII. Nowadays, we have access to all of unicode, which contains pretty much every character you can dream of in any language, plus a whole bunch of—incredibly useful, I’m sure—emoji. But even with our nearly unlimited character sets, keyboards are still of finite size, so we still mostly program using the original ASCII, at least for fundamental programming constructs like built-in operators. Now that we have seen how to type common comparison operators, let’s see what the boolean operators look like in JavaScript:

Name	Algebra	JavaScript
NOT	\neg	!
AND	\wedge	& &
OR	\vee	

Thus we have ! (exclamation point, or “bang”) for “not”, & & (two ampersands) for “and”, and | | (two vertical pipe characters, usually found on the backslash key) for “or”. We can play around with them in the console like this:

```
:console:
> true && true
< true
> true || false
< true
> !true
< false
```

Go ahead and play around with them a bit. Because the regular math operators take precedence over all of these (just as they do in algebra), you can also do things like this:

```
:console:
> 5 < 6 && 3 < 2
< false
```

Pretty neat! We can do all of the basic boolean operations in JavaScript, and they evaluate to the same kinds of things you would expect: the values `true` and `false`, with type “boolean”. With truth values like these, we can start to think about conditional computation, which is one of the fundamental things that makes computers work at all.

Technically, it is possible to represent conditional evaluation using nothing more than functions. It’s kind of a strange thing to think about, but you can represent the ideas of “true” and “false” as functions, you can represent numbers as functions, and then of course, you have functions. With all of that, you can define computing as we know it

(though Category Theory layers a thing or two on top). That's what Lambda Calculus is all about. Theoreticians like boiling things down to their absolute essentials, and for computation, it turns out that functions are absolutely essential, but not much else is.

It's much easier to just have native boolean types and operators, however, so that's what people actually use, even in purely functional languages like Haskell or Racket. Programming is a fundamentally human endeavor, and syntax matters a lot.

Ternary Conditional Expressions

Now it's time to create a program in a file. It has been a little while since we did that, so recall the steps:

- Open a new file in your text editor,
- Write code into it and save it as something with a `.html` extension, and
- Open it in your browser.

Remember that what we are actually writing is HTML that contains `script` tags, where we put our JavaScript code. With that in mind, let's create a simple number-guessing game. We know how to get input using `prompt`, and how to `alert` values to the user, so we make use of those functions here:

:html:

```
<script>
// What we're thinking of. Don't tell.
var answer = 6;
var guess = prompt("Guess the number I'm thinking of:");
alert(answer === +guess);
</script>
```

Here we see our familiar unary plus, to ensure that we are working with a number. That's why we say `+guess` above. It ensures that our `===` operator will be comparing two numbers instead of a number and a string (`prompt` returns a string). If not, we would get `6 === "6"`, which is false because an integer cannot `===` a string.

We *could* just use `==` instead, which is more forgiving about types (more coercive, really), but it is a bad practice as programs grow larger, so we will not reinforce that particularly dangerous habit. It is usually better to do conversions explicitly instead of hiding them within the details of how overly permissive operators work.

Furthermore, using `==` hides bugs. If a user types something like "hello" instead of a number, `==` will simply return `false` and your program will go on its merry way. But in reality, that's a bug! Your program isn't making sure that the user gives a valid input (a number), so what you want is an error that induces you to fix that. When you try to compute `+guess`, because you can't apply unary plus to a non-numeric string, you will get a noisy error in the console, which is what you want: errors should never pass silently.

If we play this game, we will see `false` in the alert when we guess wrong, and `true` when we guess right. That's useful, but wouldn't it be nice if we could pop up a message that said "You win!" if you guess it, and "You lose." otherwise? It turns out that we can. In fact, there are multiple ways to do it, but we will start with a **conditional expression**.

A conditional expression looks like this:

:javascript:

```
condition ? true_expression : false_expression;
```

If the `condition` evaluates to `true`, then the value of this entire expression (the whole line) is `true_expression`. If it evaluates to `false`, then the value of the entire expression is `false_expression`. That's somewhat abstract, so after you have pondered it, see if you can figure out how it is used in our improved guessing game:

:html:

```
<script>
var answer = 6;
var guess = prompt("Guess the number I'm thinking of:");
var msg = (answer === +guess) ? "You win!" : "You lose.";
alert(msg);
</script>
```

What happens when you run this program now? Now, if you get the answer correct, the message sent to `alert` will be "You win!". If you get it wrong, the alert will be "You lose". Depending on how complex the expressions are in the conditional `?:` operator, you may see parentheses around parts of it, as well, as was the case here. It's generally a good practice to put the condition in parentheses just to make things extra clear.

To explain this more simply in context, the truth of `answer === +guess` determines which value `msg` gets, selecting from the two choices on the right.

If and Else

Conditional expressions are helpful and they are used all over the place, allowing us to respond to the truth of things that we don't know before the program runs (like whether someone will type what we want them to), but if we have a lot of things we want to do when something is true or false, they can be pretty hard to read. We know that we can do a lot of things in a function, which we could call inside of the `?:` operator, like this:

:javascript:

```
function whenTrue() {
  console.log("whenTrue");
  return 10;
}
```

```
function whenFalse() {
    console.log("whenFalse");
    return -10;
}

var num = +prompt("Enter a number");
console.log("Result",
    (num >= 0) ? whenTrue() : whenFalse());
```

Since we just call functions in the conditional expression, we can actually put a lot of code in there that responds to the two possible situations. This works fine, but there is another, often clearer, way. That way is the `if` statement.

It has been a while since we did any drawing, so let's start this next section with the standard canvas skeleton code to give us a chance to draw some more:

:html:

```
<canvas id="drawing" width="400" height="300"></canvas>
<script>
var canvas = document.getElementById('drawing');
var ctx = canvas.getContext('2d');
ctx.fillStyle = 'black'; // Default to black shapes.

// And now for something fun. What should we do?
</script>
```

Instead of making a guessing game, we will ask a “yes/no” question using the `confirm` function, and we will color a rectangle if we get a “yes”. Otherwise we will use the boring default.

The `confirm` function pops up a little window like the `alert` function does, but it has two buttons: “OK” and “Cancel”. If you press OK, the function returns `true`, otherwise it returns `false`. We can use that to demonstrate how the `if` block works.

After getting the context above, try adding the following code to your program (after getting the context) and running it in the browser:

:javascript:

```
ctx.fillStyle = 'black';

var changeColor = confirm("Change color?");
if (changeColor) {
    ctx.fillStyle = 'blue';
}
ctx.fillRect(10, 10, 100, 100);
```

After we set the `fillStyle` to solid black (the default, but we're making it explicit for clarity), we check what we get back from `confirm`. If we get `true`, we change the fill color

to blue before actually drawing the rectangle. Give it a try with one of “OK” and “Cancel”. Once the program has run, reload the page and try the other button.



Figure 5.2: OK on the left, Cancel on the right.

As you can see, if the condition in parentheses is true, the body is executed, otherwise it is skipped. That is the essence of the `if` statement. The body is delimited by curly brackets, just like in a function. Thus, we get the value for `changeColor` from the button pressed, then if it was “OK” we set the fill style. If not, we skip that step.

No matter what happens, we still call `fillRect` to draw the rectangle.

There is a kind of interesting detail here about the condition that needs to be brought up. We have said that the body executes if the condition is “true”, but what really happens is it executes if the condition is “truthy”². Yes, that’s the real word for it. The opposite is “falsy”³.

It is actually easier to define “falsy” than to define “truthy”. Something is “falsy” if it is one of the following values:

- `false`
- `0`
- `''`
- `NaN`
- `null`
- `undefined`

Everything else is truthy.

At least, *mostly* everything else is truthy. There are caveats and exceptions that you may never run into, like the long-deprecated and Microsoft-invented `document.all` object, which violates the standard on purpose for browser compatibility’s sake.

²<https://developer.mozilla.org/en-US/docs/Glossary/Truthy>

³<https://developer.mozilla.org/en-US/docs/Glossary/Falsy>

Thus, the body of an `if` statement executes if the condition is *truthy*, but does not execute if the condition is *falsy*. That is fairly awkward to say, so we will keep saying “true” and “false”, but you should know that we mean “truthy” and “falsy” when we refer to any kind of conditional statement or expression. This includes `if` statements, conditional ternary expressions `?:`, and loop conditions, which we will talk about later.

What if we want to do one thing when the condition is true, and another thing when it is false? We can actually do that right now without any new ideas, because we know how to negate the meaning of a boolean value: we use the “not” operator. It looks like this: `!`. Let’s see what the code would look like if we had two `if` statements used in this way:

:javascript:

```
var useBlue = confirm("Use blue?");
if (useBlue) {
    ctx.fillStyle = "blue";
}
if (!useBlue) {
    ctx.fillStyle = "orange";
}
ctx.fillRect(10, 10, 100, 100);
```



Figure 5.3: OK on the left, Cancel on the right.

One unfortunate thing about the “not” operator `!` is that it is so visually narrow compared to what it is inverting. Sometimes it’s easy to miss, so take a close look at that code and convince yourself that it will use blue when we select “OK”, and orange when we select “Cancel”. The condition `useBlue` is true precisely when `!useBlue` is false, and vice versa. But, there is a way to do this without mentioning the condition `useBlue` more than once; it’s the second half of the `if` statement, called `else`:

:javascript:

```
var useBlue = confirm("Use blue?");
if (useBlue) {
    ctx.fillStyle = "blue";
}
```

```

} else {
  ctx.fillStyle = "orange";
}
ctx.fillRect(10, 10, 100, 100);

```

This does the same thing as our previous example, but it only mentions `useBlue` once. The body of the `if` is executed if the condition is truthy, and the body of the `else` block is executed if the condition is falsy. Either way, the `fillRect` statement is always executed, since it comes after the whole `if/else` statement.

Where you put the `else` keyword *visually* is not terribly important—as long as the only thing between the `if` block and the `else` keyword is space (including empty lines), it will work fine. The formatting style shown here is in fairly common use because it is both compact and readable. It makes very clear the fact that the `if` statement is not complete yet, as well, making it easier for others to follow your code. Other formats are also used.

Else If

What if you want to test for more than two conditions? For this, we have the `else if` pattern. Let's go back to our number-guessing game for a moment, but this time we want to handle *three* cases: a correct answer, an answer that is too high, and an answer that is too low. Since that is three different things, a single `if/else` block won't really cut it. One way to deal with this is multiple `if/else` blocks, with one inside another:

:javascript:

```

var answer = 6;
var guess = +prompt("Guess a number");
if (guess === answer) {
  alert("You win!");
} else {
  if (guess < answer) {
    alert("Too low");
  } else {
    alert("Too high");
  }
}

```

The `else if` pattern allows us to collapse this logic into a single list of bodies, like this:

:javascript:

```

var answer = 6;
var guess = +prompt("Guess a number");
if (guess === answer) {
  alert("You win!");
} else if (guess < answer) {
  alert("Too low");
}

```

```
} else {  
    alert("Too high");  
}
```

You can add as many `else if` sections as you like. It's a nice, clean way to test for multiple things in order. The first body with a true condition runs, then the rest are ignored and the whole pattern is finished.

Let's see what it's like to add yet another `else if` block, by also testing for guesses that are "way too high":

:javascript:

```
var answer = 6;  
var guess = +prompt("Guess a number");  
if (guess === answer) {  
    alert("You win!");  
} else if (guess > answer + 10) {  
    alert("Way too high");  
} else if (guess > answer) {  
    alert("Too high");  
} else {  
    alert("Too low");  
}
```

There we have it: the `else if` pattern. If you're testing for multiple possible outcomes, then you can check each one in order by using the `if else if else if ... else` style shown above. It's very common and pretty readable. As long as you keep in mind the fact that the first thing to be true will run, and the rest will be skipped, you should be in good shape. This is why we put "way too high" before "too high". See if you can figure out what would happen if the order were reversed.

As is commonly the case, we have glossed over a fairly important feature of the `if/else` statement, and that is that *curly braces are not strictly required*. What is required is that there be a *single statement* after each of the `if` and `else` keywords. Using curly braces allows us to collect *multiple statements* together to treat them as a single compound statement.

In practice, ***please never leave off the braces***. Always use the curly braces on the body. To illustrate why, consider the following:

:javascript:

```
if (useBlue)  
    ctx.fillStyle = "blue";  
else  
    console.log("not using blue");  
    ctx.fillStyle = "orange";
```

That *looks* okay on the surface, but it really isn't. Let's format it so that it is very clear which statements are attached to which bodies:

:javascript:

```
if (useBlue) ctx.fillStyle = "blue";  
else console.log("not using blue");  
ctx.fillStyle = "orange";
```

Aha! So, because we did not use curly braces on the `else` block, the `ctx.fillStyle = "orange"` command is run *every time no matter what useBlue says*. That's not what we signed up for, is it?

This situation, where new statements are added to `if` and `else` bodies, is very common, so follow this advice and you will be glad: *always use the braces*, even when they are not technically needed. Severe security errors have occurred because that advice was not followed, like Apple's SSL/TLS bug in 2014^a. That bug was introduced because a programmer added one more statement to an `if` body, except it did not have braces, so the statement ended up outside the body and was *always executed*. That was incorrect and not the programmer's intent, and it opened up a rather serious security hole.

It's true that there are debates surrounding the 2014 Apple SSL/TLS bug that point out a problem with the tools missing unreachable code, and the programmer using `goto`, and tests being inadequate, and they are all right, as well. But, if braces had been used consistently throughout the code, the error would have been less likely to creep in at all. Some contend that an extra curly brace does not make enough of a difference visually to make bugs like that easy to spot, but they are kind of missing the point: an extra curly brace on a line all its own makes it less likely for the wrong thing to be typed or pasted outside the block *in the first place*.

Always use braces.

It is useful to note, however, that the entire reason that `else if` works at all is because of this single-statement behavior that you should basically never use. In the `else if` pattern, the single statement after `else` is an entire `if` statement. We just write it in a nicer, more convenient-looking way. It's the only exception you should ever make for the "always use braces" rule, and it's safe to do so because this specific pattern is so rigid and predictable.

^a<http://mfukar.github.io/2014/02/22/Apple-SSL-bug.html>

Analysis Using Truth Tables

Sometimes when reading someone else's code, you will come across something that is really hard to understand. Perhaps it is a long nested chain of `if` and `else` blocks. If you do come across something like that, you can nearly always untangle things by building a sort of truth table from the statements. Let's look carefully at our example from before, but written with only `if` and `else`, no `else if` blocks:

:javascript:

```

var answer = 6;
var guess = +prompt("Guess a number");
if (guess === answer) {
    alert("You win!");
} else {
    if (guess > answer + 10) {
        alert("Way too high");
    } else {
        if (guess > answer) {
            alert("Too high");
        } else {
            alert("Too low");
        }
    }
}
}

```

This works exactly the same way as our chain of `else if` above, but it's a bit harder to be sure. Let's try untangling it using a truth table. Truth tables are pretty helpful tools for cases like these, though you might never actually need to do this. Some programmers never do. It's useful enough to go over it here, though, so you can choose to use it if you want to later.

To build a truth table for the above, we must first figure out what all of the variables in the table will be. These are the things that appear in conditions, and they are

- `guess === answer`
- `guess > answer + 10`
- `guess > answer`

Of course, for each of these there is a corresponding `else`, so each condition can be true or false. Let's build our table (we will use 0 for false and 1 for true, as is common). Remember, to get all of the row inputs, we just count in binary, giving us the values of the first 3 columns, over 8 rows in the table below (we let the word "guess" be implied in our table headings).

We have a truth table, now, so we're ready to start filling it in. Note that some of the rows are impossible: you can't have `guess > answer + 10` while not *also* having `guess > answer`. Similarly, you can't have equality and inequality simultaneously. So we mark these impossible cases right away. We can discard those rows.

=== answer	> answer + 10	> answer	Output
0	0	0	
0	0	1	
0	1	0	<i>impossible</i>
0	1	1	
1	0	0	
1	0	1	<i>impossible</i>

===	answer	>	answer + 10	>	answer	Output
	1		1		0	<i>impossible</i>
	1		1		1	<i>impossible</i>

Having discarded the impossible rows, our task is to ensure that all of the possible outputs are triggered at the right times, and the table will help us with that.

Proceed by following the code for the first row, where we find that the first check is false, so we skip the “You win!” clause and enter the `else` block. The next check is also false, so we skip the “Way too high” clause and enter that `else` block. The third check is *also* false, so we skip the “Too high” clause and enter the final `else` block, where we output “Too low”. Let’s add that to our table, and we can do the second row while we’re at it (see if you can fill it in yourself):

===	answer	>	answer + 10	>	answer	Output
	0		0		0	“Too low”
	0		0		1	“Too high”
	0		1		1	
	1		0		0	

Now we look at the third row. This row says that things are not equal, and the guess is both higher and “much higher” than the answer. Since `guess === answer` is false, we skip the first block. The next test is for `guess > answer + 10`, which is true, and we output “Way too high” immediately. That means we *never check for* `guess > answer`. When we come upon a case where we never check one of the conditions, we mark it with a ? to indicate “don’t care” (or “never tested”):

===	answer	>	answer + 10	>	answer	Output
	0		0		0	“Too low”
	0		0		1	“Too high”
	0		1		?	“Way too high”
	1		0		0	

Now that we have a “don’t care” value in there, it’s useful to ask ourselves: is this the right thing? Let’s reason through it by asking the question and coming up with an answer to it:

Question: If `guess > answer + 10`, do we care about `guess > answer`?

Answer: No, we don’t care, because `guess > answer + 10` *implies* `guess > answer`. It’s not possible for the guess to be *much* bigger than the answer without it also being *bigger* than the answer.

Oh, good. Logically, it makes perfect sense that we don't care about `guess > answer` if we get a true value for `guess > answer + 10`. That is evidence that the order of our `else if` blocks is well-designed and matches intuition. It is possible to do this wrong, for example, if we checked `guess > answer` *first*, then we would never notice when `guess > answer + 10`, and the truth table would show us that error.

See how putting the truth/output table together like this can help us to catch errors and understand code? It is a very useful technique.

Let's continue with what is now the fourth row, the row with 1 0 0 as its set of conditions.

- `guess === answer` is true, so output "You win!" and skip the rest of the code.

Here we go again. We have ignored *two* conditions, now. That means we don't care about them, so we end up with this:

<code>=== answer</code>	<code>> answer + 10</code>	<code>> answer</code>	Output
0	0	0	"Too low"
0	0	1	"Too high"
0	1	?	"Way too high"
1	?	?	"You win!"

Once again, we ask ourselves: is this right? The answer is again "yes". The reason is simply that the guess cannot *equal* the answer at the same time that it is *bigger than* the answer. Therefore, if we get a true value for equality, we will *never* get true values for the inequalities, so we can just not test them. We don't care about them.

Well, good! Our code seems to be structured reasonably, and we just learned how to analyze complex conditional expressions using the idea of a truth table, which we morphed into an output table. That's a very powerful tool.

There is one more thing that we can get out of this output table, and that is the fact that we can take our nested conditionals and convert them into a chain of "else if" statements. How do we see that in the table? Well, let's turn it upside-down and see if we notice any interesting patterns within it. An important hint: look for things that jump out at you diagonally:

<code>=== answer</code>	<code>> answer + 10</code>	<code>> answer</code>	Output
1	?	?	"You win!"
0	1	?	"Way too high"
0	0	1	"Too high"
0	0	0	"Too low"

The upper right of the variables table is all ?s, there are 1s along the diagonal, and there are

0s in the lower left.

If you can organize your table like this, you can just use an “else if” chain to represent your logic. Why is that? Look at what is now the first row: if the first variable is true, we don’t care about the rest. Now the second row: if the first variable is false and the second variable is true, we don’t care about the rest. Finally, if the first two are false and the third is true, we output something, and then we handle the “everything is false” case (which would be the final `else` block).

If you build a table like this, and rearrange the rows and columns to get this pattern, you have an “else if” chain in disguise. That can be very useful indeed. Here is what the code looks like when inspired by the table transformation:

:javascript:

```
var answer = 6;
var guess = +prompt("Guess a number");
if (guess === answer) {
    alert("You win!");
} else if (guess > answer + 10) {
    alert("Way too high");
} else if (guess > answer) {
    alert("Too high");
} else {
    alert("Too low");
}
```

That looks just like we want it to. Now, it’s very important to note that using an output table like this is *just one tool in your toolbox*. Use it when it makes sense or when things are confusing, or when logic gets to be pretty subtle and you want to be sure of things.

Most of the time, however, programmers don’t think about conditions in this way, because they’re building the logic from scratch and already have an intuitive idea of what it should do, but occasionally the logic gets a little bit tortured. When that happens, knowing how to write it out and carefully trace through all outcomes in a table like this can be extremely valuable. You know about boolean logic, now, so you can switch back and forth between boolean algebra and JavaScript conditionals to really be sure you understand whatever code you’re looking at. That’s the kind of skill that can really set the good programmers apart from the rest.

Exercises

Exercise 5-1: Comparators and Boolean Values

Solution on page 389

Every expression in mathematics that involves a *comparator* has a boolean type. A comparator is something like $=$, $<$, $>$, \leq , \geq , etc.

1. What is the value of $7 > 10$?
2. What is the value of $3 = 3$?
3. What about the value of $3 = 3 \vee 7 > 10$?
4. What is the value of $3 = 3 \wedge 7 > 10$?
5. If $x = 2$, find two values of y that make this false, and two values of y that make it true: $x < 2y \wedge y < x + 4$.

Exercise 5-2: Boolean Connectives

Solution on page 389

- What is the symbol for OR?
- What is the symbol for AND?
- What is the symbol for NOT?
- What is their precedence order (highest to lowest)?

Exercise 5-3: Truth Tables

Solution on page 390

Compute a truth table for the expression $\neg A \vee B$. It will have 4 rows. You may use 0/1 or T/F:

A	B	$\neg A \vee B$
0	0	
0	1	
1	0	
1	1	

Exercise 5-4: More Truth Tables

Solution on page 391

Fill out the truth table for the expression $A \wedge B \vee \neg C$. Remember the order of operations!

A	B	C	$A \wedge B \vee \neg C$
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Exercise 5-5: Most Truth Tables

Solution on page 392

Create the truth table for $A \wedge B \vee C \wedge \neg D$. Hint: it will have 16 rows of values. You can use any representation of true and false you like, including ones and zeros.

Exercise 5-6: De Morgan

Solution on page 396

Use De Morgan's Law to transform the following expressions (move the outer negation into the expression). Hint: when there is no outer negation, you can first pretend that there are *two* outer negations instead, like this: $A \wedge B = \neg\neg(A \wedge B)$ (since one undoes the other), and you can move one of them into the expression.

1. $\neg(A \wedge B)$
2. $\neg(A \vee B)$
3. $\neg(A \wedge B \vee \neg C)$
4. $\neg(\neg A \vee B \wedge \neg C)$

Exercise 5-7: JavaScript and Equality

Solution on page 397

Give the true or false value for each of the following expressions:

1. `3 === "3"`
2. `1.6 !== 1.6`
3. `"hi" === 'hi'`
4. `"hi" !== "hello"`
5. `3 > 6`
6. `2 <= 2`
7. `7 >= 6`
8. `!(9 >= 9)`
9. `10 < 8 || 8 < 9`
10. `9 === 9 && 2 !== 3`

Exercise 5-8: Converting Strings to Numbers

Solution on page 398

When prompting for an answer using `prompt`, the value returned is a string representing the characters the user typed. For example, if you run this:

:javascript:

```
var val = prompt("Please enter a number:");
```

The user is shown a window with a text box into which they can type. There is nothing in that box that *forces* them to type a number, other than your message, which is really just advice.

Suppose, however, that the user does indeed type a string representing a number, like `11`. What should you do to `val` to ensure that you can do numeric things (like addition, subtraction, multiplication, etc.) to it?

Exercise 5-9: Conditional Expressions

Solution on page 399

It's possible to create an expression in JavaScript that is one thing for one situation, and something else for another. This is done using the ternary conditional operator: `?:`. What are the values of the following expressions?

Note that we threw a little surprise in here, but it's not too hard to make use of it: the number `0` is falsy, and *any other number* is truthy. Similarly, the empty string `' '` is falsy, but every other string is truthy. (Note that the concepts of “truthy” and “falsy” come up in the discussion of `if/else`).

1. `true ? "hello" : "good-bye"`
2. `5 < 6 ? "smaller" : "not smaller"`
3. `'hi' === 'hello' ? "yes" : "no"`
4. `5 ? "truthy" : "falsy"`
5. `0 ? "truthy" : "falsy"`
6. `' ' ? 1 : 0`

Exercise 5-10: If and Else

Solution on page 399

Write a program that does the following using `if` and `else` (don't forget your braces!):

- Ask the user for a number,
- If less than zero, output “negative”,
- Otherwise, output “non-negative”.

Exercise 5-11: More If and Else

Solution on page 400

For this program, use two techniques: the first will use nested `if/else` blocks, and the second will use an `else if` chain.

Write a program that outputs “negative” for a number less than zero, “zero” for a number equal to zero, and “positive” for a number greater than zero. Get the number using `prompt` and output either to the console or via `alert`.

Exercise 5-12: Output Tables

Solution on page 401

Make an output table for the “negative/zero/positive” program in the previous problem.

Exercise 5-13: More Output Tables

Solution on page 402

Consider the guessing game problem from this chapter, where different outcomes are displayed for guesses that are too small, too large, *much* too large, and correct. Now we are going to change it slightly. For the following altered code, build the output table and see if you can use it to spot the problem in our new implementation:

:javascript:

```
var answer = 6;
var guess = +prompt("Guess a number");
if (guess === answer) {
  alert("You win!");
} else {
  if (guess > answer) {
    alert("Too high");
  } else {
    if (guess > answer + 10) {
      alert("Way too high");
    } else {
      alert("Too low");
    }
  }
}
```

Midterm 1

We have covered a lot of ground! You have been exposed to some of the most fundamental aspects of programming in JavaScript, and now it is time to make sure it is all clear and comfortable before moving on.

The most important concepts covered thus far are

- Syntax
 - Comments
 - Strings
 - Numbers
 - Variables
- Evaluation steps, including functions (algebraic)
 - Expand, substitute, evaluate, repeat until finished
 - Order of evaluation (inside-out, left-to-right, operator precedence)
- Basic algebraic recursion
- Writing our own functions in JavaScript
 - Variable scopes and `var`.
 - Anonymous functions.
 - Timer events.

- Objects as key-value storage.
 - Insertion, deletion, retrieval, and rules for dot notation.
- Boolean concepts
- If/Then/Else and ternary conditionals.

Before taking the test below, take some time to study these concepts in the preceding chapters.

Exercise 5-14: Basic types and outputs

Solution on page 403

In the following code, fill in the missing values (look for '?'). Also say what *type* the result has. What kind of a thing is it?

:console:

```
> 10 + 5
< ?
> 3 * 12
< ?
> "hello" + " " + "world"
< ?
```

Exercise 5-15: Variables and evaluation

Solution on page 403

Fill in the output of the console below where they are missing ('?'). Remember to be careful about assignment - a variable only changes value when assigned.

:console:

```
> x = 10
< ?
> x + 15
< ?
> x / 5
< ?
> y = 'hello'
< ?
> y + ' there'
< ?
```

Exercise 5-16: Show the steps for evaluating a numeric expression

Solution on page 404

Using the expansion technique demonstrated earlier, show the steps of evaluation for the following expression:

$$3(2 + 4) - (3/(12 + 2))$$

Exercise 5-17: Show the steps for evaluating a different numeric expression

Solution on page 405

Do the same as before, recalling that the \cdot symbol stands for multiplication and the $/$ symbol stands for division:

$$3 + 5 \cdot 2 + 6/3 - 4$$

Exercise 5-18: Show the steps for evaluating a function with variables

Solution on page 405

The function $f(x, y)$ is given below. Using the expansion technique demonstrated earlier, show all of the steps of how you would evaluate $f(25, 3)$:

$$f(x, y) = \frac{x}{5} + 3 + y^3$$

Exercise 5-19: Recursion in algebraic evaluation

Solution on page 406

Evaluate $f(4)$:

$$f(x) = \begin{cases} 0 & \text{for } x \leq 0 \\ x - f(x - 1) & \text{otherwise.} \end{cases}$$

Exercise 5-20: Calling JavaScript Functions

Solution on page 407

Write a one-line program that shows a pop-up window with the text “Hello!” in it. Use the `alert` function.

There is no need to show the surrounding HTML—just show the relevant JavaScript code.

Exercise 5-21: Prompting For Values

Solution on page 408

Using the `prompt` function, ask the user for their name, and display “Hello <name>!”, where <name> is replaced by whatever the user types. For example, if I were to type “Chris” in your prompt box, you would display “Hello, Chris!”.

The correct answer need not contain surrounding HTML tags, though it can. The JavaScript that displays the prompt and the final message is the important part of the answer.

Hint: you will call two functions in a correct program, and only one of them will be `prompt`.

Exercise 5-22: Writing Functions

Solution on page 409

You will write two function definitions for this exercise:

- A niladic function (a function that accepts no parameters) called `getName` that prompts the user for a name and returns the value, and
- A function called `showGreeting` that accepts a name and displays “Hello <name>!”, as in the previous exercise.

A correct set of function definitions will allow you to write the previous answer as

:javascript:

```
var name = getName();  
showGreeting(name);
```

or even

:javascript:

```
showGreeting(getName());
```

Exercise 5-23: Events

Solution on page 409

Write a short program that causes “Hello!” to be displayed in a pop-up window after 5 seconds have passed. Use `setTimeout` to accomplish this.

Chapter 6

Iteration Through Recursion

You now have the ability to change the course of your programs based on things that happen outside of them, like user input. You also have the ability to define and call functions. Since JavaScript functions are **recursive**, you can perform just about any computation you want to with only what you have learned so far. Recursion is a really important idea in computer science, and getting at least a little exposure to it can really enhance your brain, and this chapter provides some of that exposure.

Before we begin, however, an important note is in order. While recursion is a fascinating and powerful idea, it has a reputation of being intimidating to some, especially at first. This book endeavors to make it as simple and easy as possible to understand, but if you find yourself struggling, that's okay. You can focus your attention on the parts that teach you how to draw lines on a canvas, not worry too much about the rest, and then move on to the next chapter without any trouble. Meanwhile, give it a try and see how it goes!

We begin our journey with a story.

Gauss and the Sum of Integers

Carl Friedrich Gauss was in school in the 1700s, and reportedly his class was given some busy work. The teacher asked all of the students to sum the numbers 1 to 100, which was expected to take some time.

It is possible that this story is apocryphal, but it is fun and widely reported.

The child Gauss, however, noticed a pattern. If you line up the numbers from 1 to 100 right next to the numbers from 100 to 1, you can add up pairs like this:

$$\begin{array}{rcl}
 1 + 100 & = & 101 \\
 2 + 99 & = & 101 \\
 3 + 98 & = & 101 \\
 4 + 97 & = & 101 \\
 \vdots & & \\
 99 + 2 & = & 101 \\
 100 + 1 & = & 101
 \end{array}$$

Do you see how we have the sequence 1 to 100 written down twice? On the left of $+$ we have it in ascending order, and on the right of $+$ we have it in descending order. Remarkably, pick any row, and it sums to 101. We have 100 of those rows.

If we add up 100 rows of the number 101, we get $100 \cdot 101 = 10100$, which is really easy to calculate. But that number is the sum of two sequences from 1 to 100 (one forward, one backward). We want the sum of just one of them, so we just divide by 2 to get our answer! $10100/2 = 5050$, and now we're done. We have the sum of all integers from 1 to 100.

Multiplication by 100 is obviously a lot simpler and faster than adding up 100 things, so Gauss was able to obtain the sum in a few moments instead of the time it would take to add up each number individually. Cue astounded teacher and (we can hope) early recess for Gauss, who might have spent it working out more math problems.

By the way, *this is real math*: this creative process of looking for patterns and finding interesting things in them, things that can make your life easier, or lead to interesting results. Math is not about “learn to multiply 3-digit numbers” or “solve for x ”. Those are just tools and practice.

Math is about *exploring reality using self-consistent models*, and learning where that takes us. It's about figuring out cool things and having lots of fun “aha!” moments like this one.

If you have, in the past, felt like math wasn't your thing, it is quite possible that you weren't really doing math, just computation and arithmetic. Those can indeed be boring, but *math* can be **delightful**.

This pattern actually holds for any positive integer n (not just 100). We obtain a more general formula by replacing the number 100 in the table above with n :

$$\begin{array}{rcl}
 1 + n & & = n + 1 \\
 2 + (n - 1) & & = n + 1 \\
 3 + (n - 2) & & = n + 1 \\
 4 + (n - 3) & & = n + 1 \\
 \vdots & & \\
 (n - 3) + 4 & & = n + 1 \\
 (n - 2) + 3 & & = n + 1 \\
 (n - 1) + 2 & & = n + 1 \\
 n + 1 & & = n + 1
 \end{array}$$

All we did was replace 100 from the previous table with n . With that transformation, there are n rows in the table and each row sums to $n + 1$, so we multiply n by $n + 1$ and then divide by 2 (because, as before, the sequence appears twice). That gives the following more general formula:

$$1 + 2 + 3 + \dots + n = \frac{n(n + 1)}{2}$$

That's pretty fun! You can sum up all of the numbers from 1 to n by plugging in n and doing a multiplication and division. The formula works for all positive integers ($n \geq 1$). We can check that it still works for 100 by plugging that in for n : $100(100 + 1)/2 = 5050$.

By the way, what we just did is called **abstraction**. We took a concrete problem—the sum of numbers from 1 to 100—and turned it into a slightly more abstract problem by replacing one of the constraints with a variable, giving us the problem of summing numbers from 1 to n . We can reason about what that sum looks like without knowing anything about n except that it's a positive integer.

Programming is often an exercise in finding and creating the right abstractions. We will talk more about that throughout the course.

Recursive Dumb Sums

Gauss was a brilliant mathematician, and contributed much to science and mathematics that we still benefit from today. One wonders what he could have done with a computer!

Imagine the cat pictures!

Even though we may not feel as smart as Gauss, we can calculate the sum from 1 to 100 pretty quickly using a computer, even without his remarkable insight and formula. Let's see what it would take to get JavaScript to give us an answer to the sum problem, doing it the hard way on purpose.

One thing we could do is type all of the numbers into one big expression, like this:

:javascript:

```
function sum() {
  return 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 +
    11 + 12 + 13 + 14 + 15 + 16 + 17 + 18 + 19 + 20 +
    21 + 22 + 23 + 24 + 25 + 26 + 27 + 28 + 29 + 30 +
    31 + 32 + 33 + 34 + 35 + 36 + 37 + 38 + 39 + 40 +
    41 + 42 + 43 + 44 + 45 + 46 + 47 + 48 + 49 + 50 +
    51 + 52 + 53 + 54 + 55 + 56 + 57 + 58 + 59 + 60 +
    61 + 62 + 63 + 64 + 65 + 66 + 67 + 68 + 69 + 70 +
    71 + 72 + 73 + 74 + 75 + 76 + 77 + 78 + 79 + 80 +
    81 + 82 + 83 + 84 + 85 + 86 + 87 + 88 + 89 + 90 +
    91 + 92 + 93 + 94 + 95 + 96 + 97 + 98 + 99 + 100;
}
```

That's obviously not very fun, nor is it generally useful: you can only sum the numbers 1 to 100 this way. Summing the numbers 1 to an arbitrary n is not something we can do with this. How could we write this to sum 1 to n ?

Of course, we could just define a function that takes n as a parameter, and define it using Gauss's formula, like this:

:javascript:

```
function sum(n) {
  return (n * (n + 1)) / 2;
}
```

That's actually pretty cool, but we did say we were going to do this the hard way on purpose. We therefore begin by creating a **recurrence relation**, then we will figure out how to write code that computes it. Why a recurrence relation? Because recursion is amazing, powerful, and useful, and because we already know a bit about how it works. Let's use it!

If we're going to make a recursive function do this work for us, it makes sense to look at the algebra of this "sum everything from 1 to n " problem. We begin as usual with recurrence relationships, by using our imagination. We start by pretending that we have a function $f(n)$ that finds the sum we want. How would we define it? Here is one way:

$$f(n) = 1 + 2 + 3 + 4 + 5 + \dots + (n - 2) + (n - 1) + n$$

That is nice, and correct, but if we tried to write a function for it using what we have learned so far, it would be hard: we don't know how many things we are adding! But that's okay,

because right now we are *pretending* that we already have a good definition for $f(n)$, and that means we can pretend to have a good definition for $f(n - 1)$, as well. Let's see how we might make use of that pretend function to make a real one.

If we write $f(n)$ next to $f(n - 1)$, we start to see something interesting:

$$\begin{aligned} f(n) &= 1 + 2 + 3 + 4 + 5 + \cdots + (n - 2) + (n - 1) + n \\ f(n - 1) &= 1 + 2 + 3 + 4 + 5 + \cdots + (n - 2) + (n - 1) \end{aligned}$$

It looks like $f(n - 1)$ is just the first part of $f(n)$. In fact, it contains the entire sum except for the final number n :

$$f(n) = \underbrace{1 + 2 + 3 + 4 + 5 + \cdots + (n - 2) + (n - 1)}_{f(n-1)} + n$$

and therefore,

$$f(n) = f(n - 1) + n$$

That's nice and clean, and we're nearly there!

This process—defining a problem as a smaller version of itself plus some trivial additional computation—is a key idea. Here, we have discovered that you can compute the sum of numbers 1 to n by *first* computing the sum from 1 to $n - 1$, then adding n . We have thus defined $f(n)$ in terms of $f(n - 1)$ and simple addition, which is an incredibly useful way to think about it if you are writing a recursive function.

Now, what's missing? A way to stop expanding this when we plug in a number. Consider this:

$$\begin{aligned} f(3) &= \underbrace{f(2)} + 3 \\ &= \underbrace{f(1)} + 2 + 3 \\ &= \underbrace{f(0)} + 1 + 2 + 3 \end{aligned}$$

Oops! We don't know how to compute $f(0)$ because we only know what $f(n)$ means for $n > 0$. It is inevitable that we get there, however, because each time we expand the function definition, we subtract 1 from a smaller value of n .

To solve this, we need an **initial condition** (or a “stopping condition”), a value of n that is valid and that makes $f(n)$ trivial to compute. Since we are limiting ourselves to positive integers, we will choose the first of them as our initial condition: $f(1) = 1$. That’s easy to compute, trivial to see, and now we can create a complete recurrence relation:

$$\begin{aligned} f(1) &= 1 \\ f(n) &= f(n-1) + n \end{aligned}$$

A more common way of writing this is as a piecewise function, where the use of each part is a bit more explicit, and it’s clearly not defined for $n \leq 0$:

$$f(n) = \begin{cases} 1 & \text{when } n = 1 \\ f(n-1) + n & \text{when } n > 1 \end{cases}$$

We don’t always use the word “when” in piecewise function definitions like the one above. Generally, if you see a large left curly brace with multiple things in it for a function definition, the stuff on the left is the value and the stuff on the right is the condition when that value applies.

To refresh your memory from the previous time we discussed recursion, here is how that expansion process works:

$$\begin{aligned} f(100) &= \underbrace{f(99)} + 100 \\ &= \overbrace{f(98) + 99} + 100 \\ &= \overbrace{\underbrace{f(97)}_{f(96)+97} + 98} + 99 + 100 \\ &\vdots \\ &= \overbrace{\overbrace{f(2)}^{f(3)} + 3} + \cdots + 99 + 100 \\ &= \overbrace{f(1)} + 2 + 3 + \cdots + 99 + 100 \\ &= \hat{1} + 2 + 3 + \cdots + 99 + 100 \end{aligned}$$

Hopefully it is clear how the recurrence relation gives us back exactly the sequence we are interested in, just by performing repeated expansions. Hopefully it is also clear where we

used the definition $f(1) = 1$ at the bottom of the expansion to stop the process and finally produce an answer.

With our recurrence relation in hand, we can translate it into a JavaScript function:

:javascript:

```
function sum(n) {  
  // Initial condition.  
  if (n === 1) {  
    return 1;  
  }  
  if (n > 1) {  
    return sum(n-1) + n;  
  }  
}
```

As a reminder, no matter where `return` is encountered inside a function body, the function exits immediately with that value, skipping anything that might be below. This is called a “short-circuiting `return`”, and we make use of it here.

With that reminder, take a moment to be sure you understand what is happening, and then we will move on with a different example. The key thing to take away from this is that you can create a function that does multiple things in a row, without knowing exactly how many there are before you run it!

You should type this in and try it out by calling it with some things you know the answer to, like `sum(3)` or `sum(100)`.

A note about undefined behavior is in order here. The function is undefined if $n < 1$, since it doesn't do anything in that case. That's what we want, and JavaScript will literally return the value `undefined` when that happens (no `return` statement is reached in that case). We'll let that slide for now because fixing it properly would muddy things substantially by introducing the concept of **exceptions**, but in general it's not great to have implicit, silent failure modes in your programs. Errors should be loud and obnoxious. That's how they get discovered and fixed.

Another more advanced note is in order, as well. This is a serviceable recursive function in many ways, but it has some subtle flaws. You would not really write it this way in production software. For example, it cannot be “tail-call optimized” because what it returns is not merely result of calling itself again with different arguments. That means that large values of n will actually cause the interpreter to run out of memory. This, however, is a perfectly fine way to *learn* about recursion, it just misses some important real-world considerations.

In other words, We have abstracted the idea of summing positive integers up to n , without first knowing what n is. If you want to try running this in a program, do feel free to give it a try. Also try making n fairly large, like a million, and see what happens.

Recursion is pretty neat, but this was obviously just practice since we can more easily and efficiently solve the summed integers problem using Gauss's formula. Let's practice it next in a more practical and visual way.

Drawing Lines

It's time to dust off our canvas element again. We will use it to draw a regular grid. To do that, we will need to draw lines. So far, we have only drawn filled-in rectangles, but the canvas has a lot more capabilities than that. We can find out from the canvas context documentation¹ that there are `moveTo`, `lineTo`, and `stroke` functions, for example. Those look pretty promising. There are also `lineWidth` and `strokeStyle` properties that we can use to control how lines are drawn. Let's take a look at how those work, and practice using recursion to do repetitive things in JavaScript.

A Detour Through Forbidden Paths

When drawing lines on a canvas, we think about it in terms of an invisible brush. We first *move* our invisible brush to the right place, then we instruct the context to (eventually) draw a line from there to a new spot, and finally we make it all real by calling `stroke`.

Create a new program, because we are going to be doing a lot of coding now. Remember, just open your text editor to a new file, write your code in it, and save it as `something.html` (where "something" can be whatever you want). In that file, indicate where your JavaScript code is by using `<script>` and `</script>` tags.

Here is a very basic start for our line-drawing program file. The first few lines of code are the familiar setup for a canvas and 2-dimensional context. After that, we draw a single line:

```
:html:
<canvas id="drawing" width="300" height="200"></canvas>
<script>
var canvas = document.getElementById('drawing');
var context = canvas.getContext('2d');

context.moveTo(10, 30);
context.lineTo(canvas.width - 10, 30);
context.stroke();
</script>
```

Find your file, drag it onto a new empty tab in your browser, and you should see a single horizontal black line.

How does this work? If you imagine an invisible coordinate system on your canvas, you can think of the operations above like this:

¹<https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D>

- `moveTo` moves the brush with it lifted away from the “paper” - it just positions it without dragging it there.
- `lineTo` drags the brush on the canvas from where it was to where you tell it to go, drawing a line on the way.

Thus, the code above goes to position $(10, 30)$, then draws a line from there to $(290, 30)$. Since the start and end points have the same y coordinates, the line is horizontal. It goes up to $x = 290$ because the canvas is 300 pixels wide, and we draw a line to that point minus 10. Doing it this way is not required, but it makes it easier to change the canvas size without having to alter all of our code to make the line fill it up.



Figure 6.1: One horizontal line.

One thing about the canvas isn't really obvious at first: nothing is actually drawn when calling `lineTo`. That function is just part of making a drawing *plan*. If you want the lines to actually appear, you have to tell the context that it's time to execute that plan. To do that, we call `context.stroke()` and then the line finally shows up. You can actually call `moveTo` and `lineTo` multiple times, then call `stroke` once and all of them will show up, like this:

:javascript:

```
context.moveTo(10, 30);
context.lineTo(canvas.width - 10, 30);
context.moveTo(10, 50);
context.lineTo(canvas.width - 10, 50);
context.stroke();
```

What's happening here? We are basically making a list of instructions, but not actually doing them until calling `stroke`. The instructions are to move somewhere, draw a line, move somewhere else, then draw another line. Finally, when we say `stroke`, all of those instructions are followed in the order we gave them, ending in an actual rendering of the lines.

If you want to change the properties of the lines, like making them wider or changing their color, you can do it as before, by setting style variables. With our earlier filled rectangles we



Figure 6.2: Two horizontal lines, one `stroke`.

used `fillStyle`, this time we will use properties specific to strokes. Because nothing is actually drawn until `stroke` is called, you can set the stroke style and width anytime before calling `stroke` and it will apply to everything drawn since the last output.

Note that the `fillRect` function is kind of special, in that it both computes a rectangle **and fills it** immediately. Most canvas drawing functions don't work like that.

:javascript:

```
// Drawing commands here or anywhere before stroke.  
  
context.lineWidth = 3;  
context.strokeStyle = 'red';  
context.stroke();
```



Figure 6.3: Red applies to everything before `stroke` is called.

There is another non-obvious thing to note about strokes: if you want to draw multiple strokes with different colors and styles, you will need to be more careful about how you signal when

things are changing. To paint in multiple colors, for example, you need to start your drawing with `beginPath` and end it with `stroke`, then change the settings and do it again.

Otherwise all of your strokes are assumed to be part of the same group, and they will all be drawn in the same style. It's a bit confusing if you don't know how operations are batched up and drawn after your code finishes running, so for now it's best to just remember: start lines with `beginPath` and end them with `stroke` if you want to change their styles.

Here is an example of what that looks like. Try replacing all of your drawing commands with these:

:javascript:

```
context.lineWidth = 3;
context.strokeStyle = 'red';

context.beginPath();
context.moveTo(10, 30);
context.lineTo(canvas.width - 10, 30);
context.stroke();

context.lineWidth = 5;
context.strokeStyle = 'blue';

context.beginPath();
context.moveTo(10, 50);
context.lineTo(canvas.width - 10, 50);
context.stroke();
```



Figure 6.4: Separate paths allow separate colors.

That's awfully verbose, which is one of the reasons there are about a million libraries to make the JavaScript canvas easier to use, but we're going to stick with built in facilities for this course. You can always learn to use another library on your own (and that is recommended—the canvas really is fairly clunky to use in its native form).

A Recursive Grid

Now that we know how to draw lines and have briefly reintroduced recursion, we're ready to write a program that draws a grid on our canvas.

Let's say that we want a grid where each square is about 10 pixels on a side. That might mean that we draw a vertical line at $x = 0$, another at $x = 10$, another at $x = 20$, and so on until $x = 300$ (or whatever our canvas width is). That makes 31 different lines to draw, something like this:

:javascript:

```
context.beginPath();
context.moveTo(0, 0);
context.lineTo(0, canvas.height);
context.moveTo(10, 0);
context.lineTo(10, canvas.height);
// A whole lot of other stuff here, every 10 pixels...
context.moveTo(300, 0);
context.lineTo(300, canvas.height);
context.stroke();
```

That's a lot of typing and a whole lot of mess just to get the vertical lines. Plus, we're only half done; we would need to do something similar to get all of the horizontal lines drawn!

Basically what we have done is taken a simple problem statement:

- Draw 31 vertical lines, spaced 10 pixels apart.

and turned it into a really annoying and difficult statement:

- Draw a vertical line at $x = 0$
- Draw a vertical line at $x = 10$
- Draw a vertical line at $x = 20$
- ...
- Draw a vertical line at $x = 300$

The first way we wrote the problem makes sense to us. It's very human-friendly. It's also more *abstract*. When we are solving this problem, the abstract version is easier to think about and to write down. But that's not all: it is also easier to rewrite for different square and canvas sizes! After all, if you want to instruct someone to draw 5-pixel squares, you can just say

- Draw 61 vertical lines, spaced 5 pixels apart.

Similarly, if the canvas size changes, you can just draw a different number of vertical lines.

We have kind of belabored this point, but it's important: abstraction allows us to not only make our problem statements (and code) more concise, but it also allows us to make it more general; it will work in more situations than those we have already thought of.

Let's take this idea and apply it now. How can we write a function that draws all of the vertical lines that we want it to? Does this feel like a place where we might apply recursion?

Yes! The trick that we used for summing numbers was to find a smaller version of the problem embedded in the problem we wanted to solve, then to write the bigger problem in terms of the smaller problem. We can do that here! We can pretend that our canvas is made up of two parts: a slightly narrower canvas that only allows 30 lines, and the rest of it that allows one more. Our base condition will be a drawing area that is too narrow to draw anything on at all.

Our problem statement now looks like this:

1. To **fill up a space** that is too small, do nothing.
2. To **fill up a space** with some non-zero `width`, draw a line at `width` then **fill up a space** that is smaller: `width - 10` pixels wide.

In other words, to fill a canvas with vertical lines, we draw a line at the right edge of it (where $x = \text{width}$), then pretend the canvas is 10 pixels narrower and fill that up. We stop when the width is zero. Do you see the recursive pattern here? How would you do this in code?

Here is one possibility. Note that we need to pass the context into the function to ensure that it is available for drawing:

:javascript:

```
function vertLines(ctx, width) {
  // 1) Don't draw if we have no room.
  if (width < 0) {
    return;
  }

  // 2a) Draw the last one.
  ctx.moveTo(width, 0);
  ctx.lineTo(width, ctx.canvas.height);

  // 2b) Draw all but the last one.
  vertLines(ctx, width - 10);
}
```

We are taking advantage of the fact that a context knows about its canvas. That allows us to use `ctx.canvas.height` to get the canvas's height from the context. We first get the canvas from the context, then we get the height from the canvas.

In general, when following dot notation, you can think of it as going deeper into objects contained within other objects: `height` is inside of `canvas`, and `canvas` is inside of `ctx`. But, it isn't really containment, is it? We get a context from a canvas, but then we can get the canvas from the context? The answer to this conundrum is that we have been using containment as a metaphor, but it breaks down, here. Really it is better to think of items within an object as being variables that can be assigned to anything a variable can

be assigned to. If I have an object `a` and an object `b`, they can have items that refer to each other, so that `a.thing === b` and `b.stuff === a`. That is totally valid; it just breaks our container metaphor.

That's it. If you remove the comments, it's a very short function. Let's see it in context of a larger program. If you have been following along thus far in your own code (which would be very helpful for learning), you just need to add a call to `vertLines(context, canvas.width)` and a final call to `context.stroke()` to make this all work:

:html:

```
<canvas id="drawing" width="300" height="300"></canvas>
<script>
function vertLines(ctx, width) {
  if (width < 0) {
    return;
  }
  ctx.moveTo(width, 0);
  ctx.lineTo(width, ctx.canvas.height);
  vertLines(ctx, width - 10);
}

var canvas = document.getElementById('drawing');
var context = canvas.getContext('2d');

// Now actually call this and stroke the lines:
vertLines(context, canvas.width);
context.stroke();
</script>
```

It's a really good idea to run this in the debugger by setting a breakpoint on the first line of `vertLines` and watching what the local variables do every time you hit "play". You will see the value of `width` count down by 10 from 300, and you will see the **call stack** grow for a while until it finishes drawing all the lines.

The call stack is always there, but invisible to you unless you look at the debugger (or your program crashes because it runs out of stack space). It's an implementation detail inside the JavaScript interpreter itself. Let's talk a little bit about it in terms of the code above.

The recursive line-drawing function works the same way any function call works: when a function is called, the current state of the program is saved in the **call stack** so that the program can continue where it left off when the function exits. Most of our program can be considered to be at the "top" of the call stack. The interpreter marches along, creating our `vertLines` function definition, then getting the value for `canvas`, then getting the `context` from that. Then it calls a function. To do that, it remembers what comes next (`context.stroke`), then creates some space for the function call's **local variables**, which are all **parameters** in this case: `ctx` and `width`. This space on the stack is often visualized as a block containing variable settings. We are glossing over a few details for the sake of

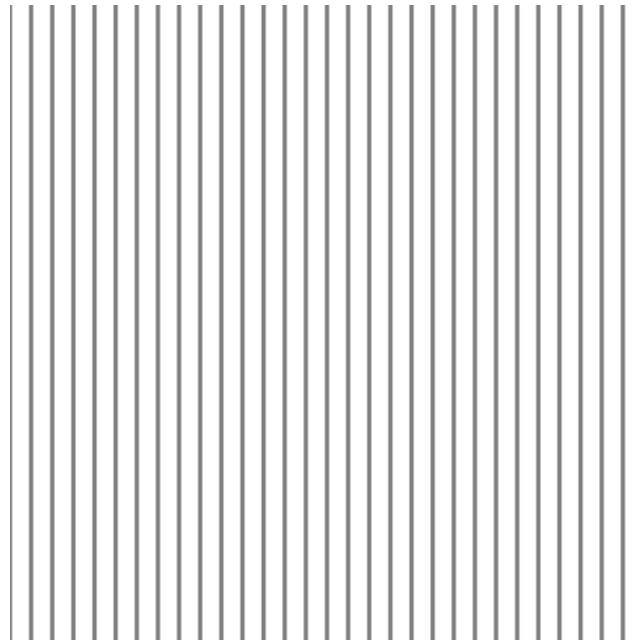


Figure 6.5: Vertical lines, generated recursively.

clarity. We will also leave out `ctx`, since it is always the same:

depth	name	value
1	width	300

Here we show the value at each “depth” of the stack. Since we just barely called `vertLines` once, the stack is only 1 entry deep, and the value of `width` at that depth is 300: the width of the canvas.

In order to do its work, the `vertLines` function draws a vertical line at `x = width`, then—before returning—calls `vertLines` again. That means our stack has to grow in order to remember where it left off so it can finish up after the (internal) call to `vertLines` returns. This gives us (after a second call) a stack that has stuff in it like this:

depth	name	value
1	width	300
2	width	290

But, in order to do the work for `width = 290`, the function has to call itself again with `width = 280`, and so on all the way down to some number less than 0, drawing vertical lines as it goes:

depth	name	value
1	width	300
2	width	290
3	width	280
...
31	width	0
32	width	-10

Before our main call to `vertLines(context, canvas.width)` finishes, then, the stack ends up having 32 entries on it. What happens at this last entry? Well, our function notices that `width < 0`, so it immediately returns. That removes its entry from the stack:

depth	name	value
1	width	300
2	width	290
3	width	280
...
31	width	0

Where the function that called it left off was right before its own return, so it is immediately removed from the stack as well. This happens to every one of the invocations of `vertLines` until the main call exits in our main program, leaving us at the line `context.stroke()`, which actually draws the lines we just set up.

That's how recursion works. Remembering where the interpreter left off takes space, and that space has to grow with every function call. Thus, recursive calls can end up using all available memory pretty fast even though you don't feel like you are using any more than usual. Once that happens, your program stops: no memory means no program.

The JavaScript standard briefly flirted with allowing **tail call optimization**, but it was never widely implemented and was ultimately abandoned in favor of something more explicit that has not yet been adopted, as of 2022, and it isn't clear that it ever will be. Thus, recursion will cause memory growth *in general*, and there is no currently accepted solution for that in JavaScript. The way we have written this example function, with only a function call as the last thing it does (the tail position), can already be optimized in that way if JavaScript ever becomes capable of doing it.

If tail call optimization is ever implemented in JavaScript, then the recursive line-drawing

function we've been playing with here would not actually grow the stack at all. Instead, each recursive call would *replace* the current function's space on the stack, because the interpreter can tell that it won't need to remember anything to clean up.

The important thing to note is that it is not magic: a function calling itself is just creating a new version of itself and remembering where it left off so it can continue later. It still feels magical sometimes, though.

To make the rest of the grid, you can easily create a similar function that draws horizontal lines and call it before `context.stroke()`.

There are many more cool things you can do with recursive repetition, like making rays using a bit of trigonometry. Here's one way to do something like that:

:javascript:

```
function rays(ctx, byDegrees, toDegrees) {
  if (toDegrees < 0) {
    return;
  }

  // Find the center, convert degrees to radians, etc.
  var angle = toDegrees * Math.PI / 180,
      size = ctx.canvas.width / 3,
      ox = ctx.canvas.width / 2,
      oy = ctx.canvas.height / 2;

  // Draw a single ray at the "toDegrees" angle.
  ctx.moveTo(ox, oy);
  ctx.lineTo(ox + size * Math.cos(angle),
             oy + size * Math.sin(angle));

  // Draw the rest of the rays.
  rays(ctx, byDegrees, toDegrees - byDegrees);
}

var canvas = document.getElementById('drawing');
var context = canvas.getContext('2d');

// Now actually call this and stroke the lines:
rays(context, 10, 360);
context.stroke();
```

This has essentially the same structure as our vertical lines problem, but it does something completely different! Here are some questions to ask yourself, even if you are not familiar with this use of the sine and cosine functions for plotting lines with a certain angle:

- What are the initial arguments to `rays`?
- Where does `rays` recur?
- What is changed in the arguments each time `rays` recurs?

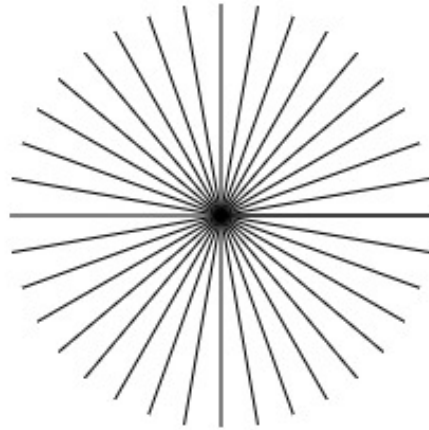


Figure 6.6: Recursive rays.

- What causes it to finish?

By answering those questions, you will probably begin to see how you could change this function to do more of what you want it to do, even without a perfect knowledge of the math behind it. That is pretty neat!

Onward

This chapter was meant to be a fairly gentle introduction to recursion in JavaScript, and it also teaches a thing or two about drawing lines and using information from the canvas like its `width` and `height`. The next chapter teaches a simpler, more common way of doing repetitive work.

Even if you don't feel supremely confident in everything you have learned here, each exposure to recursion can help you learn to think more like a successful, problem-solving programmer. Congratulations on getting through it!

Exercises

Exercise 6-1: Grid Recursion

Solution on page 411

Use recursion to implement the `hLines` function. It works like `vertLines`, but it draws horizontal lines instead by marching along the `y` coordinate instead of the `x` coordinate. Run our program with both horizontal and vertical lines and verify that it works properly.

Exercise 6-2: Partial Rays

Solution on page 412

Change the arguments passed initially to `rays` (the first time it is called by the interpreter) to end on something other than 360 degrees, and make the lines closer together (the original used 10 degrees, pick something smaller, like 5). Use this to make a sort of “sun on the horizon”, where only the top half of the circle is drawn.

Exercise 6-3: Circle-like Things

Solution on page 414

The `rays` function basically draws a bunch of straight lines emanating out from a central point, and it does it in order, starting at zero degrees (pointing to the right), then going around counterclockwise until it reaches the final angle. It does this by repeatedly calling `moveTo` to get to the central point, then calling `lineTo` to get away from that point.

Note that, without a `moveTo` function, the first `lineTo` behaves as though it were a `moveTo`. This is in the documentation for the context object, but in case you don't have access to that, you're welcome.

In this exercise, copy the `rays` function somewhere and rename it to be `polygon`. Then see what happens if you don't call `moveTo` at all. Before running your program, answer this question:

- What will happen if you don't call `moveTo` between line drawings? Hint: what was it doing before?

Once done, increase the `byDegrees` value in the first call to `polygon` to be 120. Before running the program again, answer these question:

- What shape will that draw?
- Why?

Exercise 6-4: Time to Fly

Solution on page 416

Write a program that draws `n` rectangles of decreasing height, starting at the left edge and finishing at the right edge. The first rectangle's height will be the height of the canvas, and the last rectangle's height will be `canvas.height / n`. Use a recursive function name `bars` to draw these.

Once complete, demonstrate that it works for any `n` by plugging in different values and showing how the shape changes.

Chapter 7

Arrays, Loops, Switches, and Randomness

Recursion is a powerful tool—and sometimes the most sensible one—but it’s not always the most obvious way to express common ideas. Furthermore, many popular languages—including the default version of JavaScript in the majority of modern browsers in 2022—do *not* employ the necessary optimizations to make it efficient. Sometimes it’s still the best tool for the job, but you always have to be mindful of how deep your stack is going to get when you use it.

So far, we have used recursion to do things one at a time, in order. That process is called **iteration**. We “iterate” over elements of an array, or over line positions on a canvas, or the numbers from 0 to $n-1$ to do something useful with each of them. The nice thing about recursion is that once you know functions, you don’t need to know much else; you can recur to iterate.

Still, that isn’t the clearest way to do it in JavaScript. In this chapter we are going to learn about **loops**, by far the most common way of doing iteration. As a bonus, we will also learn about the conditional `switch` statement, which will complete our exploration of JavaScript syntax concepts for a little while. After this, we will focus more on creating programs using the syntax we already know.

But first, let’s begin by introducing one more thing that holds data: the **array**.

Arrays

An **array** is basically an ordered list of things. You can put anything into an array, and any number of things (up to the limits of the memory). To create an empty array with nothing in it, you use square brackets, like this:

:javascript:

```
var a = [];
```

Indexing

To get an idea of the most common operations on an array, the following console exercise shows examples of how to create a non-empty array, and how to change and access values within it based on their **index**:

:console:

```
> a = ["milk", "butter", "bread"];
< ["milk", "butter", "bread"]
> a[0]
< "milk"
> a[2]
< "bread"
> a[1] = "margarine";
< "margarine"
> a
< ["milk", "margarine", "bread"]
> a.length
< 3
```

Note that the first element in an array has index 0. Arrays in JavaScript are 0-based, as all good arrays should be.

Pushing and Popping

You can also treat an array like a stack, where you push things onto the end. You can also pop them off:

:console:

```
> a.push("napkins")
< 4
> a
< ["milk", "margarine", "bread", "napkins"]
> a.pop()
< "napkins"
> a
< ["milk", "margarine", "bread"]
> a.push(42)
< 4
> a
< ["milk", "margarine", "bread", 42]
> a.pop()
< 42
> a.pop()
< "bread"
```

Pushing happens at the end (the largest index), and so does popping.

Contents and Length

As mentioned earlier, you can put anything you like into an array. It can hold strings, numbers, functions, objects, other arrays, etc. Anything that is a value can be stored there, just like values in an object. The main difference is that order matters in an array, and the elements are accessed by **index** instead of by **key**. The first index is always 0, and the last available index is one less than the length of the array.

Arrays are also objects with a `length` property (the number of elements in the array) and several useful methods like the `push` and `pop` methods demonstrated above. The full list of these methods is found with the Array documentation¹.

The length of an array is not necessarily the number of things it contains, because JavaScript arrays are *sparse*. This is not likely to come up in normal use, though, so we won't worry about it here.

As shown in the examples, to `push` something onto an array adds something to the end of it, returning the new length. To `pop` something removes the last element and returns it. These are probably the most commonly-used array methods.

Slicing

Arrays have a very useful method called `slice` that bears going over in some detail. We will use it later on. The `slice` method takes an array and gives you only part of it back. The first argument to `slice` tells it the first index to be part of the new array, and the second element is one beyond the last index of the new array.

Let's say we want to keep everything but the first element of an array. We could do something like this:

:javascript:

```
var ar = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
var newArray = ar.slice(1, ar.length);
```

Going from some element all the way to the end is pretty common, so there's a shorthand way to do it by just leaving off the last argument. This is therefore equivalent:

:javascript:

```
var newArray = ar.slice(1);
```

That gives us a new array with all of the same elements as the old one except for the very first element `ar[0]`.

¹https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

Note that `slice` does not change the original array, but rather it creates a new one with all of the appropriate values copied into it. If you want to change the original in place, `slice` is not going to do it. You can reassign the same *variable*, though, like this: `ar = ar.slice(1)`.

You can play around with `slice` some more in the console to see how it works. It's useful to start with an array that has values matching the indices so you can really get a handle on things:

:console:

```
> ar = [0, 1, 2, 3, 4, 5]
< [0, 1, 2, 3, 4, 5]
> ar.slice(1, 4)
< [1, 2, 3]
> ar.slice(2)
< [2, 3, 4, 5]
```

As you can see, the first argument tells us which element will be taken to be the first one in the new array, and the second argument tells us where we stop. As is hopefully abundantly clear now, the first index is **inclusive**, meaning it is included in the final result, and the second is **exclusive**, meaning, as you might expect, that it is excluded from the result—we take the element just before it to be the last one.

While Loops

With a data structure like the array, we might want to do something to every element in it. We have seen earlier how we could devise a recursive function to do something to a sequence, but there is a far simpler, memory-efficient, and syntactically clean approach: **loops**.

A loop, at its most fundamental, is a chunk of code that you run multiple times while some condition is true. The simplest kind of loop is the `while` loop. Here is its basic structure:

:javascript:

```
while (condition) {
  // If true, run this code
  // Without stopping, repeating,
  // 'Til false condition.
}
```

It looks an awful lot like an `if` statement, right? In fact, it's very similar! The difference is this: when an `if` statement executes, the body (in braces) is run exactly once when the condition is true. With a `while` loop, the body is run when the condition is true, then the condition is checked again. If it is still true, the body is run again, and again, and again until the condition becomes false.

Just like `if` statements, all loops can omit the curly braces if their bodies contain exactly one statement. Don't do that. That's off the edge of the map, and *there be dragons*.

We call this a **loop** because if you were to draw a diagram showing what happens, you would need an arrow “looping back” to the beginning after the body is executed.

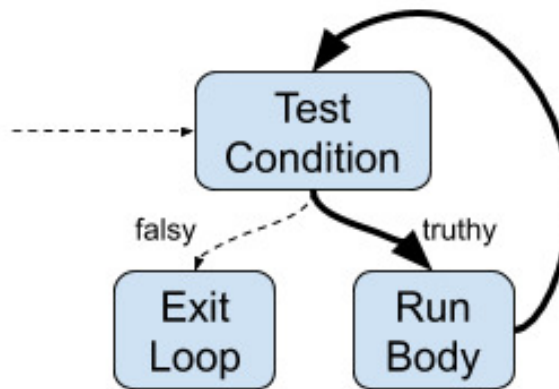


Figure 7.1: A while loop: the loopy part is shown with bolded lines.

If the condition never becomes false (or never encounters a `break` or `return` statement), then the loop never terminates. It is therefore important to remember to ensure that something in the body eventually changes the condition.

That seems like something we can do. Remember our grid-drawing task from the previous chapter? As a reminder, here is the code for one of the recursive `vertLines` function. To refresh your memory, try tracing through the execution path, starting at the call to `vertLines` at the bottom of this code:

:javascript:

```

<canvas id="drawing" width="300" height="300"></canvas>
<script>
function vertLines(ctx, width) {
  if (width < 0) {
    return;
  }
  ctx.moveTo(width, 0);
  ctx.lineTo(width, ctx.canvas.height);
  vertLines(ctx, width - 10);
}

var canvas = document.getElementById('drawing'),
    context = canvas.getContext('2d');

vertLines(context, canvas.width);

```

```
context.stroke();  
</script>
```

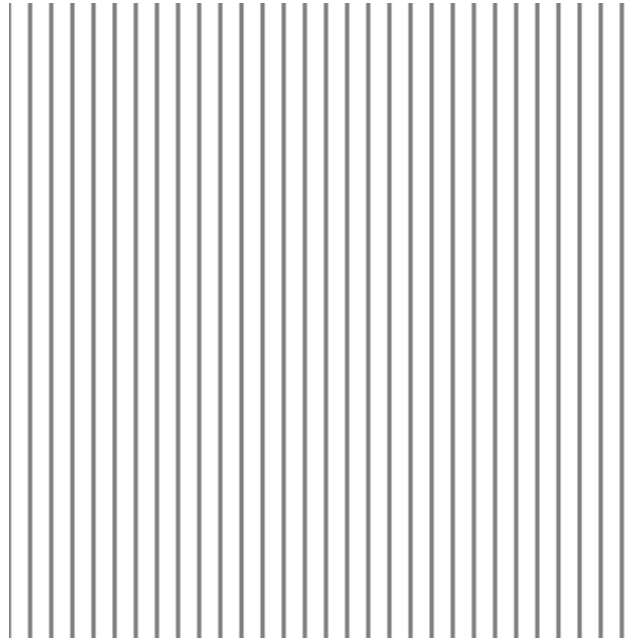


Figure 7.2: Vertical lines with a recursive function.

Let's see if we can replace our recursive `vertLines` function with a `while` loop. Remember our statement of what we want to do:

- Draw vertical lines to fill the canvas, spaced 10 pixels apart.

Thinking about this iteratively instead of recursively means taking an **imperative** “shampoo instructions” approach instead of a **functional** “recurrence relation” approach. On a shampoo bottle, we often see the instructions “Wet hair, lather, rinse, repeat if desired.” See how these are all commands? Where a recurrence relation describes *relationships between steps*, an imperative definition describes *the steps themselves*. Here, we will do something similar:

1. Set the current position `x` to be `canvas.width`
2. Draw a line at the current `x`
3. Subtract 10 from `x`
4. Repeat from step 2 as long as `x >= 0`.

The key difference here, the thing that allows loops to work for iteration, is the fact that we are doing something repetitive, with small changes each time. In our recursive functions, no variables ever really changed value. Instead, new variables were always created based on the value of other variables. When we use a loop, however, we create a variable whose value changes over time. For example, we can make an `x` variable that “keeps track” of the current line position.

Our loop will follow exactly that pattern. Here is a new `vertLines` function that uses a `while` loop. See if you can match the steps above with the loop code below:

:javascript:

```
function vertLines(ctx) {  
  var x = ctx.canvas.width;  
  while (x >= 0) {  
    ctx.moveTo(x, 0);  
    ctx.lineTo(x, ctx.canvas.height);  
    x = x - 10;  
  }  
  ctx.stroke();  
}
```

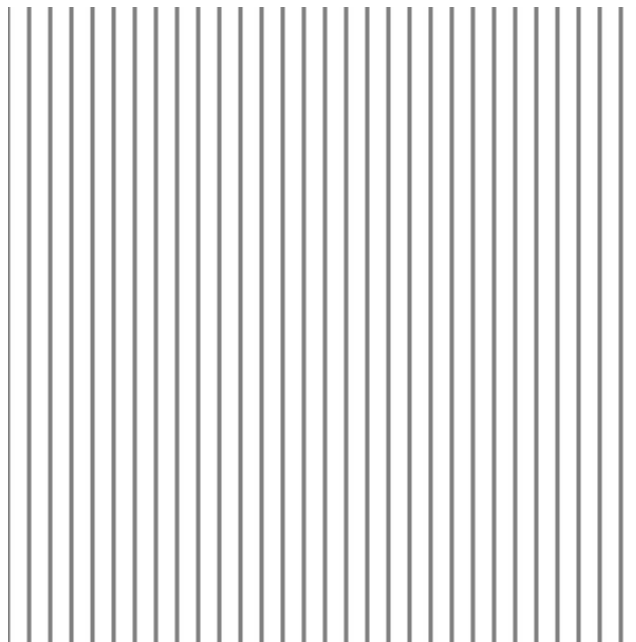


Figure 7.3: Vertical lines with a `while` loop.

Did you get the lines matched up with the code? Here's how I would do it, with the steps in comments off to the side:

:javascript:

```
function vertLines(ctx) {  
  var x = ctx.canvas.width; // 1. Set the current position.  
  while (x >= 0) {          // 4. Repeat as long as x >= 0.  
    ctx.moveTo(x, 0);        // 2. Draw the line.  
    ctx.lineTo(x, ctx.canvas.height);  
    x = x - 10;              // 3. Subtract 10 from x.  
  }  
  ctx.stroke();  
}
```

This is a very straightforward way to do multiple things in order, and for most folks it is a more natural way of thinking about iteration; there is no need to first develop a recurrence relation. Recursion is the best way to approach some problems, still, but not this one. How can you decide, though? A simple rule for languages that have loops is this:

Use loops when possible, fall back to recursion when it's a natural fit.

Most of the time that means “just use loops”.

Another benefit of loops is shorter, clearer, more readable code. You can see more of the pattern all at once, without having to trace through function calls.

Incidentally, mutability is a major distinction between “functional programming” (describe what you want to *compute*) and “imperative programming” (describe what you want to *do*). In pure functional programming, every value is the result of a function call—including things like `+` that don't look like functions, but can easily be thought of as such—and every variable is assigned exactly once. The value of a variable never changes. That's true for our recursive `vertLines` function: we might *pass in* computed values for `width`, effectively initializing a brand new variable for that function call, but we never *reassign variables* to hold new values.

Keeping things unchanged (*immutable*) in a pure functional way requires recursion. Allowing variables to be changeable (*mutable*) opens the door to iteration using loops. That's typical in imperative languages, and most mainstream languages are fundamentally imperative with numerous helpful functional concepts sprinkled in. JavaScript is very much like that.

Loop iteration is often simpler to understand than recursion, but that doesn't mean that it's better in any *fundamental* way. There are pure functional languages that have no concept of loops at all, and they are every bit as powerful and expressive as imperative languages. In fact, it is sometimes convincingly argued that they are *more* expressive than mainstream imperative languages.

The key is to use the best tool *for the language you are working in right now*: when it's possible and clearer to do so, use loops.

Getting back to our concrete problem, you can probably see how you would adapt this function to draw horizontal lines. Give that a try, now!

Incrementing Variables

As you might expect, the various steps in a loop description have names, which we associate with the code above:

- **Initialize:** set the starting value `x = ctx.canvas.width` once, at the beginning.
- **Check:** check the condition to see if we should continue, if `x >= 0`.
- **Increment:** change the value of `x` to `x - 10` to get closer to a false condition.

The initialization and condition parts are somewhat familiar now, because of previous discussions on variable assignment and `if` statements. Incrementing a variable can be surprisingly interesting, though! This gives us a chance to take a quick detour into assignment operators². So far we have seen and used exactly one assignment operator: `=`. It turns out that there are several more.

Why might we need an alternate assignment operator? Let's take another look at that funny assignment at the bottom of our loop:

:javascript:

```
x = x - 10;
```

That assignment references `x` twice. It reads from it, subtracts 10, then stores it. It turns out that programmers really hate repeating themselves, so a shorthand was developed for this kind of thing. It's called a "subtraction assignment" and it looks like this:

:javascript:

```
x -= 10;
```

That's basically the same as what we had, but we only mention `x` once. The nice thing about it is that it's obvious at a glance that `x` is being modified based on what it used to be. Clutter has been removed, clarifying the most important information. It now reads like "`x` is reduced by 10". There is a similar "addition assignment" written `+=`, and there are several more.

For the extra common case of adding 1 to a variable, we have these tantalizing possibilities that all do the same thing:

:javascript:

```
x = x + 1;
x += 1;
x++;
++x;
```

The last two are new in this discussion: `x++` and `++x`. They are called **increment operators**³ and are equivalent to each other as written: they add 1 to your variable. There are similar **decrement operators**⁴ that subtract 1: `x--` or `--x`.

Good code will not exercise the difference between `++x` and `x++` because good code doesn't use the value of an assignment expression. It is a source of subtle bugs to do so.

What is that difference, though? The difference between the prefix (`++x`, `--x`) and postfix

²https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Assignment_Operators

³[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Arithmetic_Operators#Increment_\(\)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Arithmetic_Operators#Increment_())

⁴[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Arithmetic_Operators#Decrement_\(\)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Arithmetic_Operators#Decrement_())

(`x++`, `x--`) versions of the **increment** and **decrement** operators lies in what happens when they are treated as *expressions*, such as during assignment: `y = x++` or `y = ++x`. The first one assigns `y` to the value of `x`, *then* increments `x`. The second one increments `x` first, then assigns `y` to the *newly incremented* value of `x`. The difference in these cases is what `y` is after they are done.

This means you are doing two nontrivially related assignments in one statement, and that means clarity suffers. Don't do it. It can be a source of subtle bugs. Instead, always use increment and decrement operators as statements rather than as expressions that produce a value. Thus, a better way of writing `y = ++x` is `++x; y = x`, and a better way of writing `y = x++` is `y = x; ++x`. These are far clearer.

That was a long-winded way of getting to the fact that instead of writing `x = x - 10`, we could and usually would write `x -= 10`.

Early Stopping And Skipping Stuff

All loops in JavaScript, including `while` and the `for` loop we will discuss below, understand two special keywords: `break` and `continue`. These are not used much in this text, but it is important to be aware of what they do, starting with `break`.

In a nutshell, `break` says to the interpreter, “Stop this loop right now and jump to the end.” Thus, this loop will terminate after outputting a single “hello”, even though its condition is never false:

:javascript:

```
while (true) {
  console.log("hello");
  break;
  console.log("Never let them see you print.");
}
```

Note also that it stops exactly where the `break` is located and skips over the rest of its body. The second `log` function is never called.

The `continue` keyword also interrupts the flow of the loop, but instead of jumping to the end, it jumps back to the beginning and attempts to do the next iteration. It's like saying, “Hey, this trip through the loop has finished its useful work, so start the next trip through it right now.”

In the case of a `while` loop, that just means evaluating the condition again and, if true, going through the loop again.

As mentioned, we won't use `break` and `continue` very much in this course, but they're very useful. For example, if you want to know whether your value appears in an array, you can look at each index one at a time until you find the value you are looking for. Once you have found it, you don't need to keep going: you already know it's there. In that situation you would use `break`. To illustrate that point, look at this code and see if you can tell how it works:

:javascript:

```
var values = ['a', 'b', 'c', 'd'];
var i = 0;
while (i < values.length) {
  if (values[i] === 'c') {
    break;
  }
  ++i;
}
console.log("Loop exited at index " + i);
```

There's that pattern again: initialize, check, do stuff, and increment. But in the “do stuff” portion, we end early if we find what we were looking for. That's the way you use `break` in a loop.

Do-While Loops

There's a variation on the `while` loop that you won't see very often. It's called a `do while` (or just a `do`) loop. It looks like this:

:javascript:

```
do {
  // The body runs once,
  // Condition finally checked,
  // Runs perhaps again.
} while (condition);
```

It works just like the `while` loop, but the body is guaranteed to execute at least once. It is structured to make that somewhat easy to see: we `do` the stuff in the body, then we check the condition to see if we want to `do` it again. This actually matches the whole “shampoo instructions” idea a little more closely: lather and rinse, repeat if desired. Like this:

:javascript:

```
do {
  lather()
  rinse()
} while (desired())
```

You will see these occasionally and may find a handy use for them from time to time, but they are not as common as the others.

For Loops

Using a `while` loop is great for iteration because it is easy to understand: it's basically a repeating `if` statement. Our `while` loops have always been shaped like this:

:javascript:

```

var i = 0;
while (i < N) {
    // do stuff
    i++;
}

```

The full pattern, as mentioned earlier, is **initialize-check-increment**, and you can see those three things in the loop: `i` is **initialized** to 0, the condition **checks** that it is less than `N`, and it is **incremented** by 1 at the bottom of the loop. Because this particular thing is so common, and because it is equally common to do silly things like forget to increment your variable inside the body of a `while` loop (causing it to run forever), there is another loop that you will see even more often. It is called a `for` loop.

You can think of it as “Do this thing `for` a certain number of iterations”, though it is much more flexible than that. Let’s have a look at the shape of a `for` loop that prints vertical lines on a canvas:

:javascript:

```

for (var x = ctx.canvas.width; x >= 0; x -= 10) {
    ctx.moveTo(x, 0);
    ctx.lineTo(x, canvas.height);
}
ctx.stroke();

```

This behaves just like the `while` loop we wrote earlier:

:javascript:

```

var x = ctx.canvas.width;
while (x >= 0) {
    ctx.moveTo(x, 0);
    ctx.lineTo(ctx.canvas.height);
    x -= 10;
}
ctx.stroke();

```

In the `for` loop, however, all of the loopy stuff is in the first line and only the meaty stuff is in the body. This makes it much easier to remember things that you might otherwise forget, like initializing and incrementing the loop variables; in a `for` loop those steps are right next to the terminal condition, instead of spread around in a similar `while` loop. Additionally, it is easy to see how the loop *progresses* without looking at what it *does*; everything is right there in one place. Let’s look at how these loops work right next to each other, in abstract:

:javascript:

```

// While loop:
initialize x
while (condition) {
    body
}

```



```

    increment x
}

// For loop:
for (initialize x; condition; increment x) {
    body
}

```

These loops behave differently when `continue` is used. Looking at our specific line-drawing loops above, if we were to `continue` in the `while` loop, we would just check `x >= 0` again and possibly execute the body. If, however, we were to `continue` in a `for` loop, we would *perform the `x -= 10` increment and then* check `x >= 0`. This is one of the powerful and useful things about `for` loops: every time through the loop, you can be sure that the “increment” will be run: in this example, `x` will be sure to change even when `continue` is used. That’s not true for `while` loops, where a badly placed `continue` can cause an infinite loop.

Finally, there are other forms of `for` loop: `for-in` and its younger, more talented cousin `for-of`. We won’t go over those just yet, though. This “initialize; check; increment” form is very useful and common, and we can do just about everything we need with it.

The loop variable `x` can, of course, be named whatever you want; `x` is just used as a simple example here. The name `i` is very common when it’s the index of something like an array.

The thing to take away from this is the transformation between `while` and `for`. Let’s redo our grid using this new knowledge, repeating all of the setup code to make it simpler to remember past lessons:

:html:

```

<canvas id="drawing" width="300" height="300"></canvas>
<script>
function vertLines(ctx) {
    for (var x = ctx.canvas.width; x >= 0; x -= 10) {
        ctx.moveTo(x, 0);
        ctx.lineTo(x, ctx.canvas.height);
    }
    ctx.stroke();
}

var canvas = document.getElementById('drawing');
var context = canvas.getContext('2d');

vertLines(context);
</script>

```

You should be able to run this. Can you see how you would add a horizontal lines function? Or perhaps you could change `vertLines` to be called `gridLines` and put a second loop right inside of it.

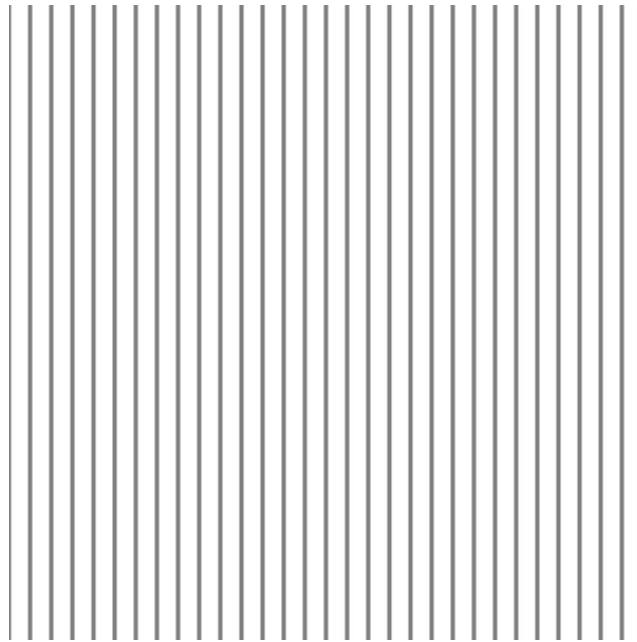


Figure 7.4: Vertical Lines.

All elements of the `for` loop are optional. The condition is assumed to be `true` if left off, so “forever” can be spelled `for(;;)`, but you rarely see that in practice. Usually you will see “forever” spelled as `while(true)`.

It’s useful to know what it means to leave out elements, though. You will often see loops that leave off the initializer in real code.

Without even looking at each loop’s body, you can tell basically how the loop marches along. You can see immediately in the `for` loop which variable is changing, what will cause the loop to finish, and how much the loop variable changes each time. Now we can make a nice, relatively simple grid-drawing routine, with no recursion in sight. Neat!

Tiling a Canvas

Drawing a grid can certainly be useful, but not very exciting. Let’s try something else with our newfound understanding of loops. Let’s make our canvas into more of a patchwork quilt.

It’s time to start a new program. By now, you know the drill. Make a file, put stuff in it, save it as `some_name_you_like.html`, and open it in your browser. The standard canvas stuff will be used again, so we will start with our familiar skeleton:

:html:

```

<canvas id="drawing" width="300" height="300"></canvas>
<script>
var canvas = document.getElementById('drawing'),
    ctx = canvas.getContext('2d');
// Fun things await.
</script>

```

We will start by creating a loop in place of `// Fun things await.` Let's make a single row of filled rectangles, all the same color, separated by a single pixel of space. We will stick with squares 10 pixels apart:

:javascript:

```

// Constants should be named,
// Keeping magic numbers out,
// Often UPPER_CASE:
var SQUARE_SIZE = 10;

// Draw a row of squares.
for (var x = 0; x < ctx.canvas.width; x += SQUARE_SIZE) {
    ctx.fillRect(x+1, 1, SQUARE_SIZE-1, SQUARE_SIZE-1);
}

```



Figure 7.5: A row of squares.

Note that this time we used a more common way of writing a `for` loop: starting low and ending high. That's a bit different than what we did for grid lines earlier, where we started high and ended low to keep things similar to the recursive version, but the principles are

all exactly the same.

Since this is still new, let's go over it again in detail. As can be seen in the program, we are being careful to not put magic constants everywhere. If we used the number 10 instead of `SQUARE_SIZE`, we would see 10 and 9 in multiple places. Instead, we can see the relationship between all of the numbers, and if we want to use squares of size 5 at some point, it's just one change instead of several. That's an important practice: no magic numbers. The most common exceptions to this are 0 and 1, of course.

The loop is the interesting part. We start at the left side, with `x = 0` in the initializer position. We only execute the loop body if `x < ctx.canvas.width`, since otherwise we would be drawing outside of the canvas's extent where anything we do is lost. Finally, in the increment position we have `x += SQUARE_SIZE`, which advances us (in this example) 10 pixels to the right.

We use the words “left” and “right” in that explanation, but that's not really what `x` means, at least not yet. It's how `x` is used that gives it meaning to us (the computer assigns no meaning to anything at all—it is only doing computations; that is all it knows how to do—we humans assign meaning to the output of those computations, that happens to look to us like colored shapes), and the loop body is where that happens. There, a rectangle 1 pixel to the right of `x` and one pixel down is drawn. We draw just shy of `SQUARE_SIZE` since we already used a pixel for the space on the edges.

Each time through the loop, it draws another rectangle, the same size as all the others, but shifted to the right. That's a good start!

Nested Loops

What about filling the rest of the rows? This only does the first one. For that, we will use a **nested loop**. In the same way we used the term “nested” with the `if` statement, here we use it to mean “one loop inside of another”. Basically, we need to run our row-filling loop once for each row, and each row starts with a different value for `y`:

:javascript:

```
for (var y = 0; y < canvas.height; y += SQUARE_SIZE) {
  for (var x = 0; x < canvas.width; x += SQUARE_SIZE) {
    ctx.fillRect(x+1, y+1, SQUARE_SIZE-1, SQUARE_SIZE-1);
  }
}
```

We have taken our original loop and moved it to the inside of another one. We had to change one thing in that inner loop, though: instead of drawing at `(x+1, 1)`, we now draw at `(x+1, y+1)`. Our original version assumed that we were just drawing on the top row.

What does the outer loop do? It starts us at the top with `y = 0`, does everything inside over and over again so long as `y < canvas.height`, and always adds `SQUARE_SIZE` to `y`

when it finishes one iteration.

Remember that the body of a loop is executed multiple times, with different values for the loop variable each time. So, our `for (var y ...)` loop is going to execute its entire body several times, once for each row that we want to create. But in this case the body of the loop is another loop! This “inner” loop outputs a square for each `x` coordinate, starting at 0 and ending when it reaches the width of the canvas.

This is something you are going to want to run, both in and out of the debugger. In the debugger, put a breakpoint on the call to `fillRect` and watch what happens to `x` and `y` each time you advance the program.

To make it easier to experiment, here is the full, surprisingly short program:

:html:

```
<canvas id="drawing" width="300" height="300"></canvas>
<script>
var canvas = document.getElementById('drawing');
var ctx = canvas.getContext('2d');
var SQUARE_SIZE = 10;

for (var y = 0; y < canvas.height; y += SQUARE_SIZE) {
  for (var x = 0; x < canvas.width; x += SQUARE_SIZE) {
    ctx.fillRect(x+1, y+1, SQUARE_SIZE-1, SQUARE_SIZE-1);
  }
}
</script>
```

Load it up and play around!

Color Cycling Using Else-If

It would be more interesting to have more than one color represented here. What if we defined a few colors and then cycled through them to get a rainbow effect? How could we even do that?

There are a couple of ways to go about it. Here is one idea. Inside of the inner loop, we could change the color based on what it was the previous time through, like this:

:javascript:

```
var color = 'red';
for (var y = 0; y < canvas.height; y += SQUARE_SIZE) {
  for (var x = 0; x < canvas.width; x += SQUARE_SIZE) {
    ctx.fillStyle = color;
    ctx.fillRect(x+1, y+1, SQUARE_SIZE-1, SQUARE_SIZE-1);

    // Let's pick a new color before we draw again!
    if (color === 'red') {
```

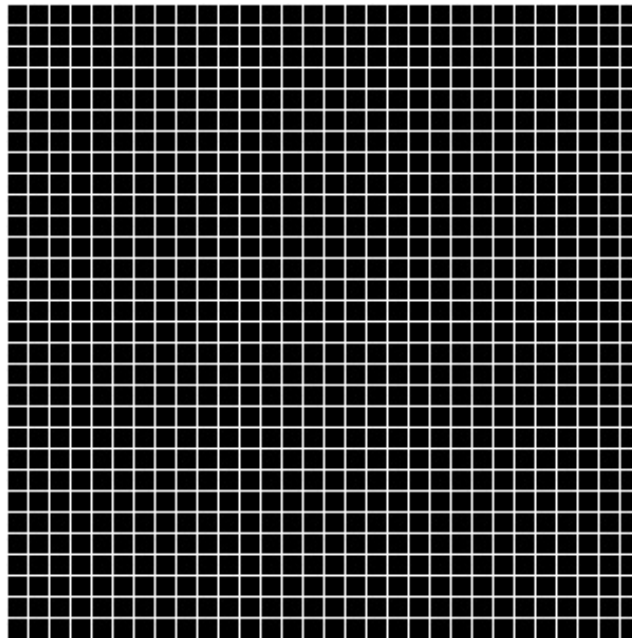


Figure 7.6: All tiled.

```

    color = 'orange';
  } else if (color === 'orange') {
    color = 'yellow';
  } else if (color === 'yellow') {
    color = 'green';
  } else if (color === 'green') {
    color = 'blue';
  } else if (color === 'blue') {
    color = 'purple';
  } else if (color === 'purple') {
    color = 'red';
  }
}
}

```

Do you see how this works? We start out with the color “red”, and we set the context color and draw a square. Then we check the current color and change to the next one for the next time through the loop. In this case, if it was “red”, it will be “orange” next. If it was “orange”, it will be “yellow” next, and so on through the colors of the rainbow until we get to the end, where “purple” goes back to “red” and the colors repeat.

We don’t just use `ctx.fillStyle` to determine what color to use next, because it actually transforms our nicely named colors into RGB (Red-Green-Blue) specifications, and those don’t have nice names like what we start with. A color like “red”, for example,

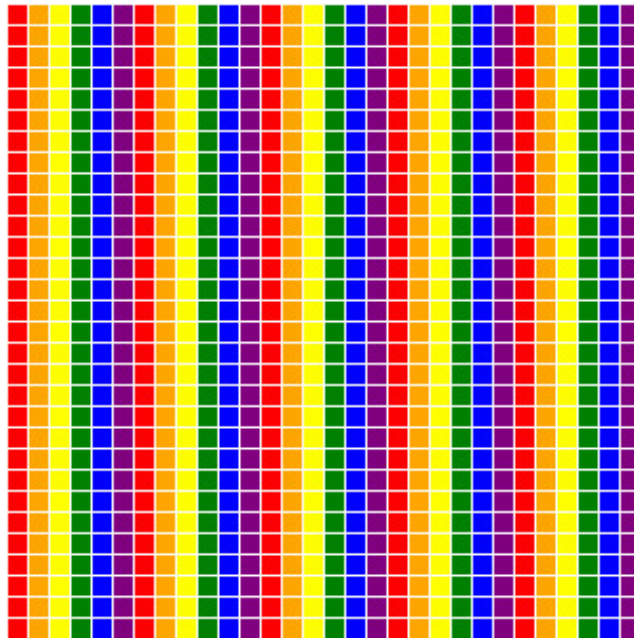


Figure 7.7: Color Cycling.

becomes `'#ff0000'` when we read it back. The use of hexadecimal for colors is a topic for another time; just know that it is usually best to keep track of things in your own variables rather than relying on browser objects to give you back exactly what you gave them. Browser document elements are notorious for not working in intuitive ways all the time (in this particular instance, the surprise is due to not “round-tripping” the value properly: you read something different than you write), and this is one instance of that.

Also, for those interested in more vocabulary, what we have created here is a very simple version of a **state machine**: something that determines what to do next by the current state. When it does the next thing, the state changes, and it has to re-evaluate based on that when it goes around again. This particular state machine changes state in this order: “red, orange, yellow, green, blue, purple” and then it repeats.

One thing that can be fun is changing the square size. If you change it to something like 15, the pattern will change. The colors are cycling based on individual squares, not based on rows and columns: each square gets a different color, and since the size of the square affects how many appear in a row, you can use that to alter the overall pattern.

Switch It Up

There is another interesting conditional statement in JavaScript called the `switch` statement. It can basically replace an “else if” pattern, but it has some additional and interesting quirks. Let’s start by just doing a straightforward replacement of our `else if` chain:

:javascript:

```

var color = 'red';
for (var y = 0; y < canvas.height; y += SQUARE_SIZE) {
  for (var x = 0; x < canvas.width; x += SQUARE_SIZE) {
    ctx.fillStyle = color;
    ctx.fillRect(x+1, y+1, SQUARE_SIZE-1, SQUARE_SIZE-1);

    // Let's switch up the color, and switch *on* it, too.
    switch (color) {
      case 'red':
        color = 'orange';
        break;
      case 'orange':
        color = 'yellow';
        break;
      case 'yellow':
        color = 'green';
        break;
      case 'green':
        color = 'blue';
        break;
      case 'blue':
        color = 'purple';
        break;
      case 'purple':
        color = 'red';
        break;
    }
  }
}

```

This looks a bit different than what we had before, but it is very close to the same thing. The first thing to notice is that we only have to mention `color` once to check its value. It's up there in the switch header. Then there are all of these `case` sections as well as a ton of `break` commands. What can we make of all that?

Here is the basic structure of a switch statement:

:javascript:

```

switch(checkValue) {
  case value1:
    body1
    break; // skip all the rest of the cases
  case value2:
    body2
    // no break---falls through to next body
  case value3:
    body3
    break;
  default:

```



```
    bodyDefault // executes when nothing else matches
}
```

Note that once a case matches, the interpreter will run everything it finds until it sees a `break` or the end of the `switch` statement. In other words, if one of your cases doesn't use `break`, execution will continue into the next case body. This is called “falling through” to the next case and is a sometimes useful behavior, but the general rule is, “Don't do it.” Fall-through is pretty dangerous. Let's look at how it works below, but remember that this is not recommended. It is better, in this case, to repeat yourself and keep things clear for the reader: don't forget your `break` statements in case bodies!

:javascript:

```
switch(color) {
case 'red':
    console.log("matched red");
case 'blue':
    console.log("running blue");
    color = 'yellow';
    break;
case 'green':
    color = 'blue';
    break;
// etc.
}
```

If the color is "red" in this example, the console will output this:

:console:

```
matched red
running blue
```

The “red” case has no `break` in it, so both 'red' and 'blue' case bodies are executed. This is a very unfortunate default behavior because it makes the most common situation the most verbose. We will see alternative approaches using data structures and closures that don't suffer from this shortcoming, and you may decide that you like them better as we move one. There are trade-offs involved in all possible choices.

Note that `switch` is implicitly doing an equality check, which brings up the question of which kind of equality it is using. It turns out that `switch` uses `===`, which is what we have been using all along. This is the safest equality operator, as it compares types as well as values, and does not implicitly coerce strings to numbers and vice versa. Thus, the output of the following program is “safe”:

:javascript:

```
switch("50") {
case 50:
    console.log("unsafe");
}
```

```

    break;
default:
    console.log("safe");
    break;
}

```

Arrayed in Color

If we had even more colors, even our nicer `else if` expressions would quickly get unwieldy and hard to follow (note: this may have already happened). Also, we keep repeating ourselves—we have to mention a color both when we check for it (to change it) and when we set it to the new value. That’s two mentions for every color, and lots of chances to misspell something. In fact, I did exactly that when writing this, multiple times, so for me it’s not merely an academic concern.

There is a principle in programming called the “Don’t Repeat Yourself” (DRY) principle. The idea is this: if you are repeating yourself, you are introducing opportunities for mistakes as well as wasting effort. Where possible and sensible, don’t repeat yourself. Let’s see how we could apply that here.

What we want is a list of colors that we care about, and a way of cycling through that list without repeating ourselves. An ordered list of colors...that sounds an awful lot like an array. What if we used one of those? Let’s see what that might look like:

:javascript:

```

// Create an ordered list of colors: an array.
var COLORS = [
    'red',
    'orange',
    'yellow',
    'green',
    'blue',
    'purple',
];

// Initial color index. This will be red.
var colorIndex = 0;

// Do the normal nested-loop thing to make a grid.
for (var y = 0; y < canvas.height; y += SQUARE_SIZE) {
    for (var x = 0; x < canvas.width; x += SQUARE_SIZE) {
        // Get the current color and fill the rectangle.
        ctx.fillStyle = COLORS[colorIndex];
        ctx.fillRect(x+1, y+1, SQUARE_SIZE-1, SQUARE_SIZE-1);

        // Change to the next color.
        colorIndex++;
    }
}

```

```

    // If we went off the end of our array, go back to
    // the beginning.
    if (colorIndex >= COLORS.length) {
        colorIndex = 0;
    }
}
}
}

```

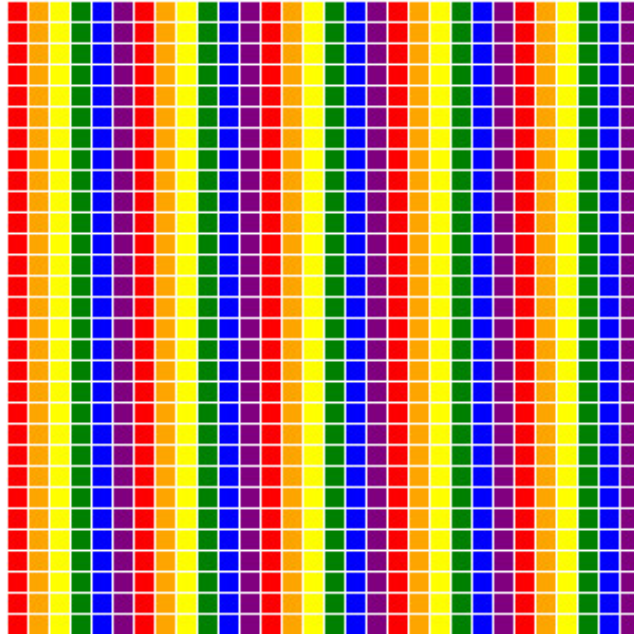


Figure 7.8: Repeating colors.

We just created an array of color values and cycled through it without an `else if` in sight. That's interesting. How does it work? The main attraction is this line:

:javascript:

```
ctx.fillStyle = COLORS[colorIndex];
```

What is that doing? It is setting the drawing color based on the contents of our array and the current color index. The `colorIndex` is a number telling us which position in the array we are interested in. The `colorIndex` starts at 0, so the first color we will use will be 'red' because `COLORS[0]` contains the value "red" (remember, the first element of an array is at index 0). If you are comfortable with that, you can hopefully see, for example, that `COLORS[3]` evaluates to 'green'.

Armed with that information, you are now ready to understand how we are cycling through the colors using an array instead of a `switch` statement. Let's look at the last few lines of the inner loop, starting with this one:

:javascript:

```
colorIndex++;
```

Remember our friend the increment operator? That's our friend right there. It changes `colorIndex` to be one more than it used to be. It's equivalent to `colorIndex += 1` or `colorIndex = colorIndex + 1`.

After we increment `colorIndex`, we check whether it has gotten too big for its array, and if it has, we set it back to 0 to start over in our list of colors:

```
:javascript:
```

```
if (colorIndex >= COLORS.length) {
  colorIndex = 0;
}
```

Thus, for each square we draw, we move the color index forward one, pointing at the next color in the array. If we get past the end of the array, we start back at the beginning. It's pretty simple once you see what is happening. It is also elegant: we list the colors in exactly one place, and then we just cycle through them. It is also extremely easy to add more colors or rearrange the colors that are there: just change the array, and the code adapts without any other changes!

Modulus

There is another common way of doing this kind of repetitive cycling (where we start over at 0 after reaching some terminal value), and that is using the modulus operator `%`. Modulus (frequently shortened to “mod”) is basically the same as taking the remainder after division. Let's look at what that means by using a table of values for dividing by 5 and taking the modulus of 5.

The remainder is zero when our number is perfectly divisible by 5, then it counts upward as our number counts upward. After it reaches 4, it resets to 0 and counts up again.

Input	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
/ 5	0	0	0	0	0	1	1	1	1	1	2	2	2	2	2
% 5	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4

That is exactly the behavior we want for our array, but using the array's length instead of the number 5! Let's look at what happens to our color cycling when we use modulus (remainder):

```
:javascript:
```

```
var COLORS = [
  'red',
  'orange',
  'yellow',
```

```
'green',  
'blue',  
'purple',  
];  
  
var colorIndex = 0;  
  
for (var y = 0; y < canvas.height; y += SQUARE_SIZE) {  
  for (var x = 0; x < canvas.width; x += SQUARE_SIZE) {  
    ctx.fillStyle = COLORS[colorIndex % COLORS.length];  
    ctx.fillRect(x+1, y+1, SQUARE_SIZE-1, SQUARE_SIZE-1);  
  
    colorIndex++;  
  }  
}
```

That is nice and concise, and once you are used to the modulus operator, it is also fairly obvious what is going on: no matter how large `colorIndex` becomes, we will always get a value out of it that fits inside our array, because we are using the array length as an argument to the modulus operator. It is quite popular for this sort of thing, and we will be seeing it again.

If we had an exceptionally large set of things we wanted to do, something in the range of 2^{53} , for example, this code would eventually fail. It's easily fixed by resetting `colorIndex` to the result of the modulus operator after incrementing it; then it just keeps resetting to zero at appropriate moments.

Let's recap. In this section we were introduced to nested loops, indexing arrays, cycling through values, and using modulus to make that simpler. That's quite a bit of stuff, and we are finally doing something interesting with our canvas!

Random Numbers

Speaking of interesting, let's make things less predictable, literally. Let's change this up so that a random color is picked from the array every time instead of just the "next" one.

There are many kinds of randomness, and a discussion of sources of randomness and their various qualities is beyond the scope of this course. We will just blindly use the provided `Math.random()` function and assume it is "random enough" for our needs here, which it is.

Some things are more random than others. Getting the radioactive output of a banana (yes, that's a thing) is much more random than grabbing the current time every so often, for example. Computers usually have multiple means of getting randomness, and more correct means are usually more costly in time. We won't get into that here, but it can be useful to know: randomness is its very own field of study in computer science, and it is

fascinating.

“Pseudo-random” basically means that a number appears usefully random, but can actually be predicted if you know the right initial conditions. Such numbers used to be created from books full of number tables, but now we have interesting algorithms for generating them instead, and we can make their seeds (initial conditions) based on things like the clock or the state of the network to reduce predictability.

The `Math.random` function takes no arguments and produces a pseudo-random real number between 0 and 1, never quite managing to include 1. Let’s use the console to try it (your numbers will be different—these are, after all, random):

:console:

```
> Math.random()
< 0.5343124314422523
> Math.random()
< 0.2841169425251435
```

If you keep doing that, you will keep getting different answers. That’s helpful, but how do we use it for color cycling?

For that, we will also want to make use of two operators: multiplication and **floor**. We can easily use multiplication to expand the range of our random numbers from `[0, 1)` to, say, `[0, COLORS.length)`. That’s very useful, but because we’re dealing with real numbers, it will include things like `3.6`, which is not a very helpful array index. To get an integer array index from a number like that, we use `Math.floor`.

Mathematical range notation uses square brackets `[]` to indicate “inclusive” and round brackets `()` to indicate “exclusive”. The range `[0, 1]` would be all numbers between 0 and 1, including both 0 and 1. The range `[0, 1)` is the same range, but excludes the number 1. It includes numbers arbitrarily *close* to 1, but never quite gets to 1.

The “floor” of a number is essentially the closest integer at or below it. Thus, the floor of `4.2` is 4, the floor of `6.8` is 6, the floor of `-1.2` is -2, and the floor of any integer is just itself. You can think of it as a rounding operator that always rounds down. Similarly, the corresponding `Math.ceil` (ceiling) function always rounds up.

If we combine these concepts, we can get a random array index. Let’s try some examples in the console (you will get different numbers because they are random). Note that you can repeat the last command by typing the up arrow on your keyboard, thus avoiding having to copy-paste or retype the command multiple times:

:console:

```
> Math.floor(Math.random() * 10)
< 7
> Math.floor(Math.random() * 10)
< 9
```

```

> Math.floor(Math.random() * 10)
< 4
> Math.floor(Math.random() * 10)
< 2
> Math.floor(Math.random() * 10)
< 1

```

What are we doing there? We are stretching the random range from $[0, 1)$ to $[0, 10)$ by multiplying whatever number it produces by 10. If the `random` function gives us 0.2, we multiply by 10 to get 2.0. If it gives us 0.98, we multiply by 10 to obtain 9.8, and so on.

Remember, this means it will produce numbers from 0 all the way up to 10, but it can never quite produce an actual 10, just a number really close to it. After expanding the range with multiplication, we take the floor, guaranteeing that we will only get one of the integers 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9, with no fractional values. If you run that line over and over again (press the up arrow to get the previous command in the console, it makes it much faster), you will see that the number is always one of those 10 numbers from 0 through 9.

That looks perfect! We can use something like that to get a color for our squares. Before we do, however, it might make sense to save our color-cycling code somewhere and start a new program file. Copy the color-cycling code into the new file, and we will make some changes there. That way you will have a record of what you did and you can go back to it if you want.

Ready? Good. Here is a review of what we had already:

:javascript:

```

var COLORS = [
  'red',
  'orange',
  'yellow',
  'green',
  'blue',
  'purple',
];

var colorIndex = 0;

for (var y = 0; y < canvas.height; y += SQUARE_SIZE) {
  for (var x = 0; x < canvas.width; x += SQUARE_SIZE) {
    ctx.fillStyle = COLORS[colorIndex % COLORS.length];
    ctx.fillRect(x+1, y+1, SQUARE_SIZE-1, SQUARE_SIZE-1);

    colorIndex++;
  }
}

```

There are two lines in this code that deal with cycling the colors: the line that initializes the color by setting `colorIndex = 0`, and the line that changes the color by using modulus to compute the next index. When we use randomness, we don't need to keep track of where we

are in the array, since the current color doesn't depend on the previous one anymore, so the `colorIndex` variable is no longer strictly needed. We will use it anyway for clarity, though. We will thus do two things:

- Remove both of those lines (the one that sets `colorIndex` to 0, and the one that changes it). We won't be needing them. Farewell!
- Produce a random `colorIndex` in the beginning of the inner loop body.
- Use that when setting `fillStyle`.

:javascript:

```
var COLORS = [
  'red',
  'orange',
  'yellow',
  'green',
  'blue',
  'purple',
];

for (var y = 0; y < canvas.height; y += SQUARE_SIZE) {
  for (var x = 0; x < canvas.width; x += SQUARE_SIZE) {
    var colorIndex = Math.floor(Math.random() * COLORS.length);
    ctx.fillStyle = COLORS[colorIndex];
    ctx.fillRect(x+1, y+1, SQUARE_SIZE-1, SQUARE_SIZE-1);
  }
}
```

If you run that, you will get a random pattern of squares. That alone is pretty neat, but even cooler is the fact that every time you run it the pattern will be different. Try reloading the page a few times and see what happens to the colors.

Congratulations! You have now learned about randomness, which is an essential component of many computer programs, including games. That means we can start thinking about even more interesting things like animation!

Exercises

Exercise 7-1: Array Basics

Solution on page 419

Write a small program that performs the following steps in order. Before running the program, answer the questions in the steps below and put your answers in the comments:

1. Create an empty array,
2. Push the numbers 1, 2, 3, and 5 onto the array,
3. Add a comment indicating the length you think the array has at this point,
4. Alert the value of `pop`,

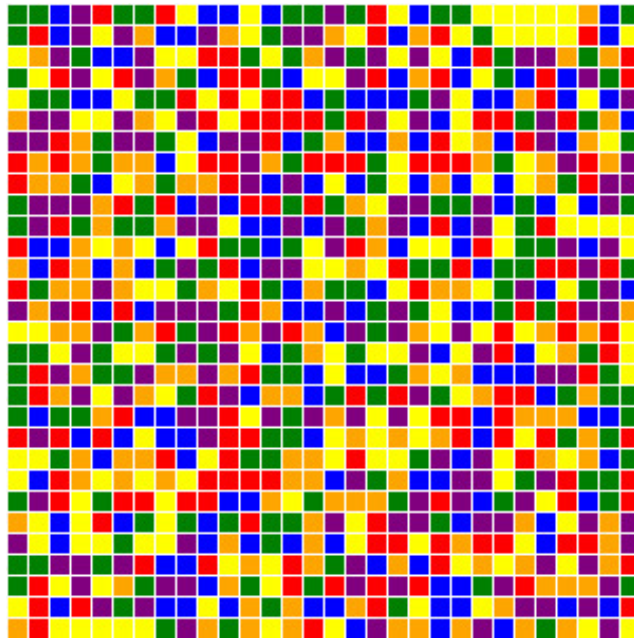


Figure 7.9: Random colors.

5. Add another comment indicating the length you think the array has at this point,
6. Push the values 4 and 5,
7. Add final comment indicating the length you think the array has at this point, and
8. Alert the array.

Exercise 7-2: Slicing Arrays

Solution on page [420](#)

A slice of an array is a copy of a contiguous part of it. For the array `ar` below, write down the value of the following expressions:

:javascript:

```
var ar = [0, 1, 1, 2, 3, 5, 8, 13, 21];
```

1. `ar.slice(0)`
2. `ar.slice(1)`
3. `ar.slice(1, 3)`
4. `ar.slice(4, 5)`
5. `ar.slice(3, ar.length-1)`

Exercise 7-3: Finding elements in arrays

Solution on page [421](#)

After running the “find an element in the array” function below, what is `i` when the value is found? What would it be if we were looking for `'e'` instead of `'c'`?

:javascript:

```

var values = ['a', 'b', 'c', 'd'];
var i = 0;
while (i < values.length) {
  if (values[i] === 'c') {
    break;
  }
  i++;
}

```

Exercise 7-4: Addition and Subtraction Assignment Operators

Solution on page 421

There are various ways to add and subtract values from things. After each line of the following program, show the value for `i`.

:javascript:

```

var i = 1;
i = i + 13;
i = i - 1;
i += 5;
i -= 2;
i++;
i--;

```

Exercise 7-5: Horizontal Lines Using While Loops

Solution on page 422

Write a program that draws a grid using `while` loops. To do so, take the `vertLines` function that uses a `while` loop and use that as a template to create a `hLines` function that looks similar. Put these together to make a grid on a canvas.

Exercise 7-6: While Loops and Arrays

Solution on page 423

Write a function that, using a `while` loop, computes the sum of all elements in a given array. The function can be called whatever you want, but it should accept exactly one argument: the array.

Exercise 7-7: While Loop and Graphing Functions

Solution on page 425

A really interesting thing to do with loops is function graphing. When you go to graph a function like $f(x) = x^2 - 3$, what do you usually do? You probably make something like table of x and $f(x)$ values, draw dots at all of those locations, and then connect them with lines. It turns out that computers are really good at that.

To start, here is some skeleton code. Your task will be to implement the function that plots $f(x) = x^2 - 3$. We have appropriately transformed the canvas coordinates to make this sensible (positive y values go *up*, and the origin is in the *middle* now), but without explanation. If you want to learn more about canvas transforms, we will get to that near the end of the course.

Notes and requirements:

- The function is $f(x) = x^2 - 3$.
- The function should be plotted for integer values $-30 \leq x < 30$.
- Use `lineTo` to draw line segments (remember that the first `lineTo` acts like a `moveTo`, which is helpful here).

:html:

```
<canvas id="graph" width="300" height="300"></canvas>
<script>
var canvas = document.getElementById("graph"),
    ctx = canvas.getContext("2d");

function graph(ctx) {
    // YOUR LOOP GOES HERE
}

// Move origin to center, flip y.
ctx.translate(canvas.width / 2, canvas.height / 2);
ctx.scale(1, -1);

graph(ctx);
ctx.stroke();
</script>
```

Exercise 7-8: While With Break

Solution on page 427

Without using the computer, what does the following code display? Remember that alerting an array converts it to a comma-separated list of its contents.

:javascript:

```
var i = 0,
    vals = [];
while (true) {
    if (i > 10) {
        break;
    }
    vals.push(i);
    i++;
}
alert(vals);
```

Exercise 7-9: While With Continue

Solution on page 427

Without using the computer, what does the following code display? Remember that `%` is the “modulus” or “remainder” operator, so for example, `5 % 3` is 2: the remainder of dividing 5 by 3.

As a second part to this question, what happens if the `i++` immediately before `continue` is removed?

:javascript:

```
var i = 0,
    vals = [];
while (i < 10) {
  if (i % 2 === 0) {
    i++;
    continue;
  }
  vals.push(i);
  i++;
}
alert(vals);
```

Exercise 7-10: Basic For Loops

Solution on page 428

Show the `for` loop that is equivalent to the following code that uses `while`:

:javascript:

```
var i = 0;
while (i < 50) {
  console.log(i);
  i += 10;
}
```

Exercise 7-11: For Loops and Arrays

Solution on page 428

Write a loop that outputs (to the console, one at a time) the value of every item in an array named `ar`.

Exercise 7-12: For and Continue

Solution on page 429

Without using the computer, show the value that will be alerted by the following code. Remember that when dealing with numbers, 0 is considered “falsy” and *everything else* is considered “truthy”. Coupled with your understanding of the `%` operator, you can see when `continue` will be triggered.

After determining what this function does, give an alternative approach that just changes the increment and does *not* use `continue`.

:javascript:

```
for (var i = 0; i < 10; i++) {
  if (i % 2) {
    continue;
  }
  console.log(i);
}
```

Exercise 7-13: Nested Loops

Solution on page 430

Output all of the rows of a truth table with three variables `a`, `b`, and `c`. Use 0 for `false` and 1 for `true`. Use `console.log(a, b, c)` to output each row.

Your output should look like this:

:console:

```
0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1
```

Hint: Approach this by decomposing the problem into smaller ones. How would you output the table for a single variable using a loop?

:console:

```
0
1
```

Now, for each time through *that* loop, how would you output a 0 and a 1 for the next variable to produce this?

:console:

```
0 0
0 1
1 0
1 1
```

From there it should be a similar step to get the complete 3-variable answer.

For bonus points:

There is a way to output a table for *any* number of variables using only two loops, so if you want to try for bonus points, you can write a function that accepts a number of variables and outputs the entire table for that many variables. The foundation of this idea is the fact that every time a variable on the right “rolls over” (from 1 to 0), the variable next to it on the left should change.

Exercise 7-14: Switches

Solution on page 432

Write a function that uses `switch` to return “yes” if given a string “a”, “b”, or “c”, and “no” otherwise. Use `default` as one of your cases.

Note that you can use a “fall-through” pattern to make your code smaller, and empty cases can go on the same line.

Can you do this without using `break` at all? Why?

Exercise 7-15: Random Numbers

Solution on page 433

To produce pseudo-random numbers, JavaScript gives us the `Math.random()` library function, which gives us a different number between 0 (inclusive) and 1 (exclusive) every time we call it. Using that function and basic arithmetic, write a function `dieRoll` that represents a single roll of a 6-sided die: it should produce a number in the set $\{1, 2, 3, 4, 5, 6\}$.

Hint: remember that `Math.floor` can be used to discard everything to the right of the decimal place (for non-negative numbers).

Exercise 7-16: Lab: Objects, Randomness, and the Canvas

Solution on page 435

We are far enough into the course that it is time to start introducing some lab work. This assignment brings together many of the things that we have already learned, including randomness, canvas drawing, and storing data in arrays.

Basic Setup:

For this assignment we are going to plot a distribution over dice rolls. We will use our `dieRoll` function created in the recent homework assignment to roll *two* 6-sided dice over and over again, keeping track of how many times we get each sum. For example, to get a single value, we can do this:

:javascript:

```
var val = dieRoll() + dieRoll();
```

And we might get the number 5 as an answer. That is a valid number for two dice, and we can get it as 1+4 or 2+3, with two ways to get each (because it doesn’t matter which die has

which value). As you might expect if you have ever played games with dice, other numbers are more or less likely. For example, there is only one way to get the number 2, and similarly there is only one way to get the number 12. This should show up in our lab results.

In this example, where we just rolled a 5, we need to keep track of the fact that we just got that one more time. So, we will keep a counter for each possible number, and we will increment it every time we get that number. There are many ways to do this, but for this assignment we will use an array with everything initialized to zero.

Once you have rolled the dice a number of times (say, 1000 or so), the next task is to display the results in a meaningful way, using a **histogram**. This is basically just a bar graph. You will need to figure out how long to make each bar based on the counts obtained from rolling dice. That might mean scaling your results, since it is possible to have more counts than you have space on the canvas. How you do that will be up to you. If you roll the dice 1000 times, then one pixel per roll is probably going to work out fine on our (heretofore) typical 300 x 300 canvas size. You might want to get more accuracy as you go, though, rolling 10,000 or even 100,000 times, and in that case the counts will increase far beyond the bounds of the canvas. Figure out some sort of scaling that keeps them within bounds for this lab so that you can easily change the number of rolls and have the picture automatically adapt.

Some advice: start small. First get your `dieRoll` function working and test it in the console (you should have had a homework assignment for this already, but if you didn't end up doing that, you will need to do it now). Then log what happens when you roll two dice over and over again. After you are satisfied that those small things work, then create an array that has space for numbers up to 12 and store zeros in it everywhere (use `push` in a loop). Output that to the console.

Do you see a pattern? The advice is to do things one step at a time. There are usually ways to tell if little pieces of your program are working before moving on to the bigger parts, and that is how you should proceed. We are stopping short of talking about unit tests, which are a great idea but are a bit beyond where we are right now, but this is part of the way you get testable code that you can be confident in: you get confidence in all of the little pieces before sticking them together into a larger program.

After getting counts, storing them in an array, and outputting to the console, only then should you start thinking about displaying things on the canvas. That will be its own chunk of work, and you want to be sure you have a good feel for what the data will look like before you tackle it.

Requirements:

- Keep track of how many times each value is obtained after rolling two dice multiple times.
- Use an array to store your counts. The index into the array will be the value obtained from rolling two dice. (HINT: this means that there will be a couple of entries that never get any value at all.)

- Use a variable to set how many times to roll, and make it easy to change it.
- Draw a bar graph on a canvas to show the final outcome of the dice rolls.
- Scale the bar graph so that no matter how many times you roll, the graph always fits on the canvas.

Bonus:

- Change your `dieRoll` function to accept a number of sides for the die instead of always assuming 6. Try using larger dice, like 10-sided dice. Assume that they are made to be perfectly fair (sometimes physical geometry doesn't allow this, by the way, but we will pretend that it does in this lab).
- Change your program to roll *three* dice and display the results.
- With that change, it should be relatively easy to see how you could roll *any number* of dice. Change your code to roll `N` dice. What happens if you roll 50 or 100 dice on each turn?

Chapter 8

Timers, Closures, and Animation

Now that we have reached the halfway point of this course, we know about loops (`while`, `do`, and `for`), conditionals (`if`, `switch`, and the `?:` operator), function creation and evaluation, using functions as event listeners (by passing them to other functions to be called later), arrays, basic objects, and randomness, not to mention how to do some basic canvas manipulation. That's actually quite a bit. Congratulations!

You have written some small but very real programs and are now ready to put all of these ideas together. If you were to stop now, you would have experience with some of the most fundamental aspects of programming, and you can actually build some really interesting software just using what you already know.

There are a couple of concepts that still need to be covered, but for the most part we are going to be spending the rest of this course putting together all of the pieces that you have already seen and practiced in previous chapters. We are now officially over the hump.

One Thing At A Time

Whenever your programs interact with the world in a visual way, you might find that some of your intuition fails you. For example, when you drew numerous rectangles in a nested loop, it's possible that you expected (as I did when I was first learning) to see them appear one at a time. Instead, they appeared all at once.

You might think that this is because they just got drawn really quickly, but that, while true, is not the only reason that they appear all at once. You can usefully think of what's happening this way: your code draws to a canvas that is hidden off-screen, then the browser reveals the picture all at once when it is finished.

This is actually very common in computing, particularly when drawing to a screen. Usually what you do does not appear immediately. Instead, it has to wait for some other unseen

process to kick in and do the real work for you; you just tell it what to do when it eventually gets a chance to do it.

There are many good reasons for this behavior. One interesting reason when dealing with JavaScript in the browser is that it is “single-threaded”. This essentially means that only one thing can happen at a time (there is one “thread of execution”, kind of like I only have one “train of thought”). The interpreter can either be running your code, or it can be painting your drawing to the screen, but it can’t be doing both at once. Thus, your drawing function has to finish before other things can have a turn.

To work within this fairly limiting context, much of what we do is **event-driven**. We figure out what we want to do, and we tell the browser when we want it to happen, just like we saw earlier with `setTimeout`. It calls our functions when something happens.

In the case of drawing on the canvas, we already know that we don’t need to do that when triggered by a timer—our first drawing code just did the drawing and that was that. But the *actual work* of drawing on the screen still didn’t happen until our code was finished, which happens when our called function exits, or we go off the end of our script. Our code merely wrote down a list of things to do and handed them off. That’s what calling things like `ctx.fillRect` does: it adds to the interpreter’s to-do list, and the interpreter will start checking things off of that list only after we have stopped adding things to it.

Timeouts Revisited

Animation is a pretty interesting topic all by itself, one that I personally find fascinating. At its core, though, is what the film industry still calls “persistence of vision”. If you have ever watched a movie or an animation, you have experienced this effect.

Every animation or movie that you see is actually composed of a long sequence of still images, shown to you in rapid succession. Various aspects of your visual system, all the way along the path from your eyes to your brain, participate to give the illusion that you are seeing a continuous scene with real motion, when in reality it’s just a bunch of changing light patterns on a screen.

To animate, then, we just show a person a bunch of images, called “**frames**”, in rapid succession. How do we do that with a computer? Well, the first thought that many people have is to use a loop. You can imagine something structured like this:

```
                                :javascript:
while (moreImagesToDisplay()) {
    showNextImage();
    waitABitBeforeTheNextFrame();
}
```

Because JavaScript is single-threaded, looping like that would also cause everything else in

the tab to stop functioning. While our code is running, the browser can't show us anything because it is busy running our code.

To work within this framework, we instead need to write a small amount of code that just draws one image, then asks the browser to call it later. When the browser isn't busy running our drawing code, it can work on actually displaying it to the screen before calling our function to draw the next one.

In other words, we have to write our code to draw one frame at a time and exit. We will keep track of which frame we're on, and the browser will tell us when to draw it. Once the process is started (the browser is ready to call our frame-drawing function for the first time), that function will basically do the following:

- Ask the browser to call us again in the future.
- Draw the current frame.
- Store information about the next frame, but don't draw it.
- Exit.

Every time our function exits, the browser gets a turn to run things like displaying our newly-drawn frame to the canvas. Then the process repeats.

The key to all of this is **events**, which we have touched on very briefly in the past with `setTimeout`. In fact, we can use `setTimeout` for animation!

It's time to start another program, using the familiar file approach. We will be using the canvas, as usual. Feel free to put a border style on it if you would like to see the edges. Here is the skeleton of a program, this time with an `onTick` function that just outputs to the console after 1 second (1000 milliseconds):

:html:

```
<canvas id="drawing" width="300" height="300"></canvas>
<script>
function onTick() {
  console.log("tick!");
}

setTimeout(onTick, 1000);
</script>
```

This should look familiar, but it might be a distant memory at this point. Let's review what is happening:

- We create a function called `onTick` that, when called, says "tick!" in the console. We create it, but we do *not* call it. It doesn't run, so nothing is displayed in the console yet.
- Then we call `setTimeout`, instructing the interpreter to call `onTick` for us one second from now.
- Then our program exits. Our code is no longer running because we reached the end of the `script` block.

At this point, nothing has actually happened, yet. We have merely set things up to happen one second in the future.

The interpreter has dutifully recorded our `onTick` function as something it should call. It then goes about its business, whatever that is, until one second has passed, at which point it finally calls `onTick`. Thus, when you load this in your browser, roughly one second will pass and then the word “tick!” will appear in the console. You can also use the fake `console.log` from an earlier chapter. The `alert` function can work in this instance, but will quickly become annoying as we start triggering multiple events in sequence.

Note that `setTimeout` is a little bit iffy when it comes to precise intervals, a fact that we will deal with later. It doesn’t wait 1000 milliseconds precisely, but waits something reasonably close to it. For smooth animation this inaccuracy will be a bit of an issue, but it’s really easy to reason about and we are just going to run with it for now. Improvements are coming.

If you missed it, you can leave the console open and reload the page to see the effect of the 1-second delay.

The reason we saw something in the console is that our program stopped running and gave the browser time to do its work. What do we mean by “stopped running”? Just that, really. Our program merely sets up `onTick` and calls `setTimeout`. That’s it. Then the overall program is finished.

Part of our program, though, runs later because we told the interpreter, “Run this chunk of code at a specified time,” and it complied. When that function finished running, the browser was able to update the console in the way that we asked. But then it’s done. That function won’t get called again. That’s interesting, but not good enough for animation. Animation needs to run something every frame, not just one time before giving up. How do we do that?

What if, inside of `onTick`, we asked the browser to run our function again later? In other words:

:javascript:

```
function onTick() {  
  console.log("tick!");  
  setTimeout(onTick, 1000);  
}  
setTimeout(onTick, 1000);
```

Now, when the browser runs our program, it does the same things as before: creates `onTick`, calls `setTimeout` and exits. One second later, the interpreter diligently calls `onTick`, which displays “tick!” to the console. But, before our function finishes, it asks the interpreter to call it again in another second. Then it quits again. One second later, as instructed, the interpreter calls `onTick`, which displays another “tick!” to the console, and again asks the browser to call it in one more second. This continues forever, or until you close

the browser tab running your program (you do not have to close the whole browser).

Some browsers won't show multiple lines with the word "tick!" in the console, but will show a count next to it, indicating how many times that exact line has appeared. That's to avoid having a huge scrolling log full of identical lines. You should still see something changing every second in the console.

Hopefully the idea that a function references itself by name from inside its own body is comfortable to you by now, since that is necessary to make this work: a function can pass itself into `setTimeout` without problem.

If any of the above wasn't quite comfortable for you, it's time to review. Do the examples, read the text until you are feeling good about things, and then proceed. We are about to introduce a new and very powerful characteristic of functions.

Closures

JavaScript functions each have a **lexical closure**, and thus the functions themselves are typically referred to as **closures**. This is a fancy way of saying, "They can always see variables defined in their surrounding scope (just outside) no matter when or how they are called." Rather than getting precise with the definition—and there are absolutely some subtleties hiding in there—let's get some experience with the idea, then the definition will make more sense. We will begin by changing our `onTick` function to show a different number each time it is called:

:javascript:

```
function main() {  
  var count = 0;  
  
  function onTick() {  
    console.log("Tick", count);  
    count++;  
    setTimeout(onTick, 1000);  
  }  
  
  setTimeout(onTick, 1000);  
}  
  
main();
```

Without running this yet, can you tell what it will do? Once you have an idea, run it and verify that it works like you would expect it to. Did you get it right?

What is happening with this code? First, the whole program is now inside of a `main` function, which is a good practice, as it keeps our variables locally scoped. We call that function at the end of the script.

Inside of `main`, we define a variable called `count` to keep track of how many times the `onTick` function is called, defined next. Inside of `onTick`, we not only use the value of `count`, we also change it! We haven't made changes to a variable in a surrounding local scope like this before.

If this concept looks simple and intuitive to you, great! It is supposed to be fairly intuitive: functions can always see and manipulate the variables in their **immediate outer scope**, also known as the **enclosing scope**: the scope that contains the function. When you think about it, though, it's kind of remarkable. It means that a function somehow remembers how to access those variables just outside of it even when it is passed elsewhere to be run later, like with `setTimeout`. That makes this a powerful idea indeed, and we will see why as we continue to apply it.

We say that a function's behavior is **closed** over its body and surrounding variables. That's where the name **closure** comes from.

For example, even though our code never calls `onTick` directly, the code that runs inside of `onTick` (a second later) can still access `count`, defined just outside of `onTick`. That variable is still hanging around in memory, known to `onTick` even though `main` itself has long since stopped executing (and will never execute again). Why is it still around? Because the interpreter knows that `count` has to be around in order for `onTick` to use it. The `count` variable is still needed as long as `onTick` is stashed somewhere, so the interpreter tracks it along with `onTick`.

If we were to call `main` again, a new `count` variable would be created, and a new `onTick` function would be created, as well. We would then have two separate `onTick` functions registered with `setTimeout`, each with their own `count` variable attached, that would be writing to the console. In fact, you should try this and see what happens. If you just call `main` twice in a row you will see that everything is getting logged twice. If you delay the second call using something like `setTimeout(main, 500)` the effect will be even more obvious.

Furthermore, the variables in JavaScript are closed **lexically**, meaning that if you want to know where a variable is defined, you can tell by looking at the (lexical) structure of the code. Basically, if you can see it in your scope or any surrounding scope (including the global scope, of course, since that's always visible from anywhere), you can access it.

One notable and important exception to the "you get what you see" rule is the special dynamically-scoped `this` variable. We are avoiding that drama for now, and honestly for as long as possible. Dynamic scopes are usually an unmanageable disease. Lexical scopes won for a reason.

Thus, when we call `setTimeout(onTick, 1000)`, we are indeed passing `onTick` to a routine that will call it later, but what `onTick` contains is not just the code inside its brackets, it also contains information about its surrounding scopes: all of the variables and

functions found there.

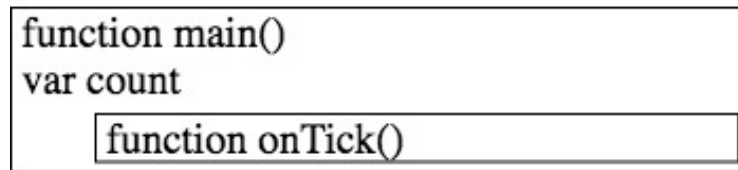


Figure 8.1: All of this is passed around with `onTick`.

Why are we talking so much about something that seems so simple to use? Mostly because understanding gives you the power to make good choices. Knowing that closures carry around references to every variable in every immediately surrounding scope allows you to think about and have some control over how much space those functions use, for example.

Why not just use global variables, since those are visible everywhere? For one thing, global variables are generally a pretty bad idea precisely because they are accessible from anywhere. We don't want to stick everything we know about into the global scope because it might accidentally overwrite something someone else put there. Or someone might do that to us! Or we might even do it to ourselves if we have two functions that want to use the same global name to mean different things.

Global variables are a common source of subtle bugs. Using a closure allows us to manage the scope of our variables a lot more carefully, and that's a good thing. Consider what happened when we moved everything into `main`: now there is only one new thing in the global scope: `main`. Everything else is local and under our careful control. The closure can see and operate on variables in its outer scope, but nothing else can. That's good code hygiene.

In summary, closures are amazing and useful. They remember the variables in their surrounding scopes, which allows us to manage the lifetime of variables that need to live longer than the function call itself without chucking them into the global object, and they remember those things even when they are called from elsewhere. Because they are so useful, we will see them all the time from now on.

A quick aside, just for fun, is in order here. If you want to write a program without putting anything at all into the global scope, you actually can! Instead of making a `main` function, you can create and call an anonymous function in one step, like this:

:javascript:

```
(function() {
  console.log("I'm immediately called and have no name!");
})();
```

This is called an **immediate function**, and the technique is prevalent in real code because of its ability to keep things completely out of the global scope. Do you see what's

going on here? We created an anonymous function using the `function` keyword, and its definition ends with the last curly brace `}`. As soon as it is defined, we call it by appending `()` to the end. That's it!

There is one more detail, actually, and it's this: the function is defined inside of parentheses. Why? Because unless we make it part of an expression, it isn't really defined in place to create a value, and the above looks like a syntax error. You will also see it done this way:

:javascript:

```
(function() {  
  // stuff  
})();
```

See how the function is defined inside of parentheses? Those parentheses bracket an expression, which in this case is an anonymous function. Once the parenthesized expression is closed, that function exists, and then we call it. If it helps you to think about things that way instead, then it's a fine way to go, too. Either way works, because it sends a message to the interpreter that it needs to create a function value and call it.

Finally, you can even create self-referential functions that don't create any new global names by just giving the function a name when you create and call it, like this:

:javascript:

```
(function onTick() {  
  console.log("Global:", window.onTick);  
  console.log("Local:", onTick);  
  setTimeout(onTick, 1000);  
})();
```

This outputs `undefined` for the first line and the function's contents for the second. That can be quite helpful: it shows that we defined a function that is only visible to itself, then started it up without it bleeding into the global scope.

Note: this approach to scope creation is falling out of favor now that JavaScript is getting more implementations with *modules*, so we may at times be a bit cavalier about global variables, just to avoid cluttering things up in this book. You will see this technique sometimes in example code online, though, so it's good to know what it does.

Simple Timeout-Based Animation

We can combine closures and timeouts with drawing on the canvas to make an animation that gives the illusion of movement. That's where things start to get fun, and is central to the operation of many games. Let's see if we can make a little spaceship that launches into the sky.

Drawing a Ship

Our spaceship will be really simple: just a humble triangle. Because we will be drawing it over and over again, we will put the drawing code into its own function. Go ahead and replace the entire program with this:

:html:

```
<canvas id="drawing" width="300" height="300"></canvas>
<script>
function main() {
    var canvas = document.getElementById('drawing'),
        ctx = canvas.getContext('2d');

    // Draw a spaceship with the bottom center at (x, y).
    function drawShip(x, y) {
        // No triangle function exists in the canvas, so we
        // use a path instead:
        ctx.fillStyle = 'blue';
        ctx.beginPath();
        ctx.moveTo(x, y-30);
        ctx.lineTo(x-5, y);
        ctx.lineTo(x+5, y);
        ctx.lineTo(x, y-30);
        ctx.fill();
    }

    // Draw the ship at bottom middle.
    drawShip(canvas.width/2, canvas.height);
}
main();
</script>
```

Note that we put the whole program into `main` to avoid polluting the global scope. But, because `ctx` is in the surrounding scope of `drawShip`, `drawShip` can use it.

If all went well, you will see a blue triangle on your canvas. You may want to add a border to convince yourself that it really is at the bottom.

We could, of course, make more elaborate drawings, but for now, it's a triangle whose bottom center is positioned at the given (x, y) location. At the bottom of our program, we call the function with the bottom middle of the canvas specified (x is `canvas.width / 2`, which is halfway across, and y is `canvas.height`, which is the bottom).

Launching a Ship

We've pointedly drawn a sharp-looking ship,

We're ready to send it off on a trip.



Figure 8.2: A little ship.

The basic idea is this: every time we draw a frame, we change the ship location by just a little bit. Since ships usually want to launch directly upward to avoid fiery destruction, that means we will subtract something from the `y` coordinate we give to our `drawShip` function each time it is called. It's time to call `setTimeout` with a closure that can change `y` and call `drawShip`. Let's call that closure `tick`. Before looking at the code below, see if you can figure out how to make this happen using the existing program.

What follows would go inside `main`, below the `drawShip` function. We have included all of it here, but will start omitting surrounding code as we go, to save space and focus on what's really critical.

:html:

```
<canvas id="drawing" width="300" height="300"></canvas>
<script>
function main() {
  var canvas = document.getElementById('drawing'),
      ctx = canvas.getContext('2d');

  // Draw a spaceship with the bottom center at (x, y).
  function drawShip(x, y) {
    // No triangle function exists in the canvas, so we
    // use a path instead:
    ctx.fillStyle = 'blue';
    ctx.beginPath();
    ctx.moveTo(x, y-30);
    ctx.lineTo(x-5, y);
```

```

    ctx.lineTo(x+5, y);
    ctx.lineTo(x, y-30);
    ctx.fill();
  }

  var y = canvas.height;

  function tick() {
    drawShip(canvas.width / 2, y);
    if (y >= 0) {
      setTimeout(tick, 100); // every 1/10 second
    }
    y -= 5;
  }

  tick(); // start drawing right away.
}
main();
</script>

```

See what we did? We defined a local variable `y` inside `main`, set to the bottom of the canvas. We also defined a function `tick` to draw the ship and update `y` for the next frame. Then we called it once to get things started (instead of calling `drawShip` directly, like we did earlier).

Here are the detailed steps that `tick` performs when called:

- Draw the ship at the current `y` coordinate, centered left to right,
- Check whether any part of the ship is still visible (the bottom of the ship is still on the canvas—remember that a `y` value of 0 is the top of the canvas),
- If so, instruct the browser to call `tick` again in 1/10 of a second (100 milliseconds), and
- Subtract 5 from `y` for next time.

Calling `tick` for the first time (at the bottom) sets this whole thing in motion. It doesn't look great, yet, but we'll fix it.

Note that `tick` makes use of more than one variable in its outer scope. It uses `y`, which we intended, but it also uses `canvas` to find the left-to-right center for the `x` coordinate. This works because `canvas` is defined in the surrounding scope. Finally, it also has `drawShip` in its surrounding scope, which it calls. Closures are pretty cool and are thankfully easier to use than to explain.

Clearing Between Frames

It looks like our ship is sort of smearing up the canvas instead of moving up it. That's because we forgot to erase the canvas between each frame. The computer is not at fault: it's just following instructions. To clear the canvas between frames, we add a single command to the

tick function: `ctx.clearRect`, and we give it the entire canvas as the rectangle to clear:

:javascript:

```
function tick() {  
  // Clear first, then draw.  
  ctx.clearRect(0, 0, canvas.width, canvas.height);  
  drawShip(canvas.width/2, y);  
  if (y >= 0) {  
    setTimeout(tick, 100); // 1/10th of a second.  
  }  
  y -= 5;  
}
```

That's simple enough: before drawing a frame we instruct the canvas to clear itself. For complicated animations sometimes it's useful to only clear part of the canvas, but we will always be sticking to the simple "every frame is a brand new drawing" model in this course. It works fine for our purposes.

Variable Shadowing

All surrounding-scope variables can be seen by all functions defined here, including variables defined below them. Remember that the definition of `drawShip` includes two variables:

`function drawShip(x, y)`. That means that `drawShip` defines its own `y`, while a different `y` also exists in its outer scope! What happens in that case?

If a variable is defined in the outer scope and a variable of the same name is defined locally, the local (innermost) one wins. This is called "shadowing"—the local variable blocks, eclipses, or shadows the outer scope variable, so the function `drawShip` only sees its own local `y` parameter, not the outer scope `y` that `tick` is using and changing. That outer `y` is forever invisible to `drawShip` because of the local `y` inside of it.

Shadowing is also caused by `var` declarations inside of a function. Be aware of it, sometimes it can surprise you if you are relying on your function closing over an outer variable with a common name.

Summary

There wasn't a large volume of new material here, but what was covered was conceptually very important. We talked about single-threaded programs and how that drives us to an event-driven model of computation in JavaScript, about how to work with that sort of thing using lexical closure, how to control scope using nested function definitions, how to animate something, and what is going on with variable shadowing. Put that way, it actually is a lot of new material. Make sure to do the homework and labs, and then you will be ready to tackle anything.

Also, play around! You have a basic animation here, ready to be tinkered with. There are all sorts of things you can imagine doing with the code here.

Exercises

Exercise 8-1: Practice with `setTimeout`

Solution on page 445

Write a function that draws a rectangle at a random location on a canvas every quarter second (250 milliseconds). The rectangle can be any color and the size of your choosing, so long as it is easy to tell that the program is working.

For bonus points, make the color random as well, so a different color can appear each time.

Exercise 8-2: Scope and Closure

Solution on page 446

Use the following code to answer the questions below:

:html:

```
<script>
var a = 10;

function F() {
  // code here
}

var b = 15;

function G() {
  c = "hi";

  function H() {
    var d = [];
    // code here
  }

  var e = "there";

  function I() {
    var f = {};
    // code here
  }
}
</script>
```

- List every variable and function name below that is in the global scope. Assume that all functions have run at least once (and remember what happens when you leave off

var).

- Can code inside of `H` access `f`? Why or why not?
- Can code inside of `I` call `H`? Why or why not?
- Can code inside of `H` call `F`? Why or why not?
- Can code inside of `H` access `e`? Why or why not?
- Can code inside of `F` call `H`? Why or why not?

Exercise 8-3: Animation

Solution on page 447

Write a program that draws a rectangle on the left side of the canvas and moves it to the right until it reaches the edge. The rectangle should be of size 10 by 10, and should move 5 pixels every 1/20th of a second (50 milliseconds). It can be anywhere along the `y` axis (centered top to bottom, on the bottom, on the top, or anywhere in between).

Exercise 8-4: Shadowing

Solution on page 448

What is displayed by the following program? Why?

:javascript:

```
var a = "hello";

function message(a) {
  console.log("message is:", a);
}

message("hi");
```

Exercise 8-5: Lab: Animate a Die Roll

Solution on page 449

For this lab you will animate the roll of a die. It won't be a physically accurate simulation; it will just display random dots a few times until it settles on a value. The dots will be rectangles, and the code to draw a single dot based on where it is on a 3x3 grid is given below to make it a bit easier on you.

:javascript:

```
// drawDot draws a 'dot' at the given row and column,
// assuming that size is set to the square side length
// of the die. Both row and col are zero-based, so
// 0, 0 is the top left corner and 2, 2 is the bottom right.
function drawDot(ctx, size, row, col) {
  var margin = size / 6,
      x = margin + (2*margin*col),
      y = margin + (2*margin*row);
  ctx.fillRect(x-margin/4, y-margin/4, margin/2, margin/2);
}
```

```

}

// dieDots defines where the dots are located for every
// possible value on a 6-sided die, by row and column.
var dieDots = {
  1: [{r: 1, c: 1}],
  2: [{r: 0, c: 0}, {r: 2, c: 2}],
  3: [{r: 0, c: 0}, {r: 1, c: 1}, {r: 2, c: 2}],
  4: [{r: 0, c: 0}, {r: 0, c: 2}, {r: 2, c: 0},
    {r: 2, c: 2}],
  5: [{r: 0, c: 0}, {r: 0, c: 2}, {r: 2, c: 0},
    {r: 2, c: 2}, {r: 1, c: 1}],
  6: [{r: 0, c: 0}, {r: 1, c: 0}, {r: 2, c: 0},
    {r: 0, c: 2}, {r: 1, c: 2}, {r: 2, c: 2}],
};

```

With these functions, you should be able to easily define a `drawDie` function that accepts a context, a size, and a value (the number of dots you want to show), like this:

:javascript:

```

function drawDie(ctx, size, value) {
  // Your code goes here, using drawDot and dieDots above.
}

```

And finally, with that, you can animate a die so that it picks a new random value every 200 milliseconds to display, and stops on the tenth one.

If you get this working quickly and just use random draws to create your values (that's a good idea—start with that!), you might notice that sometimes you get the same number twice in a row, and it makes the animation look like it paused. Add code to ensure that you never repeat the same number twice in a row.

Bonus: make sure you never go directly to a number on the *opposite side* of the die, either. You can easily check this because numbers on opposite sides always sum to 7.

Chapter 9

Smoother Animation Using Time and Animation Frames

We just made our first animation, and learned about closures and variable shadowing at the same time. That's a big step. Now it's time to refine our ability to animate and learn a little more about how to best make the browser work for us, along with getting a more accurate frame rate.

Animation Frames

It turns out that `setTimeout` is not really the best thing to use for animation because it's not terribly accurate and, depending on the browser, it has some unfortunate behavior with hidden browser tabs. If `setTimeout` is not good for animation, though, what is? The thing that people really use for animation is a more powerful and accurate function called `requestAnimationFrame`¹.

The browser has to draw everything you see on the page every so often, and if there is a lot of stuff moving around, it has to do that quite frequently. The human visual system is pretty awesome and can detect flashes at a very high frame rate, but movies are typically run at 24 frames per second (FPS), which is generally enough to give a convincing illusion of motion. Some high definition video runs at 50 to 60 FPS, and virtual reality gear typically runs at 90 FPS or higher to reduce motion sickness. The more rapid the motion (or especially the more quickly the rate of motion changes), the higher the frame rate should be.

Many computer games run at either 30 or 60 FPS because those rates work well with older and more modern monitors alike. It's a historical thing that has to do with CRT refresh rates and the need to render a frame in between refresh scans to avoid "tearing". It also happens that 60 FPS is a pretty solid rate for humans to perceive motion as smooth.

¹<https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame>

You might have noticed that your earlier spaceship was not animating very smoothly. That's because it was updating position and drawing at 10 FPS (once every tenth of a second), less than half the slowest movie rate. But `setTimeout` is not really accurate enough to draw a frame every 16.6 milliseconds (60 FPS), so it can't handle animation the way we would like.

The browser, however, knows when it is going to draw each of its own frames based on the refresh rate of the display it's using, and we can ask it to call our function at the same time using `requestAnimationFrame`. Let's use that instead of `setTimeout`. Here is the full listing of our earlier spaceship program, but this time using `requestAnimationFrame` (and simplified ship drawing).

:html:

```
<canvas id="drawing" width="300" height="300"></canvas>
<script>
var canvas = document.getElementById('drawing'),
    ctx = canvas.getContext('2d');

// Draw a spaceship with the bottom center at (x, y).
function drawShip(x, y) {
    ctx.beginPath();
    ctx.fillStyle = 'blue';
    ctx.moveTo(x, y-30);
    ctx.lineTo(x-5, y);
    ctx.lineTo(x+5, y);
    ctx.lineTo(x, y-30);
    ctx.fill();
}

var y = canvas.height;

function tick() {
    ctx.clearRect(0, 0, canvas.width, canvas.height);
    drawShip(canvas.width/2, y);
    if (y >= 0) {
        requestAnimationFrame(tick);
    }
    y -= 5;
}

tick(); // start drawing right away.
</script>
```

Note that we are dispensing with immediate functions and with `main`, just to keep things clean and clear for book format. Use one of them if you like.

Also note that because we aren't doing this whole thing inside of a function, we aren't really using closures, but global variables, which any function can see at any time. It's helpful that it looks and works the way we want it to in this context.

Well, that was easy. We just changed `setTimeout(tick, 100)` to `requestAnimationFrame(tick)`. But hold on—where did the frame rate go? We used to be able to tell it how often we wanted `tick` to be called, but now we can't! In fact, if you run this now, the ship will indeed be moving a lot faster. Give it a try!

Not only does the ship move faster, but if you hide this tab (by looking at a different tab or switching to a different program), then switch back to it, it will look like our animation paused while we were gone and resumed when we came back. That is one of the features of `requestAnimationFrame` that sometimes sets it apart from `setTimeout`: it is guaranteed to only run when things are visible. That can be useful on mobile devices, since it means our code isn't using up so much power when it isn't doing anyone any good visually.

A few browsers now pause things for `setTimeout` as well to help with power use for mobile devices, but this is not part of the spec and therefore can't be assumed to happen all the time.

If you run this multiple times, depending on your browser and the computer it is running on, you might notice some jerkiness in the motion, as well. If you do see that, it is probably due to one of a few factors:

1. You do not get precisely the same amount of time between frames, though it is usually very close,
2. The browser might not be able to quite manage the highest possible frame rate all the time, and
3. We are not requesting a new animation frame until after we have done a bunch of work; we might be missing the deadline to ask for the next one, so we might end up skipping frames.

Incidentally, games, particularly if they have a lot of complicated scenes that take time to render, sometimes run at 30 FPS and lock that in because our visual system is much more sensitive to *changes* in frame rate than it is to *consistently lower* frame rates. If a game is running at 60 FPS and has to drop down to 45 FPS for a particularly complicated scene, the illusion of motion is lost: our brains pick up on the change. Thus, many games take the position that it is better to have a completely consistent and smooth 30 FPS experience than to have the frame rate jumping around.

We can achieve this effect using `setTimeout`, which has sufficient accuracy if we emulate some of the timing features of `requestAnimationFrame` in most cases, but that's beyond the scope of what we're talking about here.

Smoothness and Time

There is something in common with all of the above commentary, and that is time. Our animation sped up because there is less time between frames than the 100 milliseconds we were using before, sometimes it may appear jerky because the interval is not exactly consistent or because we spent too much time drawing before requesting a new frame, and we might want to consider letting more time pass between drawings to have better control over our frame rate to keep things smooth. Thus, we need to start thinking much more carefully about time.

Since we are trying to animate something moving at a constant velocity, that ties position directly to time. If more time has passed between frames, more distance should be covered. Since animation frames are not always coming at us evenly, though they generally do come at about 60 FPS, we will want to know exactly how much time has passed between frames so that we can calculate positions to make it look like things are traveling at a constant velocity.

Thankfully, `requestAnimationFrame` sets things up so that `tick` gets a “current time” parameter. We have not made use of it before, but we will absolutely make use of it now. We will call it `t` because that is such a common name for a time variable:

:javascript:

```
function tick(t) {
  console.log(t);
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  drawShip(canvas.width / 2, y);
  if (y >= 0) {
    requestAnimationFrame(tick);
  }
  y -= 5;
}
requestAnimationFrame(tick);
```

Note that instead of calling `tick` directly at the bottom, now we are requesting an animation frame so that we can get an accurate time even on the first call. This will appear to be instantaneous (16 milliseconds is the maximum time we will wait, if we just barely missed a frame), so it doesn't cause us any trouble to do it this way and get the benefit of consistency.

Good. Now we have the time (in milliseconds) in our local `t` variable, provided to us by the browser's animation frame routines. It dumps an awful lot of stuff to the console. The times you see should be roughly 16.6 milliseconds apart, representing a frame rate of 60 FPS. Yours might actually run at a slightly different rate, though, depending on your setup.

Now that we are outputting to the console, you *might* notice things get smoother. Computers are like that, and browsers especially so. Sometimes changing what we output in one place also changes the timing of what happens in another. The true solution is to get our timing right, though, not to add statements that dump stuff to the console.

That sort of behavior, where a change in one part of a program causes (or hides) behavior in another part of a program, is often called a “heisenbug”, in reference to the Heisenberg Uncertainty Principle in quantum mechanics. Scientists, including computer scientists, are often endowed with a *quarky* sense of humor. You’re welcome.

If you don’t see any jerkiness to begin with, or any change when outputting to the console, congratulations: you lucked out! Don’t read too much into it, though; lottery tickets are still a terrible investment.

What is the actual time and how can we use it? The time given to our function is an accurate count of how many milliseconds have passed since the page was loaded. That can be useful, but when doing animations what we most often want is really the amount of time that passed *since the previous frame*. In other words, we really want the *difference* in times between frames. How can we get that? The simplest way is to keep track of the time we saw during the previous call to our function. A simple `tick` function that demonstrates this idea (but does no drawing) might look like this:

:javascript:

```
var lastTime = 0;

function tick(t) {
  var dt = t - lastTime;
  lastTime = t;
  console.log(dt);
  // Calculate position, draw stuff.
}
```

Now our `tick` function stores the time in `lastTime`. When it gets called again, that previous time is remembered and we can use it to calculate how long it has been since the last time `tick` was called. Pretty neat! It’s called `dt` because that means the difference in time, and it is a standard way of communicating that idea. In algebra we might call it Δt .

If you integrate this into the larger program, you should see that frames are approximately 16.7 milliseconds apart. That’s about 60 FPS, but will vary a bit. Some, however, might actually be much further apart than this. For example, in one of my runs, I had the following `dt` values shown next to each other:

:console:

```
18.754000004264526
16.74899998761248
```

These are clearly not the same. What may be more surprising is that this difference of 2 milliseconds can be detectable to the human visual system if it keeps happening—you can observe the jitter in the ship’s motion. That little change is enough to violate your built-in assumptions about how things move.

Even more jarring is output like this:

:console:

```

33.49799998977687
19.047000008868054
33.49799998977687
16.749000002164394

```

That run looked very jittery, as you might expect. You can see that sometimes we are missing entire frames because we wait too long to request the next one. The first fix for this kind of jitter is to move the call to `requestAnimationFrame` up to the top of the `tick` function, ensuring that we request the next frame before we do any serious work. This means we won't accidentally miss the boat for the next frame if our drawing happens to take a tiny bit too long. That fixes the biggest of the jitter problems quite nicely.

Some smaller jitter artifacts remain, however. In order to fix these, we will need to examine what we have done in terms of physics, at least a bit. If you take a look at our code, we're doing something very simple at every step: subtracting 5 from `y`. We don't use time anywhere explicitly, but we do *implicitly*, because each frame is assumed to be coming at us at a constant rate, and we therefore choose to move a constant distance every time. The frame rate is not really constant, though, as we saw with our little console test above. What can we do to reduce jitter, then? To answer that question, let's look at how speed, distance, and time are related to see if we can gain any insights.

Remember that with a “change in vertical position” Δy and a “change in time” Δt , we can compute vertical velocity v_y :

$$v_y = \frac{\Delta y}{\Delta t}$$

In other words, velocity is the change in position divided by the change in time (remember that Δ means “difference” or “change”). We are familiar with this concept because it comes up whenever we talk about vehicle speed, where we use terms like “miles per hour” or “kilometers per hour”. The “per” indicates division.

In the equation above, the velocity needs to be held constant. We can choose that, but we can only choose it once. Furthermore, we are given Δt when `requestAnimationFrame` runs, or at least, we can calculate it from `t`. That means we need to use a different Δy every time to make sure the equation is still valid: Δy is the only free variable; the others are either fixed (constant speed) or beyond our control (unpredictable frame delay). This calls for some algebra:

$$\begin{aligned}\frac{\Delta y}{\Delta t} &= v_y \\ \Delta y &= v_y \Delta t\end{aligned}$$

Well, that's handy. We can choose v_y to be some constant value, and we already have Δt (called `dt` in our function), which means we should be able to calculate the correct change in y by just multiplying those together! That means our update rule will change from something like `y -= 5` to `y += desiredVelocity * dt` (we'll make our desired velocity contain the minus sign that just disappeared). Why is that? Because Δy is the amount that we change y at each frame. So, we add Δy to y , noting that $\Delta y = v_y \Delta t$.

We don't really know what v_y should be, yet, but this is progress. At least we have some idea of how to approximate a constant speed, which keeps things moving smoothly.

How do we choose a velocity? To answer that question, let's first discuss what we were doing before. When we chose to move our ship by 5 pixels each frame, we were kind of implicitly choosing a velocity based on an assumption that `dt` was constant. If we are assuming 60 FPS, that means that the ship moves -5 pixels (because we subtract 5 from y every time), and it does that 60 times in one second. That is a v_y of -300 pixels per second. Does that seem about right? Is it taking the ship about a full second to travel from position 300 to position 0?

Yes! If your canvas is still 300 pixels high, it takes about a second to go from the bottom of the canvas to the top. Excellent. Always check that your answers make sense, and that can save you hours of debugging time. Here, we can be reasonably confident that this is a good velocity for us.

Note again that the velocity is negative. Velocities have direction, so negative in our case means "up" since the canvas is oriented with larger y values lower down. If the velocity were positive, that would mean "down" in this canvas configuration.

We now have everything we need to smooth this animation out. We have Δt (`dt`) in each animation frame, and we have v_y (we will call that `Y_VEL` and it will be equal to -300). To compute the amount we want to change y each time, we simply multiply the two numbers together, because according to our equation, $\Delta y = v_y \Delta t$. We will add that to y each time. Here is what the new `tick` function looks like when we make the change:

:javascript:

```
var y = canvas.height,
    lastTime = 0,
    Y_VEL = -300;

function tick(t) {
  if (y < 0) {
    return;
  }
  // Moved up to avoid jitter due to lost frames.
  requestAnimationFrame(tick);

  // Convert time elapsed from milliseconds to seconds,
  // since velocity is in pixels/second.
  var dt = (t - lastTime) / 1000;
```

```

    lastTime = t;

    ctx.clearRect(0, 0, canvas.width, canvas.height);
    drawShip(canvas.width / 2, y);

    // Add calculated change in position to y.
    y += Y_VEL * dt;
}

requestAnimationFrame(tick);

```

Do you see the changes? There are a handful of them:

- We moved `requestAnimationFrame` almost to the top of `tick` for the reasons mentioned earlier.
- We now bail out early if the ship is out of bounds (this keeps us from requesting a new animation frame *and* from drawing things out of bounds).
- `Y_VEL` contains our desired constant velocity of -300.
- We convert `dt` to seconds instead of milliseconds by dividing by 1000 (because $x \text{ ms} = x \text{ ms} \frac{1 \text{ s}}{1000 \text{ ms}} = \frac{x}{1000} \text{ s}$).
- The `y` position is changed not by subtracting 5, but by adding $v_y \Delta t$ (`Y_VEL * dt` in code): the (constant) velocity times the (computed) change in time.

That's it. Run it and see if it works! Hopefully the animation is more consistently smooth.

Note that our computation of `y` looks a bit different than what we had before. It used to be `y -= 5`, but then it transformed to `y += Y_VEL * dt`, which is the formula we figured out earlier. This change allows us to define direction by using the sign of the velocity instead of directly using addition or subtraction. Our earlier approach was the same as saying `y += -5`, and that makes it more clear that we are moving in the negative direction (again, this is “up” on the canvas).

It is instructive to output the value of `Y_VEL * dt` to the console inside of `tick`. You might, if you were experiencing jitter before, notice that this change in `y` is not always the same, but your animation is still smooth; it tracks the differences in frame rate to make it that way. It is compensating for unwanted changes in time.

Name Your Constants

Why not just use `-300` where we need it for the velocity instead of creating a new variable to hold it? Because that would be a magic number. It is one thing to divide by 1000 in a place where we know we want to convert from milliseconds to seconds: there is only one way to do that. It is quite another thing to have a magic number like -300 sitting around in our code: that was a choice, and we may want to change it without remembering all of the places we have used it.

Naming constants like this is a very important practice in computer programming, so we will do it even in small examples where it doesn't seem to matter much. It's a good habit to get into, and it doesn't take much time to do it right. Finding bugs because you have done it wrong, however, can take a great deal of time. Also note that constants (even when they are actually variables) are often named using `ALL_CAPS_WITH_UNDERSCORES` by common convention. It doesn't mean anything special to the interpreter; it is meant to be a signal to humans.

Note that later versions of JavaScript (available pretty much anywhere you might care about) also understand `let` and `const` for setting variables. These are strictly better than `var` in several ways, but `var` still works and is simpler to talk about in an introductory setting.

The `let` keyword works a lot like `var`, but has better behavior in certain circumstances, particularly in loops containing closures.

The `const` keyword also declares a variable, but does not allow it to be reassigned.

Again, we don't use them here, but it is good to know what they do, and you will want to learn to use them instead of `var` as you progress in your programming career.

We have now dealt with smooth motion based on constant velocity calculations. This works fine with a changing velocity, too. The key is to determine motion not based on a random number of pixels we want to move, but based on what we actually want to see happen. In the ship's case above, what we want is a constant velocity, because we want smooth flying. If we were working within the context of gravity, we would want a constant acceleration, which we would use to compute velocity, which we would then use to calculate position. All kinds of motion are possible using this idea, if you start from what you really want to see.

Another Wrinkle In Time

There is one more important oddity that we need to address with `requestAnimationFrame`, and you might have run into it already; it is colloquially referred to as the “teleportation” or “time warp” problem.

Remember how we talked earlier about the animation sort of “pausing” if we navigate away from the page? That happens because the interpreter stops calling our `onTick` function if the tab or window is not showing. If the window isn't showing, the browser isn't drawing it, and we can't hook into that to draw what we want.

When you are writing a game, this becomes important, especially when calculating the positions of things based on how much time passes between frames. Remember, the time given to our `tick` function is the wall clock time since the page was loaded. Consider this scenario for a moment:

- Load the page, start playing,
- Move away (maybe a calendar alert popped up), and
- Come back a few minutes later to continue playing.

What happens to the time in this scenario? While you are playing, the frames are coming about once every 16.7 milliseconds, which is good. We compute the position of your character or rocket or whatever based on that time, and the motion is relatively smooth. Then you switch to a new browser tab for a few minutes, and your `tick` function is not called at all during that time, so it cannot update `lastTime`. When you come back, the `tick` function is called again, but the difference between the previous frame and the current frame will be several minutes instead of a few milliseconds. This is often referred to as the “time warp” problem with animation frames, because it makes everything look like it fast-forwarded in time when you come back to the window. Some call it “teleportation”, since it can make it look like your space ship suddenly disappears from its last location and reappears far away from it.

Our animation goes by too quickly for us to easily observe this problem, but you can try changing the velocity to something like -50 instead of -300, load the page, change to another tab for a few seconds, then come back and see if the ship suddenly jumps. You should be able to actually see it jump, since navigating away stopped the frames, and coming back will give the first frame a very large time.

To fix this problem, we first have to decide what we want to have happen when we come back to a game after a long absence. Ideally, it would resume basically where we left off. In terms of our equation, that means we would like to pretend Δt is still small, as though we had a normal time between frames. How can we do that?

One possibility that is easy and relatively common is this: if Δt is much bigger than we expect, just force it to be small and go on with life. For example, we might say “if the change in time is bigger than four times what we normally see (we expect 60 FPS, but could get as low as 24 FPS depending on the setup, and this allows for that), just pretend it was what we normally expect to see.” In code, the skeleton of that idea looks something like this:

:javascript:

```
var EXPECTED_RATE = 1 / 60, // seconds
    lastTime = 0;

function tick(t) {
    requestAnimationFrame(tick);

    var dt = (lastTime - t) / 1000;
    // No teleporting!
    if (dt > 4 * EXPECTED_RATE) {
        dt = EXPECTED_RATE;
    }
    // Do other tick stuff.
}
```

After adding code like the above, the bizarre-looking jump won't happen anymore. Instead, the ship will just continue from where it was when you left. That is much better.

Because not all systems are running at 60 FPS, this isn't really the best way to go about deciding what is "expected". There are various (comparatively advanced) techniques that you can look up online for obtaining the actual frame rate, but what we have here should be safe enough for our purposes. Because we are letting `requestAnimationFrame` tell us the actual time elapsed, and we are using desired velocity against that time, the speed of the ship should be the same in wall clock time no matter what system you are on.

We have come a long way. Let's get the big picture again by looking at all of the code in one place:

:html:

```
<canvas id="drawing" width="300" height="300"></canvas>
<script>
var canvas = document.getElementById('drawing'),
    ctx = canvas.getContext('2d');

// Draw a spaceship with the bottom center at (x, y).
function drawShip(x, y) {
    ctx.fillStyle = 'blue';
    ctx.beginPath();
    ctx.moveTo(x, y-30);
    ctx.lineTo(x-5, y);
    ctx.lineTo(x+5, y);
    ctx.lineTo(x, y-30);
    ctx.fill();
}

var y = canvas.height,
    lastTime = 0,
    Y_VEL = -300,
    EXPECTED_RATE = 1/60;

function tick(t) {
    if (y < 0) {
        return;
    }
    requestAnimationFrame(tick);

    var dt = (t - lastTime) / 1000;
    if (dt > 4 * EXPECTED_RATE) {
        dt = EXPECTED_RATE;
    }
    lastTime = t;

    ctx.clearRect(0, 0, canvas.width, canvas.height);
    drawShip(canvas.width/2, y);
}
```

```

    y += Y_VEL * dt;
}

requestAnimationFrame(tick);
</script>

```

The above is an HTML page that animates a space ship at a consistent velocity, that pauses when we leave and resumes where we left off when we return. Not bad for so little code! These are the fundamental building blocks for animating anything you like, and now you have all you need to get it done using JavaScript in the browser.

Exercises

Exercise 9-1: Trivia for requestAnimationFrame

Solution on page 452

- How often is the function given to `requestAnimationFrame` called if it sets up a new call every time?
- When is the function not called at all?
- What parameters are available when the function is called?
- Where in that function should the call to `requestAnimationFrame` reside (for animation), and why?

Exercise 9-2: Lab: Constant Acceleration

Solution on page 453

For this lab, change the ship's trajectory to act more like it was thrown upward in the presence of gravity. This means that it is *always* experiencing an acceleration downward, and it only has an *initial* velocity, not a *constant* velocity.

Requirements:

- Use an acceleration of 150.
- Use an initial velocity of -300.
- Stop the animation when the ship gets back to the bottom of the canvas.

Hints:

Acceleration is a *change in velocity*. Thus, if you have an acceleration of 5 pixels per second per second, that means that your velocity will change by 5 each second. If you have less than a second to work with (as is the case with our frame rate), then you can approximate reality by multiplying it by the amount of time that has passed.

Thus, we might calculate the current velocity like this: `yVel += ACCEL * dt`. With that, we can compute the new position just like we did before.

Chapter 10

Click and Key Events

Animations are fun, but not exactly interactive. We have essentially created a short movie. You can consume it by watching it, and that's all. There's no way to even pause it other than looking at another browser tab for a while. Let's fix that. Let's make it possible to pause and restart our animation without looking away. For that, we will need some new events.

We have already been using events, since we are now familiar with `setTimeout` and `requestAnimationFrame`. These are functions that allow us to indicate that we want something to happen “in the event” that some time goes by. Now we are going to learn about what to do in the event that a key is pressed or a button is pushed.

Are We There Yet?

Remember the way we constructed our `tick` function for animation? Its essence is something like the following code. Note that this is a sort of template we have been following, not the actual code we have been using, and we will come back to this template later on. Meanwhile, look carefully at this code, and find `done`, `draw`, and `move`:

:javascript:

```
// How long we expect between frames.
var EXPECTED_DT = 1 / 60;

// Used in the calculation of dt below.
var lastTime = 0;

function tick(t) {
  // Check whether we should quit ticking.
  if (done()) {
    return;
  }
}
```

```

// Set up for the next frame after this.
requestAnimationFrame(tick);

// Calculate how much time has passed.
var dt = (t - lastTime) / 1000;
lastTime = t;

// Fix the time warp problem.
if (dt > 4 * EXPECTED_DT) {
    dt = EXPECTED_DT;
}

// Draw stuff, and calculate next positions.
draw();
move(dt);
}

```

In this code, every time we get an animation frame, we immediately queue up a request for the next one (so we don't miss out from slow calculations), but only if `done()` returns false. If `done()` returns true, then we bail out of the function altogether and don't try to set it up again.

If we're still going, then we continue on to calculate `dt`, save the previous time for the next time `dt` needs to be calculated, and adjust it in case we left the screen and came back to it later.

Finally, we draw stuff where it is and figure out where it should be next time. We will come back to the fact that most of this function has nothing to do with actually moving and drawing what we are animating, but for now let's take a closer look at what `done()` might do. Note that it doesn't have to be a function—it can be a variable or some other expression, as well.

In our previous animation, we stopped when a spaceship had left the canvas, but there is no reason we can't have more logic in there than that. Let's remember this while we discuss events, then we will come back and see how we can use `done()` to pause and restart our animation while still keeping track of whether the ship has left the canvas or not.

Adding a Button

If we are going to start and stop our animation, we will need some way to communicate with it. Let's make a button we can push! HTML is great for this sort of thing because it has so many facilities like buttons already built in. To add a button in HTML, we use the `<button>` tag, like this:

:html:

```
<button id="pause-button">Pause</button>
```

The ID is familiar; our canvas has one of those, and we use it to ask the document for the

element so we can do useful things with it in JavaScript. It can be anything, but this seems like a reasonable choice.

The text on the button is “Pause”, and you can see where that goes: between the opening and closing tags. If you are following along, it’s time to create a new program with a button and some script tags in it, and it will soon be time to detect when the button is pressed. Our program will look like this, at first, just so we can test out how buttons work. This is a good idea in general when learning a new concept: create the simplest possible program that exercises the idea, then expand it after you understand it.

:html:

```
<button id="pause-button">Pause</button>
<script>
  // Magic happens here.
</script>
```

The first thing to know is that any element in an HTML page can create events. You can click on them or tap them (if you have a touch device), you can hover over them with a pointer (mouse or pad), focus them with the keyboard, and numerous other things. When you do, those elements can be set up to notify your code that something happened.

Just like we tell the browser to call our function after a period of time using `setTimeout`, or on the next animation frame using `requestAnimationFrame`, we can tell it we want to respond to an event by calling `addEventListener` on the element we are interested in, like this:

:html:

```
<button id="pause-button">Pause</button>
<script>
var btn = document.getElementById('pause-button');
function whenClicked(event) {
  console.log('button clicked!');
  if (event.shiftKey) {
    console.log('-> with the shift key!');
  }
}
btn.addEventListener('click', whenClicked);
</script>
```

Again, if you don’t have `console` access for whatever reason, you should be able to use `alert` instead, or the fake console introduced early in the course.

When we call `addEventListener` on any element, we tell it what kind of event we are interested in (`'click'`) and what to do when it happens (call the function `whenClicked`). There are many such events, and you can read about them in the online documentation¹.

¹<https://developer.mozilla.org/en-US/docs/Web/Events>

Some elements accept different events than others, but most visual elements can accept mouse, touch, and keyboard events.

Event listeners are functions that we register to be called when events happen, just like we did with `setTimeout`. In this case, `whenClicked` registered as an event listener. That means that it will be given a special `Event`² object as its first argument that we can use to find out more information about what happened. For example, we can find out whether the shift key was held down during the click event.

If you run the above code with the console open, you will see text appear whenever you click the button, and extra text will be displayed if you hold the shift key while clicking. This is one of those things that you really need to try to see how it works. Go ahead; this text isn't going anywhere.

If you are feeling a little overwhelmed with new concepts, take a look at previous chapters and see if they feel more comfortable to you. Those used to be new and strange, too, but now you understand them better! That will happen with what you are learning now, as well.

Events and Anonymous Closures

In our most recent event listener code, we created a function and then registered it. Just for fun, let's see what happens if we use an anonymous function instead! This code does the same thing as the previous code:

:html:

```
<button id="pause-button">Pause</button>
<script>
var btn = document.getElementById('pause-button');
btn.addEventListener('click', function(event) {
  console.log('button clicked!');
  if (event.shiftKey) {
    console.log('-> with the shift key!');
  }
});
</script>
```

Quite often we define event-handling functions and only use them in one place, so they don't really need names—they can just be passed in directly to `addEventListener` as anonymous functions; we are creating a function and passing it to `addEventListener` in one step. Make sure you know how to read that code before moving on, because this technique is about to get a lot more common.

²<https://developer.mozilla.org/en-US/docs/Web/API/Event>

One of the biggest missing features in Java (not JavaScript!) up until Java 8 was *lambdas*. These are basically anonymous functions, and the lack of them was almost entirely responsible for every joke ever made about a “FactoryFactory”. Anonymous functions may seem like a mere convenience, but they can greatly improve clarity by removing the need to name every single little thing, and JavaScript has had them from the beginning. Most mainstream languages have a good anonymous function facility these days. Languages that don’t tend to feel pretty clunky after a while (Python included, as its `lambda` facility is so limited as to be almost useless).

Note that you are not required to use anonymous functions, but you will need to be able to read them, and it is more than a little helpful to not have to come up with names for all of them. If you prefer naming all of your event handlers, that’s fine; by all means continue to do so. We will still do a little of both in this curriculum, and that should give you practice with how things are done in the real world of programming.

Actually Pausing Stuff

In order to effectively pause our animation, we need to do some surgery on our previous spaceship program. Here it is, with the button added to the file, plus a line break tag (`
`) to make the button appear on the next line instead of right next to the canvas. We will make multiple changes to get a completely working pause button, but it will be a pain to show the entire program at every step, so let’s make sure we have a common starting point. This initial code has a pause button, but it doesn’t do anything yet.

Here is the spaceship program from before, with the button specified right below the canvas:

:html:

```
<canvas id="drawing" width="300" height="300"></canvas>
<br>
<button id="pause-button">Pause</button>
<script>
var canvas = document.getElementById('drawing'),
    ctx = canvas.getContext('2d');

// Draw a spaceship with the bottom center at (x, y).
function drawShip(x, y) {
    ctx.fillStyle = 'blue';
    ctx.beginPath();
    ctx.lineTo(x-5, y);
    ctx.lineTo(x, y-30);
    ctx.lineTo(x+5, y);
    ctx.fill();
}

var y = canvas.height,
    lastTime = 0,
```

```

    Y_VEL = -50,
    EXPECTED_RATE = 1/60;

function tick(t) {
    if (y < 0) {
        return;
    }
    requestAnimationFrame(tick);

    var dt = (t - lastTime) / 1000;
    if (dt > 4 * EXPECTED_RATE) {
        dt = EXPECTED_RATE;
    }
    lastTime = t;

    ctx.clearRect(0, 0, canvas.width, canvas.height);
    drawShip(canvas.width/2, y);
    y += Y_VEL * dt;
}

// BUTTON EVENT CODE GOES HERE.

requestAnimationFrame(tick);
</script>

```

If we run this, we will see a button and the ship will start moving as before. The button, however, doesn't do anything. That's because we don't have any code that knows how to respond to a click. Our first order of business is thus to hook up a click listener for the button. It won't do anything but log to the console right now, but let's at least get it hooked up. We can do that by adding the following at the bottom of our script, replacing the `BUTTON EVENT CODE GOES HERE` comment:

:javascript:

```

var btn = document.getElementById('pause-button');
btn.addEventListener('click', function(event) {
    console.log('click!');
});

```

Great! We can now see things happen in the console when we click the button. The question is, what should we really do when this button is clicked? Right now, when we load our page, the ship takes off right away and keeps on going until it leaves the canvas, which happens when `y < 0`. That behavior is managed with this code snippet:

:javascript:

```

if (y < 0) {
    return;
}
requestAnimationFrame(tick);
// Do other stuff.

```

If we want to pause from our click listener, we need to have an additional way of telling it to stop creating frame requests, something other than “you went off the screen”. What if we had a `running` variable and the logic looked more like this?

:javascript:

```
if (y < 0 || !running) {  
    return;  
}  
requestAnimationFrame(tick);
```

Remember the boolean OR operator `||`? The logic above can be interpreted to say, “If the ship is off the screen or the animation is not running, then quit immediately.” Thus, if we are either off the screen or something sets `running` to be `false`, we won’t request another animation frame. Since that means we won’t call `tick` again, no new position will be calculated and nothing new will be drawn: everything will stop.

That’s all well and good, but now we need a `running` variable that is initially `true` so that our animation doesn’t stop before it starts. We’ll add that right before the `tick` function, and then use it as described above:

:javascript:

```
var running = true;  
  
function tick(t) {  
    if (y < 0 || !running) {  
        return;  
    }  
    requestAnimationFrame(tick);  
    // ... everything else as before ...  
}
```

Then we can change the code in our button click listener to change `running` to `false`:

:javascript:

```
btn.addEventListener('click', function(event) {  
    running = false;  
});
```

Since `running` is in the event listener function’s outer scope, `running` is part of its closure. When that function is called, it can change `running` to `false`. But `running` is also part of the closure of `tick`, so it sees the change made by the click listener and stops requesting animation frames accordingly: everything pauses.

Note that this really only works reliably because JavaScript is single-threaded. That makes it safe for us to alter variables from what is otherwise an *asynchronous* process:

the button click might come at the same time that `running` is being accessed. This is called a “race condition” and is something to be aware of when graduating to languages with truly concurrent execution. Again, JavaScript is not one of those (in this context, anyway), so we can get away with being a bit cavalier and just relying on intuition in this course. Just be aware that you would need to be *far* more careful about this in many other mainstream languages. It would require some kind of guard on the variable, like a mutex.

When you run this now, it should pause you click the button. What happens when you click it again?

The answer is “nothing”. Nothing happens if you click it again. Because the code has already changed `running` to `false`, setting it to `false` again has no effect. At least it paused our program the first time, though! That part worked.

What if we wanted to restart it? Could we just change `running` back to `true`?

What do you think? What will that do?

The fact is that once things are paused, changing `running` to `true` won’t restart things by itself. It will set things up so that things *can continue* once started, but it won’t actually get them going again. The reason for this is pretty simple: changing `running` to `true` doesn’t request any new animation frames, nor does it call `tick` directly, so `tick` never runs again. Thus, when resuming the animation, we also need to request a new animation frame to kick things off, like this:

:javascript:

```
btn.addEventListener('click', function(event) {  
  if (running) {  
    running = false;  
  } else {  
    running = true;  
    requestAnimationFrame(tick);  
  }  
});
```

Does that work better? It looks like it does. If things are already running, we stop them by setting `running` to `false` and letting nature take its course the next time `tick` is called. If things are not running, we set it to `true` and instruct the browser to call `tick` again when ready. That works.

There is another race condition here, though, and this time it actually matters. What happens if your animation is running, and you manage to click the pause button twice in a row between frames? You would have to be pretty quick, but it’s quite possible to do this. If you do, the currently-running animation never realizes that it should have exited, and a brand new animation is started in parallel with it. Give it some thought and see if you can see the issue.

Race conditions can be super subtle, and are often triggered by user input or other outside events. We'll deal with this one in particular later on.

Readability

Just because we can, let's look at a slightly more succinct way of writing the same thing:

:javascript:

```
btn.addEventListener('click', function(event) {  
  running = !running;  
  if (running) {  
    requestAnimationFrame(tick);  
  }  
});
```

In this code, we set `running = !running`, and thus re-encounter another old boolean friend: the NOT operator `!`. If `running` is `false`, then `!running` is `true`, and vice versa. Thus, we change `running` to be whatever is the opposite of its current state. We call this “toggling” the variable. Then, after toggling `running`, we check to see if we should be animating. If we should, we kick off the animation by requesting a new frame just like we do when the page is first loaded, and it will take things from there.

Neither version of these is really strictly better than the other. Quite often these things are simply a matter of taste, and in this way programming is much like art: you are communicating with both computers and humans when you write code, and there is such a thing as an elegant program, or a clear program, or a weird and discomfiting program. You might choose the toggling version because a toggle is the simplest thing to understand for what you need to do. You might, on the other hand, choose the if/else version if there is more work to be done for each case.

The question I personally like to ask is this: is this the clearest expression of this idea? Subject to correctness and reasonable efficiency concerns, that is often the most important question you can ask when designing your programs, because they will be read by people a lot more than they are ever written. Even the act of writing a program requires reading it first to get your head right so you can modify the right parts in the right way. Always try to make your code easy on the eyes as well as on the silicon.

A good rule of thumb for what “clear expression” means is this: the expression that will require the least effort for a newcomer to understand. We call that “readable code”, and it's important not just because someday you might be doing this on a team, but because someday you might want to dust off your own code and do something new with it, and you will have long forgotten what you were thinking about when you wrote it. More often than not, your future self is the newcomer to your current self's code. Be kind to your future self.

With all of that philosophy behind us, let's get back to making stuff happen. Did our pause/resume change work? Give it a shot and find out!

Changing Button Text

It's great to have a working pause/resume button, but other than the motion of the spaceship, we have no indication of the state of our animation. Let's fix that by changing the appearance of the button when it is clicked:

:javascript:

```
btn.addEventListener('click', function(event) {
  running = !running;
  if (running) {
    requestAnimationFrame(tick);
    btn.innerText = 'Pause';
  } else {
    btn.innerText = 'Resume';
  }
});
```

The `innerText` property of buttons (and other HTML controls) allows us to change their appearance on the fly. If we're running, we know we can pause, so we make sure the button says that, and vice versa.

More Readability

Note that we are both toggling `running` and using an `if/else` statement here. That seems to be the most readable approach because we change the button state based on what we want, not on what we once had. If we were to remove the toggle and fold it into the `if/else` statement, we would have to invert the logic. Contrast the toggle version with this:

:javascript:

```
btn.addEventListener('click', function(event) {
  if (running) {
    running = false;
    btn.innerText = 'Resume';
  } else {
    running = true;
    btn.innerText = 'Pause';
    requestAnimationFrame(tick);
  }
});
```

Now it reads like “if running, stop running and set text to indicate that we can get started again”. With the toggle approach, it reads “if currently running, set text to indicate that we can pause”.

The principle here is to change the state once (meaning, we just changed the master `running` state on which everything depends), and then compute all of the new things for that new state. It can make code much easier to reason about, and this principle is part of the foundation of many popular web frameworks: change state, then figure out the world should

look like. Our animation works that way, too, when you think of it. We advance `y`, and later we draw things where they are.

And now it's time for an important exercise in removing race conditions, since there is still one hiding in here if you click the pause button really quickly twice in a row.

What can happen? Let's start with a running animation and look at what happens with one call to `tick`, two rapid clicks of the pause button, and the next call to `tick`:

- Run `tick`, calling `requestAnimationFrame`.
- Click button: set `running` to `false`.
- Click button: set `running` to `true`, call `requestAnimationFrame`.

Uh oh. See how we now have *two* outstanding requests for an animation frame? We assumed that setting `running` to `false` would stop the current animation, but we accidentally set things up so that clicking the pause button can race with the `tick` function checking on the state of `running`. In this scenario, the button click won the race, and `tick` was never told to stop!

Race conditions pop up all the time when you have unpredictable events that change the state of a variable that is needed by something on a separate schedule. You have to be pretty alert for them in things like games because user input is precisely that kind of unpredictable event that can cause a race condition.

To fix things like that, you can guard them with some kind of “mutual exclusion” lock (known as a “mutex”) in some languages, but JavaScript doesn't really have one of those because it is single-threaded.

So why can it happen? In this case it's because we *actively* start things, but we only *passively* monitor a value (on a schedule) to stop them! It's polling things on a schedule that creates this race condition, since it allows for a window of time where `tick` is blind to what is happening with the `running` variable.

To fix this, we have to actively stop the animation by removing the pending animation frame. That way there is no waiting for `tick` to discover that it's finished: it simply never runs again. To accomplish this, we use `cancelAnimationFrame`, which accepts an animation frame as a parameter, and then cancels it.

To get an animation frame, we will need to start using the return value of `requestAnimationFrame`, which we have been ignoring up until now. Let's store the result of that call in the `frame` variable every time we call it:

:javascript:

```
var frame = null;

function tick(t) {
  if (y < 0) {
    frame = null;
```

```

    return;
}

frame = requestAnimationFrame(tick);
// ... everything else as before ...
}

frame = requestAnimationFrame(tick);

var btn = document.getElementById('pause-button');
btn.addEventListener('click', function(event) {
    if (frame) {
        cancelAnimationFrame(frame);
        frame = null;
        btn.innerText = 'Resume';
    } else {
        btn.innerText = 'Pause';
        frame = requestAnimationFrame(tick);
    }
});

```

This resolves the race condition. Now, if the button is pressed twice in quick succession while the program is running, it cancels the animation frame that is currently waiting, and then requests another one.

Recall that the test `if (frame)` tests for truthiness. Remember, `null` is falsy, and a non-null object like that returned from `requestAnimationFrame` is truthy. If there's a real frame, we're animating, and if there's not, we're not. That's what the statement tests. Thus, `running` is like `frame`, so we take advantage of that fact here (and swap the `if` and `else` bodies to account for that).

If we really want a value like what `running` used to give us, a common shortcut for taking a truthy or falsy thing and turning it into a real true/false value is to simply invert it twice, like this:

:javascript:

```
running = !!frame
```

That's a common idiom for saying "make a truthy thing true, or a falsy thing false".

Canvas Clicks

Our little button for pausing is pretty useful! What if we set things up so that we could get the same effect by clicking on the canvas, or pressing the space bar? Let's do that. What we will first want to do is actually give our handler function a name so that multiple events can trigger it in exactly the same way. I know, I know, we went to all that trouble to use an anonymous

function, and for what? For science, that's what. Now we will get rid of it for science, too. Science can be awfully demanding.

Let's transform our button event listener into a separate `togglePaused` function and a single line of `addEventListener`. Then we can add some more events:

:javascript:

```
function togglePaused(event) {
  if (frame) {
    cancelAnimationFrame(frame);
    frame = null;
    btn.innerText = 'Resume';
  } else {
    btn.innerText = 'Pause';
    frame = requestAnimationFrame(tick);
  }
}

btn.addEventListener('click', togglePaused);
```

That was easy. Now how do we make the same thing happen if we click anywhere on the canvas? Simple, we add a click event to the canvas:

:javascript:

```
canvas.addEventListener('click', togglePaused);
```

That's it! Not only does it work to pause and resume our animation by clicking on the canvas, but it also changes the button state so that it reflects what's going on. Very nice.

Even More Readability

What we have just done is an instance of the Don't Repeat Yourself (DRY) Principle, and it is about much more than just saving keystrokes. When we have need of doing the same thing from multiple different contexts, we really should extract a function and call it from all of those places instead of copying and pasting the relevant code. That obviously saves us typing (or pasting) in the long run, but more importantly, it saves us bugs. If there is some subtlety in our pause toggle, and there definitely is (like needing to request an animation frame when we start running again, and needing to get the button text changed no matter how the pause happens), then we want to figure out that subtlety exactly one time and capture that wisdom in a function. Then, no matter where we need pause/resume functionality, it will all work the same way.

That's not all. If we decide to do other things when pausing or resuming the animation, we can make those changes in just one place and have all use cases benefit simultaneously. That can be very important when programming. Many bugs, including truly horrible security bugs in popular web browsers, things that cause people to lose credit card information and have their identities stolen, have been introduced because buggy code was copied to multiple places, but only fixed in one.

Key Events

Let's do one more thing. Let's make it possible to pause and start our animation using the keyboard, not just by clicking a button or the canvas. Once we have this concept, we will be well on our way to creating an interactive game!

Keyboard events are kind of tricky because the element that receives the event depends on what is currently “focused”. This gets into all kinds of advanced things like how events propagate through the document object model (i.e., they start at the top of the hierarchy, move down to the lowest level of detail, then propagate back up). Instead of getting into all of that here, we simplify things as much as we can by watching for all keyboard events on the top-level document object. There will likely come a time when that is overly eager for your programs, but for now it is going to be perfect for our needs.

There are three fundamental key events of interest here:

- “keydown”
- “keyup”
- “keypress”

The short version of how to use “keypress” is “just don't”³. It pretty much never does what you want. Ignoring that one leaves us with just two events to choose from.

Most of us don't really think about when certain things happen during our keystrokes, but it's an important question to answer. Do we want to respond to the action of pressing a key down, or to the action of letting it up afterward? When you type characters into an editor, the character appears immediately when you press the key down. You can tell by pressing and holding a character key: the character immediately appears, then it starts repeating while you hold it (which is one reason you don't want to deal with “keypress”, since it triggers for all of those). Thus, to have the most natural behavior, we will use “keydown” instead of “keyup”.

There are some common situations in which we will want to use “keyup”, and we will get to those in later chapters. Stay tuned!

The next question is how to tell which key was pressed. If we listen for “keydown”, we are listening for literally any key on the keyboard being pressed down, not just the space bar, and there is no way to tell the document “only call me when space is pressed”. Instead, we will listen for all keys, but do nothing when the key pressed is not the one we care about. For that, we will ask the event for the key that was pressed, using `event.key`:

:javascript:

```
document.addEventListener('keydown', function(event) {  
  if (event.key === ' ') { // quote-space-quote  
    togglePaused(event);  
  }  
});
```

³<https://developer.mozilla.org/en-US/docs/Web/Events/keypress>

```
    }  
  });
```

Why did we use an anonymous function here instead of just passing in `togglePaused` like we did with the click handlers? Because as mentioned earlier, the “keydown” event is triggered for every key on the keyboard, and we just wanted to respond to the space key. So, we used a small anonymous function to only call `togglePaused` when we actually have a space key event. If we just handed `togglePaused` to the `addEventListener` function here, we would pause no matter which key was pressed.

Preventing Default Behavior

There is one more tidbit to share about events that could be useful, particularly when trapping the space key, and that is `event.preventDefault()`.

The space key usually scrolls the page down in a browser. Our page doesn’t have any scrolling to do, so we don’t need it here, but sometimes this is really a problem. If you want the space key to pause your animation, you probably don’t want it to also scroll the page. In that case you signal to the browser that you’ve handled this event and don’t want any other code to handle it for you. That’s what `event.preventDefault()` does: it stops the default behavior for the event you just handled.

We won’t generally be using it in this course, but it was worth mentioning once. And, if you find that weird scrolling happens when you use the space bar, you know what to do!

Summary and Full Listing

In this chapter we spent some time learning about events and how to use them to pause an animation. That was pretty useful! We also spent a bit of time reorganizing our code to allow more than one thing to trigger some behavior. Our pause/resume handler started out as a named function, then became anonymous, then went back to being named because it started being used in more than one place. The process of moving things around like that, particularly of extracting out common functions, is called **refactoring**; we are factoring our code into different-shaped pieces, and sometimes doing so repeatedly.

The process of refactoring is a natural and important part of any programming endeavor. You will never get your ideas exactly right the first time, because there is always something to change after the code is “done”: bugs to fix, functionality to add, or readability to improve. That’s okay. Code evolves over time. It’s good to become comfortable with that.

Before we quit this chapter, here is our full animation code, complete with pausing using a button, the keyboard, or a click on the canvas:

:html:

```

<canvas id="drawing" width="300" height="300"></canvas>
<br>
<button id="pause-button">Pause</button>
<script>
var canvas = document.getElementById('drawing'),
    ctx = canvas.getContext('2d');

// Draw a spaceship with the bottom center at (x, y).
function drawShip(x, y) {
    ctx.fillStyle = 'blue';
    ctx.beginPath();
    ctx.lineTo(x-5, y);
    ctx.lineTo(x, y-30);
    ctx.lineTo(x+5, y);
    ctx.fill();
}

var y = canvas.height,
    lastTime = 0,
    frame = null,
    Y_VEL = -50,
    EXPECTED_RATE = 1/60;

function tick(t) {
    if (y < 0) {
        frame = null; // no longer running
        return;
    }
    frame = requestAnimationFrame(tick);

    var dt = (t - lastTime) / 1000;
    if (dt > 4 * EXPECTED_RATE) {
        dt = EXPECTED_RATE;
    }
    lastTime = t;

    ctx.clearRect(0, 0, canvas.width, canvas.height);
    drawShip(canvas.width/2, y);
    y += Y_VEL * dt;
}

frame = requestAnimationFrame(tick);

var btn = document.getElementById('pause-button');
function togglePaused(event) {
    if (frame) {
        cancelAnimationFrame(frame);
        frame = null;
        btn.innerText = 'Resume';
    } else {

```

```

        btn.innerText = 'Pause';
        frame = requestAnimationFrame(tick);
    }
}

btn.addEventListener('click', togglePaused);
canvas.addEventListener('click', togglePaused);
document.addEventListener('keydown', function(event) {
    if (event.key === ' ') {
        togglePaused();
    }
});
</script>

```

As you can see, things are starting to get just a bit unwieldy for a book format. Our code is not terribly long just yet, but it is starting to be. That means we are ripe for a refactoring exercise. Stay tuned!

Exercises

Exercise 10-1: Key Events

Solution on page 455

Write a program that registers an event listener on the `document` object for the “keydown” event, and have it print the event’s `key` and `shiftKey` information to the console. Remember that every event listener receives an event object as its first parameter, and in this case we are interested in the `key` and `shiftKey` members of that object (capitalization matters, so be careful of that):

- What happens when you press ‘a’? What happens when you press ‘A’ instead?
- What happens when you press the shift key, with nothing else?
- How do arrow keys appear when pressed?
- Log the entire event instead of just two members of it. What happens when you press other modifiers like the control, alt, or command keys?

Exercise 10-2: Click Events

Solution on page 455

Write a program that has a canvas. Each time you click the canvas, draw a line from the previous click location (start at 0, 0 by default) to the current click location. Note that you can use the event object’s `x` and `y` members to get the coordinates of the click within the canvas.

Exercise 10-3: More Mouse Events

Solution on page 456

Taking inspiration from the previous program, alter it so that it draws lines as before, but every time the mouse *moves* while the *button is down*. The events you will need to make this work are called “mousedown”, “mouseup”, and “mousemove”.

Hint: set a boolean to true when “mousedown” happens, and only draw in “mousemove” if it is true. Set it to false when “mouseup” happens.

Bonus: fix the program so that pressing the mouse button resets the line’s starting point.

Midterm 2

In the chapters leading up to now, we have covered several new concepts:

- Recurring for side-effects (like drawing on a canvas or outputting to the console)
- Recurring to build something, like an array
- Loops `while` and `for`
- The `switch` statement
- Random numbers
- Anonymous functions, passing them for event handling (e.g., to `setTimeout`)
- Immediate functions
- Variable shadowing
- Using time in animations, using `requestAnimationFrame`
- Click and key events

That’s a lot! In fact, that’s enough to do a lot of interesting things even if you learn nothing else about the language. There is still more to come, but now is a good time to take a breath and test what you’ve learned.

Before taking the test below, it might be a good idea to take some time to study these concepts in the preceding chapters.

Also, if your educational situation had you skipping much of the recursion in previous chapters, you might want to review it or ask to skip the related questions here.

Exercise 10-4: Output a Simple List of Numbers

Solution on page 458

Use recursion to output the numbers 1 to 100 in the console, in order. You may use any recursive strategy you like (for example, it is fine to either output then recur, or recur then output, it just depends on how you’re thinking of it).

Note: if you get an error about “maximum stack length” or something similar, just output fewer numbers, like 1 to 50. My browser let me do well over 10,000, however, so it is unlikely that you will run into this.

Exercise 10-5: Create an Array of Numbers

Solution on page 459

Use recursion to create an array consisting of all numbers from 1 to 100. Remember that if `a` is an array, you can create a new array consisting of `a` together with `b` by calling `a.concat(b)`. That doesn't change `a`, but creates a new array with the contents of `a`, followed by the contents of `b`. Use `concat` in your recursive code to create the array of numbers.

Any working recursive strategy is allowed, but no loops, and do not edit any array in place: always just return a new one.

Output the final array to the console when complete.

Note: it is fine to not make this efficient. Tail-call-optimizable is not a necessary condition for a correct answer, in case you were worried about that.

Exercise 10-6: Output Integers

Solution on page 460

Use a `while` loop to output the integers from 0 through 99 to `console.log`.

Exercise 10-7: Do It Again

Solution on page 461

This time, use a `for` loop to output the numbers from 1 through 99.

Exercise 10-8: Randomness, Key and Click Events, and Anonymous Functions

Solution on page 461

This exercise is more like a lab. Here we are going to ask you to write a program that creates two event listeners, one for key presses, and one for page clicks. Each of these is going to log a random number to the console, with text indicating which kind of event it was.

The requirements:

- Use `document.body` as the target for your events.
- In the click event, output "click:" and a random number (between 0 and 1).
- When a key is pressed down, output
 - the word "key"
 - the key that was pressed
 - if the key pressed is "Enter", output your name, otherwise
 - output a random number (between 0 and 1).
- Use the `switch` statement to determine whether the key was "Enter".
- Use anonymous functions as the event listeners.

Exercise 10-9: Animate With Frames

Solution on page 462

Using a canvas and `requestAnimationFrame`, make an animation. You can use any shape you like. Requirements: - Move at roughly 300 pixels per second, - Start in the upper left corner, - Move in a straight line, and - Stop when it hits an edge.

Note that “a straight line” can be diagonal. Since your shape is starting in the upper left corner, make sure that your straight line motion is to the right, down, or a little bit of both. Though potentially clever, drawing a shape in the upper left and saying “it is traveling left, and it stopped already” in order to avoid writing an animation loop is not really an acceptable answer. Your shape should be seen to be moving.

Note also that you should be writing HTML with JavaScript within it, this time, since you need a canvas element to operate on.

Bonus points: wrap all of your code in an immediate function.

Chapter 11

Behavioral Abstractions and Multi-File Programs

It's time for some more extensive refactoring. Our code is getting long enough that one listing doesn't fit well onto the page of a book, and that's not only a problem for book readers and authors, it's a problem for programmers in general. When a single function gets too long, it becomes harder and harder to follow what is happening, and therefore easier to accidentally introduce bugs. The solution to this is to take a careful look at what can be safely split out into its own **unit of abstraction**, which in this chapter will be the humble but powerful **function**.

More Files

We have, until now, written each of our programs as a single HTML file. In that file, we have our layout (`canvas` and `button` elements, for example) and our code (in `script` tags). Our code started out small, consisting of drawing on the canvas and some rudimentary animation. Once we started adding event listeners, though, the code nearly doubled in length. That's pretty typical, and when that happens it's a good idea to start looking for common functionality that we can move elsewhere.

We are about to enter the realm of multi-file projects, but we are going to do it with minimal changes to our normal workflow: just make sure that *all of the files we create are in the same folder*, and things should work out fine.

Create a new program HTML file called "main.html", but this time, it will have two script sections: one for the main program as usual, but another one above it that loads code from a totally different file that we haven't written yet:

:html:

```
<script src="animate.js"></script>
<script>
// The main program goes here.
</script>
```

The first script tag instructs the browser to look in the current folder (the place where this HTML file lives) for a file called “animate.js” and load it *as though it were typed between the script tags*. That’s all the script tag does with the `src` attribute: it slurps the contents of the given file and plops them down between the script tags. Note that you won’t see this; it happens behind the scenes.

The “.js” extension stands for “JavaScript”, by the way. The “animate.js” file won’t contain any HTML tags (including `script` tags!)—it will only contain JavaScript code.

This `src` approach is not a very sophisticated mechanism, and that has led to all sorts of module libraries being invented to make it easier to safely import other code from JavaScript itself, instead of from the HTML that contains it. That’s far too much to get into here, though. This course is about learning to program, not about finding all of the frameworks that might want to seduce you to their own version of modularity. Many of them are excellent, though, so eventually you might want to learn more. Also, modules are a first-class concept in later versions of JavaScript.

The mechanism we’re using here will do just fine for our pedagogical needs. It’s also still extraordinarily common, so you should be right at home when reading other people’s code that makes use of multiple JavaScript files.

As we mentioned before, the “animate.js” file doesn’t exist yet. When you load this program into the browser without that file (do try it), you will see an error in the console that the file couldn’t be found. Let’s fix that.

Go ahead and create the “animate.js” file in your favorite programming text editor, and save it right next to your new HTML file, in the same folder. You should now have at least two files in there: a new “main.html” file and the “animate.js” file. Don’t proceed until you do, otherwise this will get confusing in a hurry. With the empty “animate.js” file next to “main.html” in the same folder, you should be able to load your “main.html” program and have no errors in the console. Congratulations, you just made a program consisting of two files! It doesn’t do anything yet, but still!

Hiding the Plumbing

Remember our little animated spaceship? In that program, and those leading up to it, we have been seeing the same pattern over and over again, particularly where `tick` and surrounding variables are concerned. Here’s the template (pattern) that we have been following:

:javascript:

```

var lastTime = 0,
    frame = null,
    EXPECTED_DT = 1 / 60;

function tick(t) {
  if (done()) {
    frame = null;
    return;
  }
  frame = requestAnimationFrame(tick);

  var dt = (t - lastTime) / 1000;
  if (dt > 4 * EXPECTED_DT) {
    dt = EXPECTED_DT;
  }
  lastTime = t;

  draw();
  move(dt);
}

frame = requestAnimationFrame(tick);

```

If you look closely, you might notice that nowhere in here is a canvas or context mentioned. Neither is there anything to do with spaceship motion or locations. They aren't needed here—this function knows about how to trigger animation frame drawing, but doesn't know anything about spaceships, canvases, contexts, positions, or anything specific to our previous animation. All it needs to know is what functions to call to draw, move, and tell it to be done. Those operations are now abstract rather than concrete. We know that something will happen when `draw` is called, but we don't know what, and we don't need to, so long as there are appropriate functions to call.

In other words, we have taken a behavioral pattern like “animating stuff using frames” and removed all of the specifics from it so that it can work for any animation at all! We have expressed the *idea* of animation in a much more general way. If we can provide the functions above somehow, this will work for all kinds of stuff and we won't have to think about the details of teleportation, pausing, and requesting animation frames anymore—that will already be done. That is the essence of abstraction, one of the goals of refactoring, and it becomes ever more important as our programs become more interesting and capable.

But how do we make use of this abstraction in a real animation? The answer is to turn it into a function. Not only will we move all of the above code into a function that we can use from anywhere, but we will put that function out of the way in the “animate.js” file. Then, whenever we need to do animation, we can just copy that file into our program's folder and include it.

This is why we have two files now: one of them is a **library** file (“animate.js”), and the other is our **main program** file (“main.html”). Libraries are collections of code that are basically

“finished”. You put them into your projects to use, not to edit.

Because you will be working with two files at once (this particular library is not finished, so we *will* be editing it), make sure you know how to switch between “main.html” and “animate.js” in your editor of choice. If you are using anything reasonable (even relatively simple environments like Text in Chrome work just fine), your editor should make this easy, certainly easier than finding and opening the file again. Find out how to switch quickly between files and it will save you loads of hassle later on. Many editors, for example, let you open files in multiple tabs, much like pages in a web browser, and you can switch between them in a similar way.

An Animation Function

The first thing to do is to organize the above code into its own function, inside of “animate.js”. In that file, we might start like this:

:javascript:

```
// animate.js

// animate sets up an animation loop.
//
// Arguments:
// - move: a function accepting a duration dt,
//        called when it is time to position things.
// - draw: a function called when it is time to draw.
// - done: a function that returns `true` only if the
//        animation is permanently finished.
function animate(move, draw, done) {
    var lastTime = 0,
        frame = null,
        EXPECTED_DT = 1/60;

    function tick(t) {
        if (done()) {
            frame = null;
            return;
        }
        frame = requestAnimationFrame(tick);

        var dt = (t - lastTime) / 1000;
        if (dt > 4 * EXPECTED_DT) {
            dt = EXPECTED_DT;
        } else if (dt < 0) {
            dt = 0;
        }
        lastTime = t;

        draw();
        move(dt);
    }
}
```

```
}  
  
    frame = requestAnimationFrame(tick);  
}
```

That looks pretty familiar, right? We basically just took our animation code and shoved it into a function called `animate` inside of “`animate.js`”.

But, there are a couple of additional things to note. First, it is now clear where `move`, `draw`, and `done` are going to come from: they are parameters to our `animate` function. They will need to be passed in when the function is called. Second, we have **documented** our new function with a fairly structured comment. This is a good practice in general: if you don’t document how to actually call your function, anyone who wants to use it will have to guess by looking at your code, and that is a seriously not fun thing to do.

With all that said, documentation takes up a lot of space, so we will cheat in this book and not show it most of the time. In real life, you usually can’t go wrong overdoing it a bit on library documentation; your future self will thank your current self later.

This does not mean that *everything* should be documented. Too many comments in code can be detrimental for several reasons, including a tendency for comments to get out of sync with code changes and a lot of unnecessary noise when what the code does is really clear already.

Assume your audience knows JavaScript, but doesn’t know what you’re trying to do or how to use *your* function. Then comment accordingly.

You might also note that we are now checking for a negative value of `dt`, when we were not doing that before. In the process of testing this code, I found that some browsers start with a negative value for `t` on the first call to `tick`! That is really weird, so it became important to check for negative `dt` values. This is another power of abstractions: if all of our animation code is using this function, we can fix bugs like this once and have every user benefit immediately instead of having to hunt down all of the times that we calculate `dt`. Of course, that also means that introducing new bugs into the function will affect more programs, too. It does cut both ways, but it is still the right thing to do, even if care is still required.

Animating With Abstraction

Let’s see what we can do with this animation function as it currently stands.

In your shiny new “`main.html`” file, the one with two script tags, it’s time to add a canvas element. To make things a bit more interesting, we are going to build a brand new animation instead of just launching a spaceship. This time we will bounce a ball around.

To start, set up your file like this:

:html:

```

<canvas id="drawing" width="300" height="300"></canvas>
<script src="animate.js"></script>
<script>
var canvas = document.getElementById('drawing'),
    ctx = canvas.getContext('2d');

// Animation stuff goes here.
</script>

```

Take a moment to ensure that everything in here makes sense, including the canvas tags, the script that loads “animate.js”, and the main program script with the code to get a canvas and a context. There is nothing new here, but it might be organized a little bit differently than you are used to.

Specifying Animation Routines

Our `animate` function is available to us here because we included the “animate.js” file just above our program’s script tags, and it was defined in the global scope. Therefore, we can just call it. The question is what arguments do we give it? It requires a function that moves things, a function that draws things, and a function that tells it when we are done. Let’s start with empty versions of these for now, except maybe for the `done` function, which will always return `false`:

Note that we could just leave `done` empty, as well, because `undefined` is the value produced when a function doesn’t return anything, and `undefined` is falsy. The default behavior, therefore, is to never be finished.

:javascript:

```

function move(dt) {
    console.log("move", dt);
}

function draw() {
    console.log("draw");
}

function done() {
    return false; // never done
}

animate(move, draw, done);

```

Incidentally, when creating functions to be passed elsewhere and called later like we do here, we often say they are **callback** functions. There is nothing special about a callback

function; it's just a function. We merely use that name to indicate how they are *used*. When you create an event listener and register it, that's creating and registering a callback function. When we create `draw` or `move` or `done`, etc., we are creating callbacks, and we pass those into the `animate` function, which "calls them back" at appropriate times.

If we run this now, nothing will happen in the browser window, but the console will go crazy with messages. The animation is running! If you want it to stop after a few frames, you can change `done` to update a counter and return `true` after it reaches a certain number, like this:

:javascript:

```
var numFrames = 0;
function done() {
  ++numFrames;
  return numFrames >= 50;
}
```

That will return `true` when `done` has been called 50 times, and the animation (just console messages for now) will stop. Give it a try. It really is that easy.

Because we expect `true` or `false` to come from `done`, do you see exactly how we did that? Remember that comparators like `>=` return boolean values, so we can just return the result of one of those expressions to get the answer to "Are we at or over 50 frames yet?"

This way, `done()` returns `false` if `numFrames < 50` and `true` once it gets to be greater or equal to 50.

This abstraction is looking pretty good so far—to build a new animation we don't need to think about how to compute time anymore, or how to deal with pauses that happen when we navigate away, or when to move and draw. We just have to provide these three functions that deal with only what we want to think about right now. A good abstraction will do that for you. It gets things off your mind that you don't need to be thinking about so you can focus on more immediate tasks. We already solved these problems once, after all. We shouldn't have to solve them every time we want to do something new.

Tossing the Ball

We still haven't drawn anything, though. Let's do that now. We'll draw a ball in the center of the canvas:

:javascript:

```
function draw() {
  ctx.beginPath();
  ctx.arc(canvas.width / 2, canvas.height / 2,
    10, 0, 2 * Math.PI);
  ctx.fill();
}
```

Yeah, drawing circles is harder than it should be on the canvas. You have to draw an arc that covers all 2π radians (360 degrees) of rotation. Let's pause and create a `fillCircle` function inside of "animate.js" to make it simpler:

:javascript:

```
// ... somewhere in animate.js ...

function fillCircle(ctx, x, y, r) {
  ctx.beginPath();
  ctx.arc(x, y, r, 0, 2*Math.PI);
  ctx.fill();
}
```

Now when we include "animate.js", we also get `fillCircle` coming along for the ride. When we want to draw a circle, we can just call it like this in our `draw` function:

:javascript:

```
function draw() {
  fillCircle(ctx, canvas.width / 2, canvas.height / 2, 10);
}
```

That's better. It is easier to specify `(x, y, r)` for a circle and let the function handle things that remain the same. We no longer have to remember the angle arguments, and it's all one line to actually fill it. Having this extra library file is kind of handy. What is handier still is the fact that you can just copy it to your next project and it will be ready to use if you need it.

Dropping the Ball

We have now specified `done` and `draw`, but we don't really have any animation going on. That's kind of boring, so let's fix that now. We currently have three small functions defined, and they are all going to need to know about the ball's position: `move` needs to know so it can update it, `draw` needs to know so it can draw it, and `done` needs to know so things can stop if the ball leaves the canvas. That means these are all going to be closures, so we should set up some variables in their outer scope, like this:

:html:

```
<canvas id="drawing" width="300" height="300"></canvas>
<script src="animate.js"></script>
<script>
var canvas = document.getElementById('drawing'),
    ctx = canvas.getContext('2d');

var RADIUS = 10,
    xVel = 0,
    yVel = 100,
    x = canvas.width / 2,
    y = canvas.height / 2;
```



```

function move(dt) {
  x += xVel * dt;
  y += yVel * dt;
}

function draw() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  fillCircle(ctx, x, y, RADIUS);
}

function done() {
  return y + RADIUS > canvas.height;
}

animate(move, draw, done);
</script>

```

That gets us started. Keep in mind that nothing here is new to you, really. You have seen it all before. We’re simply using three nice, simple functions to do the work instead of one big, ugly one.

The `move` function adds the time-scaled velocities to `x` and `y`, as we learned to do previously. This creates a new position. In this case, though, we have both `xVel` and `yVel`. `yVel` is 100, so `y` will move down the canvas at a rate of 100 pixels per second; `xVel` is 0 so the ball will not move sideways at all. The `draw` function clears the canvas and draws a circle at `x` and `y` with the specified radius. The `done` function returns `true` when the ball’s bottom edge has hit the bottom of the canvas. Shall we try it?

This basically works! And, the code we wrote in “main.html” is only what is relevant to this specific animation. The generic animation stuff is all in the `animate` function inside of “animate.js”. That’s very neat. Literally. Our code is neater because of it. You know, in the organizational sense, like “tidy” neat. Like “you just cleaned your room and found a floor there” neat.

These things matter.

Moving Diagonally

As animations go, this isn’t the most interesting thing we could do. What would be more fun would be to have the ball move diagonally around the canvas. Our code already handles any velocity in either the horizontal or vertical direction, so all we have to do is set it differently. Here is what it looks like if we use a non-zero `xVel` to produce some sideways motion while the ball drops:

:javascript:

```

var RADIUS = 10,
    xVel = 110,
    yVel = 100,
    x = canvas.width / 2,
    y = canvas.height / 2;

```

Well, that was pretty easy. Only one value changed. As before, the ball drops, but this time diagonally to the right, before stopping at the bottom edge (off the right side).

Bouncing Off the Walls

The last item of business is to get the ball bouncing off of the walls. This is where things get interesting. In order to make this happen, we have to think about what “bouncing” means in terms of the mathematics of motion. In a very basic sense, it means to reverse direction. If a ball bounces off of the right wall, for example, it must have had a positive horizontal velocity. When it finished bouncing, it will have a similar but negative horizontal velocity. That logic applies to all bounces. Here is what bouncing off the right wall might look like:

:javascript:

```

function move(dt) {
  if (x + RADIUS >= canvas.width) {
    xVel = -Math.abs(xVel);
  }
  x += xVel * dt;
  y += yVel * dt;
}

```

All we did was add a test: if the ball is going past the right side, make sure the new `xVel` is negative. Why not just say `xVel = -xVel`? Because the ball might be just barely far enough off the right side that reversing course won’t get it all the way back onto the canvas. In that case, it will reverse again next time, and we really don’t want that. So, we take the absolute value of the velocity and make absolutely sure that the new velocity is negative. Yes, this was a fix applied after sad experience.

There is actually an important principle hiding in this small matter of whether to use `Math.abs` when bouncing, and it is this: try to make your reactionary code *idempotent*. In other words, if your code executes twice because the condition is still true, make sure that the result *doesn’t change* after the first execution. That’s what happens in the case of the ball getting slightly off the edge: the bounce code can sometimes execute twice for the same bounce: the first time starts it moving away from the edge, but it hasn’t quite escaped it yet (still some overlap), so the condition is true next time around, as well.

In that case, if you merely “reverse direction” from what you had before, your ball ends up jittering forever, stuck with a little bit of it outside of the canvas. The condition triggers, you reverse, then reverse again, etc., so on forever.

The `xVel = -Math.abs(xVel)` statement ensures that *no matter how many times it is called* for a right-side collision, the x velocity will *always end up moving it away*. That is a really nice example of idempotence, and a principle to remember.

The same thing can be done for, say, the bottom wall:

:javascript:

```
function move(dt) {
  if (x + RADIUS >= canvas.width) {
    xVel = -Math.abs(xVel);
  }
  if (y + RADIUS >= canvas.height) {
    yVel = -Math.abs(yVel);
  }
  x += xVel * dt;
  y += yVel * dt;
}
```

But this might not actually bounce because of the way our `done` function currently works. You can delete the contents of that function if you want to see these two wall bounces succeed, but it will work much better if you first have all four wall bounce routines in place:

:javascript:

```
function move(dt) {
  if (x - RADIUS <= 0) {
    xVel = Math.abs(xVel);
  } else if (x + RADIUS >= canvas.width) {
    xVel = -Math.abs(xVel);
  }

  if (y - RADIUS <= 0) {
    yVel = Math.abs(yVel);
  } else if (y + RADIUS >= canvas.height) {
    yVel = -Math.abs(yVel);
  }
  x += xVel * dt;
  y += yVel * dt;
}
```

There you have it. We guarantee that wall collisions put us back on the right track using `Math.abs` and properly-placed negatives based on which wall we have run into.

Finally, we just need to allow the animation to continue all the time:

:javascript:

```
function done() {
  return false;
}
```

And with that, we have a bouncing ball that stays within bounds and doesn't stop.

Furthermore, we did it without having to rewrite all of the timing stuff that we abstracted away into the “animate.js” file. When you look at the main program, all you see is code pertinent to this specific animation. There is no longer any boilerplate for calculating dt and requesting animation frames, just code to move and draw a ball on a canvas.

Here’s the full program (without the contents of “animate.js”):

:html:

```
<canvas id="drawing" width="300" height="300"></canvas>
<script src="animate.js"></script>
<script>
var canvas = document.getElementById('drawing'),
    ctx = canvas.getContext('2d');

var RADIUS = 10,
    xVel = 110,
    yVel = 100,
    x = canvas.width / 2,
    y = canvas.height / 2;

function move(dt) {
    if (x - RADIUS <= 0) {
        xVel = Math.abs(xVel);
    } else if (x + RADIUS >= canvas.width) {
        xVel = -Math.abs(xVel);
    }

    if (y - RADIUS <= 0) {
        yVel = Math.abs(yVel);
    } else if (y + RADIUS >= canvas.height) {
        yVel = -Math.abs(yVel);
    }

    x += xVel * dt;
    y += yVel * dt;
}

function draw() {
    ctx.clearRect(0, 0, canvas.width, canvas.height);
    fillCircle(ctx, x, y, RADIUS);
}

function done() {
    return false;
}

animate(move, draw, done);
</script>
```

Abstractions That Provide Behavior

The abstraction we wrote is nice because it keeps all of the subtle and ugly timing code out of the way. When we want to animate something now, pretty much anything, in fact, we can just use “animate.js” and call `animate` with three functions very specific to our animation. Time is handled for us.

You may have noticed, however, that our abstraction is not quite as flexible as what we used to have. In particular, we have the `frame` variable but no way of pausing or restarting the animation once it has been paused. Our pause button situation is looking bleak. How can we add pause functionality to our `animate` abstraction?

We can’t get at the `frame` variable from outside of `animate`—that’s stuck in the function’s local scope. At least, we can’t get at it directly. We could, however, return a function from `animate` that closes over that variable and allows it to be manipulated. In fact, we can return multiple such functions from `animate` to allow outside callers to control things. Let’s start by adding a `pause` capability, and we will add more shortly afterward.

Pausing Again

The `frame` variable is pretty much all we need in order to allow the animation to be paused. We just need to expose functionality around it somehow. Here is one technique for doing just that. Recall our `animate` function:

:javascript:

```
// ... somewhere in animate.js ...

function animate(move, draw, done) {
  var lastTime = 0,
      frame = null,
      EXPECTED_DT = 1/60;

  function tick(t) {
    if (done()) {
      frame = null;
      return;
    }
    frame = requestAnimationFrame(tick);

    var dt = (t - lastTime) / 1000;
    if (dt > 4 * EXPECTED_DT) {
      dt = EXPECTED_DT;
    } else if (dt < 0) {
      dt = 0;
    }
    lastTime = t;

    draw();
  }
}
```

```

    move(dt);
}

frame = requestAnimationFrame(tick);
}

```

This function doesn't return anything, but it could. What we are going to do next is create a function inside of this one and return it. The full listing is given to make it clear where it goes. Note the comment indicating what is new:

```

:javascript:

// ... somewhere in animate.js ...

function animate(move, draw, done) {
    var lastTime = 0,
        frame = null,
        EXPECTED_DT = 1/60;

    function tick(t) {
        if (done()) {
            frame = null;
            return;
        }
        frame = requestAnimationFrame(tick);

        var dt = (t - lastTime) / 1000;
        if (dt > 4 * EXPECTED_DT) {
            dt = EXPECTED_DT;
        } else if (dt < 0) {
            dt = 0;
        }
        lastTime = t;

        draw();
        move(dt);
    }

    frame = requestAnimationFrame(tick);

    // NEW:
    // Create a function that closes over
    // the "frame" variable.
    function pause() {
        if (frame) {
            cancelAnimationFrame(frame);
            frame = null;
        }
    }

    // Return the pause function.

```

```
    return pause;
}
```

The new parts are at the end. Remember how closures work? When you create a function, it remembers all of the variables and functions in its outer scope. That means this function can see `frame`, `lastTime`, and `EXPECTED_DT`. It can also see `tick`. The thing that makes closures special is that they can always see those things, no matter who calls them, or from where they are called.

When we call `animate` now, it will return a function that we can use to pause the animation, like this:

:javascript:

```
// ... somewhere in main.html ...

var pauseFunc = animate(move, draw, done);

// Now if you click on the canvas, it will pause.
canvas.addEventListener('click', pauseFunc);
```

There are a few things going on here, so you might need to break this down to understand what is happening. First, we created a `pause` function inside of the `animate` function. That `pause` function can see `frame` and even change it to `null` because `frame` is part of its lexical closure. The `pause` function is then returned by `animate`, allowing callers to use it if they want to.

We take that return value and assign it to a variable named `pauseFunc` when we create our animation. That variable now holds the pause function that `animate` created for us, and that function can see the `frame` variable that we need to set. Note that we can't see the variable, but if we call `pauseFunc`, it can, and therefore it can do what we ask it to. This is sometimes called **encapsulation**, because we are getting access to something we need, but we can't get at it directly. It's inside of a capsule of sorts, and we can only access it by calling the function we've been provided.

Finally, we tell the canvas that it should call our pause function whenever it is clicked.

If you run this, the animation proceeds as normal, but clicking on the canvas pauses it!

Resuming Again

Pausing is great, but it is better if we can also resume things later. We don't currently have any way to do that. In order to make that happen, we need to run `frame = requestAnimationFrame(tick)`, so let's create a function that does that for us and return it with our pause function.

Since we now need to return two closures from our `animate` function, we will return an object that contains both of them. In fact, we'll do it with anonymous functions this time just to

get some more practice with that concept:

:javascript:

```
// ... somewhere in animate.js ...

function animate(move, draw, done) {
  var lastTime = 0,
      frame = null,
      EXPECTED_DT = 1/60;

  function tick(t) {
    if (done()) {
      frame = null;
      return;
    }
    frame = requestAnimationFrame(tick);

    var dt = (t - lastTime) / 1000;
    if (dt > 4 * EXPECTED_DT) {
      dt = EXPECTED_DT;
    } else if (dt < 0) {
      dt = 0;
    }
    lastTime = t;

    draw();
    move(dt);
  }

  frame = requestAnimationFrame(tick);

  // NEW: Changes here.
  return {
    'running': function() {
      return !!frame; // convert to true/false
    },
    'pause': function() {
      if (frame) {
        cancelAnimationFrame(frame);
        frame = null;
      }
    },
    'start': function() {
      if (!frame) {
        frame = requestAnimationFrame(tick);
      }
    },
  };
}
```

Again, the changes are at the bottom. This time we are creating and returning a new object all

at once, and it has several keys: “running”, “pause”, and “start”. The `running` function just allows outsiders to know whether the animation is busy running or not. We use the common trick of double-boolean-inversion to get a true/false value out of a truthy or falsy object. We’ve seen the `pause` function already, and the `start` function kicks things back into motion in the familiar way. The only noteworthy thing is that we test whether things are running or not before starting up again. It wouldn’t make much sense to start an already-running animation.

Moving on, we can use the functions in this object to implement `togglePaused` again (in “main.html”):

:javascript:

```
// main.html

// Functions move, draw, and done omitted,
// but they would go here.

var animation = animate(move, draw, done);

function togglePaused() {
  if (animation.running()) {
    animation.pause();
  } else {
    animation.start();
  }
}

canvas.addEventListener('click', togglePaused);

// Add space bar pausing for good measure.
document.addEventListener('keydown', function(event) {
  if (event.key === ' ') {
    togglePaused();
  }
});
```

See how that works? The `animate` function returns an object with members `running`, `pause`, and `start`, all of which are functions we can call to do what we need. We store that object in the `animation` variable.

The `togglePaused` function uses `animation.running()` to determine whether it needs to pause or start the animation. That `running` function is pretty handy for cases like these. If the animation is running, then it calls `animation.pause()` to pause it, otherwise it is already paused and it calls `animation.start()` to get it going again.

This idea really showcases the power of having closures at our disposal. You can create systems where you pass behaviors in (like `move`) and those systems can create and provide new behavioral controls that get passed out (like `start`). Again, this idea is often called **encapsulation** because it takes functionality, hides it away inside a capsule (a function), and

only allows you to get at certain bits of it that it wants to expose. The idea of animation is now encapsulated in the `animate` function and the closures it provides us.

You can see how you would expose even more of the internals through other closures as you see fit. For example, you could move the `togglePaused` function into “animate.js” by returning it as a closure alongside the other functions returned.

Summary and Listings

In this chapter we learned how to

- Move code into a separate file,
- Create an abstraction that uses callbacks,
- Allow that abstraction to pass controls back to us, and
- Animate something in two dimensions, with collisions.

That is actually a lot. The important thing is that you are equipped to understand all of it. No new language concepts were introduced (other than the separate file), just more practice with the ones you already know.

As promised, the complete listings of both files come next.

main.html

:html:

```
<canvas id="drawing" width="300" height="300"></canvas>
<script src="animate.js"></script>
<script>
var canvas = document.getElementById('drawing'),
    ctx = canvas.getContext('2d');

var RADIUS = 10,
    xVel = 110,
    yVel = 100,
    x = canvas.width / 2,
    y = canvas.height / 2;

function move(dt) {
  if (x - RADIUS <= 0) {
    xVel = Math.abs(xVel);
  } else if (x + RADIUS >= canvas.width) {
    xVel = -Math.abs(xVel);
  }

  if (y - RADIUS <= 0) {
    yVel = Math.abs(yVel);
  } else if (y + RADIUS >= canvas.height) {
```

```

    yVel = -Math.abs(yVel);
  }
  x += xVel * dt;
  y += yVel * dt;
}

function draw() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  fillCircle(ctx, x, y, RADIUS);
}

function done() {
  return false;
}

var animation = animate(move, draw, done);

function togglePaused() {
  if (animation.running()) {
    animation.pause();
  } else {
    animation.start();
  }
}

canvas.addEventListener('click', togglePaused);
document.addEventListener('keydown', function(event) {
  if (event.key === ' ') {
    togglePaused();
  }
});
</script>

```

animate.js

:javascript:

```

function animate(move, draw, done) {
  var lastTime = 0,
      frame = null,
      EXPECTED_DT = 1/60;

  function tick(t) {
    if (done()) {
      frame = null;
      return;
    }
    frame = requestAnimationFrame(tick);

    var dt = (t - lastTime) / 1000;

```

```

    if (dt > 4 * EXPECTED_DT) {
        dt = EXPECTED_DT;
    } else if (dt < 0) {
        dt = 0;
    }
    lastTime = t;

    draw();
    move(dt);
}

frame = requestAnimationFrame(tick);

return {
    'running': function() {
        return !!frame; // convert to true/false
    },
    'pause': function() {
        if (frame) {
            cancelAnimationFrame(frame);
            frame = null;
        }
    },
    'start': function() {
        if (!frame) {
            frame = requestAnimationFrame(tick);
        }
    },
};
}

function fillCircle(ctx, x, y, r) {
    ctx.beginPath();
    ctx.arc(x, y, r, 0, 2*Math.PI);
    ctx.fill();
}

```

Exercises

Exercise 11-1: Library Functions

Solution on page 465

Add a `clearContext(ctx)` function to “animate.js” that clears the canvas. Note that you can get the underlying canvas object from the context thus: `ctx.canvas`.

Exercise 11-2: Toggle Pause Behavior

Solution on page 465

Our `animate` library function returns several useful behaviors as functions, stored in an object. Since toggling paused state is something we'll want to do with many animations, move the `togglePaused` function into `animate` and return it with the rest.

Chapter 12

Our First Game: State, Configuration, Clocks, and Winning

Now that we have worked out a pretty useful abstraction for animation, it's a lot easier to get one going without remembering all of the boilerplate for time and frame management, including pausing. All we have to do is provide `move`, `draw`, and `done` functions to our `animate` abstraction, and it gives us back the controls we need.

Furthermore, we also have a little bit of experience getting events from the keyboard, which we have thus far used to pause animations.

With all of this at our disposal, we are ready to make a complete, interactive game. What game should we make? Given that we already know how to make a ball bounce around and interact with the edges of the canvas, let's adapt that idea to using a paddle to keep it from falling off the bottom edge. We will use the keyboard to control the paddle.

Our code is getting longer, so this chapter is full of code “snippets”, little bits of text that don't show all of their surrounding context. That means you will want to follow along in your own editor, otherwise it will be easy to get lost. From here on out, homework will become more like labs all the time, and the expectation is that you will have followed along through the chapter and can continue your assignments where the text left off. So, dust off your text editor, start a project, and type things in and test them as you read.

In other words, ***follow along*** through this chapter, with hands on the keyboard, writing the code and running it as you go.

If you get lost, don't worry, that happens to everyone. Just read back a bit, reorient yourself, and try again.

A Matter of State

Let's revisit the idea of the “pause” control. When you pause the animation, it is as though time has stopped for it. When you resume it, it begins again from where it left off. How *exactly* does it do that?

The easy but incomplete answer is that we simply stop calling `tick` (by canceling the animation frame request). When we pause, `tick` doesn't get called. That is absolutely correct, but it is useful to step back and think about what happens outside of “animate.js”, specifically in the HTML file where the behaviors of our animation are actually defined. All of that business with `tick` and `frame` (or `running`, for the earlier approach) is just an implementation detail for our animation loop. We shouldn't normally have to think about things at that level, not now that we have abstracted it away into the `animate` function. What we really want to think about is what it means for time to “stop” for our animation.

In reality, we already know that time doesn't truly stop, even in the context of the web page the game is running in. When the game resumes, it gets the number of milliseconds since the page was loaded, just like it always has. That means there will be a big jump in the time elapsed, which the `tick` function handles by clipping it to `EXPECTED_DT`.

What it really means for time to stop is this: the state of the *things we care about* does not change. The “state” of something in this case is “the values of things we are keeping track of”. For the bouncing ball animation, that means `x`, `y`, `xVel`, and `yVel`, so when time stops, those are not changing and the ball is not moving. Remember this clause at the beginning of our ball animation?

:javascript:

```
var RADIUS = 10,
    xVel = 110,
    yVel = 100,
    x = canvas.width / 2,
    y = canvas.height / 2;
```

At least some of those variables are changed in `move`, and accessed in `draw`. They represent the **current state**, or status, of the animation. Time is only useful in that it tells us how and when to change that state. If the state doesn't change, it doesn't matter how much time is going by; our little world, the one that we created here in our program, is stopped.

Technically, there was another bit of state in our animation, and that was whether it was currently paused or running. As this is a piece of state in *every* animation or game we might conceive of, we will leave it implied, but it is definitely state. You can tell by looking at how we deal with it to pause or resume: what we see in `animation.running()` determines how we behave.

Time travel, if such a thing existed, would have to work like this, really. If you were to travel back in time, that would mean that every particle of the universe would have to go back to the

state it was in at that time, perhaps with the magical exception of yourself, somehow protected by a fictional machine that got the universe to do that in the first place. It works the same way in animations and games: time travel, or even stopping time, is really all about changing state.

When creating a game or an animation or even a boring business web page, figuring out what the state should contain is often the first order of business. What does the page need to remember? What parts of it can change? What should happen when these things change? These are the questions to ask. Let's ask them for our one-player Pong knockoff:

- What does the page need to remember?
 - The position of the ball and the position of the paddle,
 - The number of seconds of play time so far,
 - The score, and possibly
 - The speed of the ball and the width of the paddle, if we make the game more challenging as time progresses.
- What parts of the state can change?
 - The paddle position changes with user input (like key presses), and the position and direction of the ball change with time and collisions.
 - If we alter difficulty over time, the speed of the ball and the width of the paddle change.
 - The score changes each time we get a successful hit.
 - Time changes constantly without our help.
- What should happen when these things change?
 - The paddle position and width affect the way it is drawn and how easy it is to hit,
 - The ball position and speed affect where it is drawn,
 - A clock should be shown in the game window, and
 - A score counter should be shown in the game window.

OK, wow. For a simple game, there sure are a lot of interesting answers up there. We will tackle all of those in this very chapter, and by the end of it, the game will have a clock, a scoreboard, a moveable paddle, and a ball that bounces off of it convincingly. Let's get started!

Where To Store State

With our recent animation, we stored state in variables that lived in the surrounding scope of the `move`, `draw`, and `done` functions. When we declared them, we also set them to their beginning values. This is the **initial state**.

Then we called `animate` and handed it our closures, which access and change these state variables to do their work.

Another deep question is this: why doesn't all of this state disappear when our main program falls off the end? The answer is possibly a bit surprising, and it has everything to do with

registering listeners.

When we call `requestAnimationFrame`, we pass it the function `tick`. That function closes over variables like `frame` and `EXPECTED_DT`, so those stay alive as long as `tick` does, and `tick` stays alive (meaning it is not erased to reclaim space) as long as it is registered with `requestAnimationFrame`.

Back when we only had the ability to pause, not to resume, we could call `cancelAnimationFrame(frame)` to stop the animation. Consider for a moment what that means: any animation frame that was waiting to call `tick` is canceled, so `tick` is never called, nothing remembers it anymore, and our program finishes. At that point (or possibly a bit later), all of that stuff (`tick`, its surrounding variables, etc.) is deleted; the browser has no need of any of it anymore. We couldn't start it back up if we wanted to: it's out of our reach.

When we added back the ability to resume, how did we do that? We added an event listener to a button that is always on the page, and gave it another closure to call. This closure could see the `tick` function; it had to in order to reregister it with `requestAnimationFrame`, so that kept it from being deleted: the interpreter doesn't delete things when it can tell that they might be needed again, and it could tell in this case because we registered a function that can see `tick` (a function that resumes things).

In other words, the pause button sees the event listener, the event listener sees `tick`, and `tick` sees its enclosing scope, so all of that stuff sticks around as long as the button is there. The *pause button* is what keeps the code around!

In our most recent animation, we call `animate`, which returns an object whose values are closures. Those closures also have access to the `tick` function and all of the surrounding variables, so all of that stuff stays alive as long as our `animation` object does. That object is in the surrounding scope of `togglePaused`, and that is in the surrounding scope of the keyboard event listener that we register. The browser is holding onto all of those listeners, so it is also keeping all of their transitive dependencies. So things *also* stick around because the keyboard listener is registered for us to use the spacebar for toggling the pause state.

It's kind of mind-boggling when you think about it that way, but also really sensible. The reason our code sticks around after our main program exits is because something (a button, or a key press) is hanging around that will need to access it, and the interpreter detects that for us. Thus, our state, currently in the outer scope of our animation functions, sticks around as long as there's a registered listener that might use it.

There are many ways to reclaim useless objects in a runtime environment, including "reference counting" and "mark and sweep" (and similar) garbage collection. We won't get into either of these, but it's good to know the vocabulary in case you would like to learn more about how the language is helping you out.

Animation Augmentation

It's time to improve our `animate` function a little more. There are a few useful things that we can have it do for us that aren't yet in there.

Elapsed Animation Time

One thing that jumped out at me as I was looking over the list of things we need to keep track of was the need to keep track of total time played. That seems like something our animation object should be able to tell us. In fact, it might be hard to get it any other way without a lot of repeated effort, since time is getting manipulated in very particular ways by `animate`. We should just update the `animate` function to track all of that for us!

For reference, here is the starting point for our `animate` function:

:javascript:

```
// ... somewhere in the file animate.js ...

function animate(move, draw, done) {
  var lastTime = 0,
      frame = null,
      EXPECTED_DT = 1 / 60;

  function tick(t) {
    if (done()) {
      frame = null;
      return;
    }
    frame = requestAnimationFrame(tick);

    var dt = (t - lastTime) / 1000;
    if (dt > 4 * EXPECTED_DT) {
      dt = EXPECTED_DT;
    } else if (dt < 0) {
      dt = 0;
    }
    lastTime = t;

    draw();
    move(dt);
  }

  frame = requestAnimationFrame(tick);

  return {
    'running': function() {
      return !!frame;
    },
    'pause': function() {
```

```

    if (frame) {
        cancelAnimationFrame(frame);
        frame = null;
    }
},
'start': function() {
    if (!frame) {
        frame = requestAnimationFrame(tick);
    }
},
};
}

```

We need to add something that keeps track of total time elapsed (while running) and provides a function to get it from outside. The necessary changes will happen throughout and will be marked with a `// CHANGE` comment so you can see where the differences are. Before looking below, keep in mind that you already have all of the tools you need to build this yourself. Give it a try before peeking.

:javascript:

```
// ... somewhere in the file animate.js ...
```

```

function animate(move, draw, done) {
    var lastTime = 0,
        frame = null,
        elapsed = 0, // CHANGE
        EXPECTED_DT = 1 / 60;

    function tick(t) {
        if (done()) {
            frame = null;
            return;
        }
        frame = requestAnimationFrame(tick);

        var dt = (t - lastTime) / 1000;
        if (dt > 4 * EXPECTED_DT) {
            dt = EXPECTED_DT;
        } else if (dt < 0) {
            dt = 0;
        }
        elapsed += dt; // CHANGE
        lastTime = t;

        draw();
        move(dt);
    }

    frame = requestAnimationFrame(tick);
}

```

```

return {
  'running': function() {
    return !!frame;
  },
  'elapsed': function() { // CHANGE
    return elapsed;
  },
  'pause': function() {
    if (frame) {
      cancelAnimationFrame(frame);
      frame = null;
    }
  },
  'start': function() {
    if (!frame) {
      frame = requestAnimationFrame(tick);
    }
  },
};
}

```

Do you see the changes? They're fairly simple. We first create an `elapsed` variable, initialized to 0. Then we increment it by the change in time inside of `tick`, which will only happen while the game is running. Finally, we return a function that provides it in our animation object. To access the elapsed time in seconds, we can just ask that function to tell us, just like we ask it whether it is running by calling the corresponding function.

We could easily have put this into our main program, since we get `dt` in the `move` function, but it seems like something we will need all the time for all animations, so it makes sense to tuck it away in our “animate.js” library instead. Figuring out what should go into a library is something of an art that requires experience, and you won't always get it right at first. The neat thing is that you don't have to. It's your code, you can factor it any way you like, and you have the power to change your mind.

Pause Toggling

The `togglePaused` function is something that we are going to want to use all the time, and is also not going to change. That seems like an opportunity—perhaps we should move it into our `animate` abstraction and provide it as a function on the `animation` object. Indeed, that is a homework assignment from earlier. Still let's examine what it looks like and discuss it a little further.

We will just change the `animate` function's `return` clause and leave everything else alone. Here is what it looks like inside the `return` clause:

:javascript:

```

return {
  // ...other pre-existing stuff here...

  togglePaused: function() {
    if (frame) {
      cancelAnimationFrame(frame);
      frame = null;
    } else {
      frame = requestAnimationFrame(tick);
    }
  },
};

```

From now on, we can call this new `togglePaused` function returned by `animate` to do the work of pause toggling, which allows us to remove that function implementation from our main program. One more abstraction is now out of our way so that we can focus on our game!

Why didn't we implement it in terms of the functions `running()`, `start()`, and `pause()` like we had it before in our main program? Because inside the object where the `'togglePaused'` key exists, those functions can't be seen. They aren't in the scope of the object, but are rather inside of it. Object literals don't create scopes. If we wanted to reuse those functions, we would need to define them with names outside of the returned object, first. We wouldn't be able to use anonymous (or even named, inlined) functions the way we do. Rather than refactor things, we just implemented it slightly differently this time around.

Configuration Arguments and Defaults

Let's make a final change to our `animate` function: let's specify the `move`, `draw`, and `done` functions in an object instead of directly as arguments. We will provide that object as a single argument instead of three function arguments.

Why might we want to do this? For one thing, it makes it easier to see what we're doing when we call it. Right now you just have to get things in the right order. For another, it allows us to leave some things unspecified and have the `animate` function fill in a suitable **default**.

Let's look at how that works by making a couple more changes to `animate`. In particular, we will replace this

:javascript:

```

function animate(move, draw, done) {
  ...
}

```

with this

:javascript:

```
function animate(config) {
  var move = config.move || function() {},
      draw = config.draw || function() {},
      done = config.done || function() {};
  ...
}
```

What does this do? It basically says, “Set `move` to either the value of `config.move` or to an empty function if `config.move` isn’t there.” It works like that for the other two, as well. The variables `move`, `draw`, and `done` will either get the values passed in via `config`, or a suitable default if none exists. It takes advantage of some of the special behavior of `||`, which we will cover shortly.

You might notice that the “suitable default” for all of them is just an empty function that returns nothing (well, it returns `undefined`). This works as a default for all of them. It even works for `done`, since `undefined` is falsy. That was no accident. In general, it’s nicest when the “empty” or “zero” value of something is the most suitable default; it aids intuition.

Anyway, this does the trick. It allows us to call `animate` this way:

:javascript:

```
var animation = animate({
  move: function(dt) {
    console.log("move", dt);
  },
  draw: function() {
    console.log("draw");
  },
});
```

That’s kind of nice—we can name our arguments this way because they’re just keys in an object. We can also more clearly use anonymous functions, since they are right next to the key names. Also, we can omit `done` altogether because its default is fine for our needs right now. That’s handy!

Unfortunately, this also means that we just snuck in a concept that maybe wasn’t fully explained earlier, and that is how the boolean OR operator `||` actually works. Let’s explore that before moving on and using this new feature of `animate`.

Short-Circuit Boolean Operators

A curious feature of many programming languages is the behavior of the `||` (OR) and `&&` (AND) boolean operators. Let’s review just a bit about what those mean:

- `||` is the OR operator: truthy if either the left side **or** the right side (or both) is truthy. It is only falsy if both sides are falsy.

- `&&` is the AND operator: truthy only if both the left **and** right sides are truthy. It is falsy if either side is falsy.

These expressions are always evaluated left-to-right, and they have a property called “short-circuiting”. What this means is that if the left side makes the outcome inevitable, the right side is never evaluated. Consider this:

:javascript:

```
function loggingTrue() {
  console.log('got here');
  return true;
}

var outcome = true || loggingTrue();
```

What happens in this code? Is “got here” written to the console or not?

The answer is “no”. The logging never happens because the function `loggingTrue` is never called. Why is that? Because if either side of the `||` expression is truthy, the whole expression has to be truthy. The first thing we encounter there is a `true` value, so we already know that this expression will be `true` and we can be done. There is no need to continue.

A similar thing happens with `&&`: if either side is falsy, then the answer has to be falsy, so if the first thing it encounters is falsy, the outcome is inevitable, and the right side is not evaluated.

This should feel pretty reasonable. Now for the mind-bending part: the outcome of `||` or `&&` is not necessarily just `true` or `false`: it is *the value of one of its operands*. The logic of `||` is more succinctly described thus:

- Return the left-side value if it’s truthy,
- Otherwise return the right-side value.

Similarly, for `&&`, we have this:

- Return the left-side value if it’s falsy,
- Otherwise return the right-side value.

This behavior, coupled with short-circuiting, allows us to use `||` for the **default value** idiom:

:javascript:

```
var draw = config.draw || function() {};
```

If `config.draw` has a function value (functions are truthy), then `draw` will be assigned that function. Otherwise it gets whatever is on the right, truthy or not. In this case, if `config.draw` is falsy (e.g., `undefined`), then `draw` gets assigned a blank do-nothing function.

You can see how this is strictly more general than thinking of `&&` and `||` being applied only to boolean operands. They work just the way you would expect them to, but they have this

additional (and powerful) behavior of short-circuiting and returning one of their operands instead of just returning `true` and `false`. This works in many modern programming languages.

Again, the principle is this: if the left side makes the outcome inevitable, then that's the value of the expression: there is no need to evaluate the right side. But, if the right side is evaluated, that is the value of the expression. Both operators take the left operand if that's enough to get an answer, otherwise they take the right operand.

Writing the Game

Let's test all of this out by creating a new main program for our game, but with just a skeleton of functionality (we know we will need a canvas, so that's included even though we don't use it yet):

:html:

```
<canvas id="drawing" width="300" height="300"></canvas>
<script src="animate.js"></script>
<script>
var canvas = document.getElementById('drawing'),
    ctx = canvas.getContext('2d');

var animation = animate({
  move: function(dt) {
    console.log("move", dt, animation.elapsed());
  },
});

canvas.addEventListener('click', animation.togglePaused);
</script>
```

Much of this looks familiar. We are creating a canvas element just like before. We are pulling in our “animate.js” script, where our `animate` and `fillCircle` functions live. We have done that before, as well. And, in the beginning of our main HTML file, we have variables for the canvas and its context. That's all pretty familiar.

Where things get to be different is in the way that we create our animation. Instead of passing in three functions as in the call `animate(move, draw, done)`, we are passing in a single configuration object like this: `animate({...})`. That object has one key, “move”, and the value under that key is an anonymous function that logs timing information. Once you get used to seeing this sort of thing, it makes it easier to see what the things being passed into a function actually mean.

Our configuration strategy also allows us to not bother with the `done` and `draw` functions, since they now have suitable defaults.

Finally, instead of implementing `togglePaused` like we have in the past, we are just

passing `animation.togglePaused` directly into our event listener. That function is already defined, and it's part of our `animate` object, so we can just register it as the canvas click listener.

To see this in action, open the console, load this into the browser, and try pausing and resuming things by clicking on the canvas. What happens to elapsed time in the console (shown in seconds)?

Displaying Text on a Canvas

Let's put that elapsed time on the canvas instead of the console. The time might be a number, but when we want to display it, we're going to be interested in the text describing that number: a sequence of digit characters that we will interpret as seconds. That means we need to display a string on the canvas somehow. We haven't really done that yet, and it isn't necessarily the most straightforward thing to do.

The shapes that make up the letters we see on the screen are actually quite complex (especially compared to triangles and circles), and drawing them all by hand is probably not what you had in mind for a nice relaxing game-writing session. Thankfully, we have **fonts**. A font is just a library of instructions for how the computer should draw each character that you might want to display. "Scalable" fonts are usually represented as "splines" (mathematical descriptions of curves). Many modern fonts are scalable, which means we can make them any size we like and they'll still look reasonably good. That's good news—we don't have to worry about size affecting how pretty our characters are. The browser has a lot of built-in fonts, so we will just use the default one for "sans-serif" characters.

Modern browsers have generic font names like "sans-serif" available, which is really helpful for making sure your web site works in multiple environments. That said, it is often the case that what one browser on one operating system thinks of as "sans-serif" can be very different than another. They usually look fine, but if you want them to look *exactly the way you see them, everywhere*, then you have to pick a more specific font, and that font may or may not be installed on the target system.

To solve this, you use web fonts^a, and that's a larger discussion for another time.

^ahttps://developer.mozilla.org/en-US/docs/Learn/CSS/Styling_text/Web_fonts

Text is actually really interesting to draw. When we draw text, we draw it into a **bounding box**. We specify what we want to draw and where we want to draw it, along with some other properties that indicate things like how big it is and which corner of the box we're talking about. Since I just decided that the clock will be displayed at the top of the screen, and you can't argue with a textbook, we will use the "hanging" baseline, which means that the text is drawn below and to the right of the coordinates we specify: the box's upper left corner is given.

Remember how we drew paths and filled them to make a triangle? We will do something

similar here: we will set the properties on a context object, then we will fill our text shapes. Here is a new animation specification that draws the text “Clock!” in the upper left corner of the canvas. Give it a try!

:javascript:

```
var animation = animate({
  move: function(dt) {
    console.log("move", dt, animation.elapsed());
  },
  draw: function() {
    ctx.clearRect(0, 0, canvas.width, canvas.height);
    ctx.font = '20px sans-serif';
    ctx.textBaseline = 'hanging';
    ctx.fillStyle = 'black';
    ctx.fillText('Clock!', 2, 2);
  },
});
```

That’s a good start! We have the word “Clock!” drawn in our canvas, but, of course, it isn’t really a clock—it doesn’t reflect the number of seconds elapsed. Maybe we should put the seconds elapsed into the `fillText` function instead, like this:

:javascript:

```
ctx.fillText(animation.elapsed(), 2, 2);
```

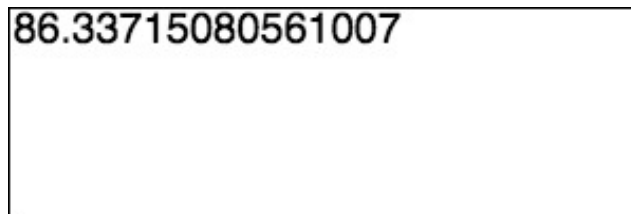


Figure 12.1: Clock using `animation.elapsed()`.

Recall that the “floor” operator (`Math.floor`) rounds to the nearest integer, always downward. A similar function is the “ceiling” operator (`Math.ceil`), which always rounds upward to the nearest integer.

What did you notice about this immediately? It’s a bit ... distracting. There are an awful lot of digits being displayed there. Let’s just make it show integer seconds instead of more accurate but distracting fractions of seconds by using `Math.floor` like this:

:javascript:

```
ctx.fillText(Math.floor(animation.elapsed()), 2, 2);
```

That’s much better. We now have seconds ticking away in our game window. Hooray! The full listing of “program.html” is here. Note that we snuck in a “keydown” listener to pause

Figure 12.2: Clock using `Math.floor(animation.elapsed())`.

when the space key is pressed, as well. It makes use of `animation.togglePaused` just like our click handler does.

:html:

```
<canvas id="drawing" width="300" height="300"></canvas>
<script src="animate.js"></script>
<script>
var canvas = document.getElementById('drawing'),
    ctx = canvas.getContext('2d');

var animation = animate({
  move: function(dt) {
    console.log("move", dt, animation.elapsed());
  },
  draw: function() {
    ctx.clearRect(0, 0, canvas.width, canvas.height);
    ctx.font = '20px sans-serif';
    ctx.textBaseline = 'hanging';
    ctx.fillStyle = 'black';
    ctx.fillText(Math.floor(animation.elapsed()), 2, 2);
  },
});

canvas.addEventListener('click', animation.togglePaused);
document.addEventListener('keydown', function(event) {
  if (event.key === ' ') {
    animation.togglePaused();
  }
});
</script>
```

There are a number of things we could do with this. If you leave this ticking long enough, you will notice that it always shows seconds. It will display “92” instead of “1:32” to mean “1 minute 32 seconds”. For now let’s not worry about that—we will just leave it as a number of seconds. And, since the game is all about survival, that can double as our score, as well.

A Good Paddling

Now we have a clock. Our animation routine works, clicking or pressing the space bar to pause works, and we know where our state is and how to use it to draw something. Let's move on to drawing a paddle.

A paddle is just a short, wide rectangle that moves side to side in hopes of catching a ball before it falls off the screen. What data do we need to keep track of for the paddle? Basically, we need its position and its size, and some parts of those will be constant.

Let's suppose we have `x` and `y` coordinates for our paddle. The first question is this: what do those coordinates really mean? Do they represent the lower left corner of the paddle? The dead center? What meaning will be the most useful?

Since we will be drawing the paddle at the bottom of our screen, and the ball will need to strike the top of it, we will be testing ball coordinates against the top of the paddle quite often. It's therefore probably most convenient to have the `y` coordinate represent the top. The `x` coordinate might as well represent the left side, so `(x, y)` will be the top left corner of the paddle.

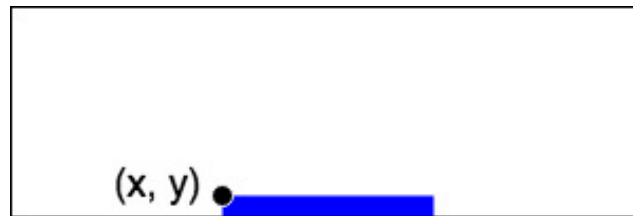


Figure 12.3: Paddle Coordinates

What is our paddle state, then? It is two coordinates (horizontal and vertical) and two sizes (horizontal and vertical). We can represent this in our state variables like this:

:javascript:

```
var PADDLE_HEIGHT = 10,
    PADDLE_Y = canvas.height - PADDLE_HEIGHT;

var paddleWidth = 100,
    paddleX = (canvas.width - paddleWidth) / 2;
```

What did we just do there? We first defined the size of the paddle, then we defined the `y` coordinate of the top left corner: the height of the canvas minus the height of the paddle. These are in all caps because they are constants—we don't expect to ever need to change them.

Then we created variables that are meant to change: the width of the paddle and the location of the paddle. We won't change the width just yet, but we will do so eventually as the game

progresses, just to make it an interesting challenge. The x coordinate of the paddle initially places the rectangle centered in the canvas. The reason it works is this little equivalence:

$$\frac{w_c}{2} - \frac{w_p}{2} = \frac{w_c - w_p}{2}$$

where w_c is the canvas width, and w_p is the paddle width. Basically, if you find the center of the canvas, then subtract half the paddle width, you find out where the left side of the paddle should be when it's in the middle. But, that's the same as subtracting the entire paddle width from the canvas width, then taking half of that.

Math.

With that state, how should we draw the paddle? We need to choose a color, which will be blue on these pages, but you should feel free to change it to your liking. The paddle will be a simple rectangle, and we already know how to draw those. We just need to provide the upper left corner and the width and height as is familiar:

:javascript:

```
ctx.fillStyle = "blue";
ctx.fillRect(paddleX, PADDLE_Y, paddleWidth, PADDLE_HEIGHT);
```

Let's give this a try! Add the new state where it belongs, and add the drawing functions into the `draw` closure. When you load the page and start the game, the paddle should be right where we put it, at the bottom of the screen. Neat!

It doesn't really do anything, though. For that, we will need to add some event listeners.

Moving the Paddle

Here we need to ask some more interesting questions, like "How do we want to control this thing?" How about this: when we are holding down one of the left or right arrow keys, the paddle will move, and when we aren't holding down any of those keys, it will stop.

We can tell whether a key (say, the "a" key) is currently being pressed by keeping track of its "keydown" event as well as its "keyup" event, something like this:

:javascript:

```
var aDown = false;

document.addEventListener("keydown", function(event) {
  if (event.key === "a") {
    aDown = true;
  }
});
document.addEventListener("keyup", function(event) {
```

```

    if (event.key === "a") {
      aDown = false;
    }
  });

```

Do you see what we did there? We have a variable `aDown` that tells us whether the key is currently pressed down or not. We change it to `true` when we get an appropriate “keydown” event, and we change it back to `false` when we get an appropriate “keyup” event. We have basically converted events into states. That’s a useful technique.

This way, if we have code that wants to behave differently based on whether the “a” key is currently being pressed, we can just have it check the value of `aDown`. This idea is something we can use for our paddle, since we just want it to move differently based on which arrow key, if any, is currently down.

This was all well and good for the letter “a”, but how do we figure out whether an arrow key is what caused an event? They don’t exactly make any characters appear, so they won’t have single-character names. Let’s go to the console and figure out what they do:

:console:

```

> function logEventKey(e) { console.log(e.key) }
> document.addEventListener("keydown", logEventKey);

```

After typing that in (you can see what it does), do the following:

- Click on the document window (outside of the developer tools). This focuses the document so it will receive your key events.
- Start typing. You will see what you type appear in the console. Try typing the arrow keys and see what appears.

Here’s what might happen after you

- Add the event listener above into the console,
- Click on the main window, and
- Start pressing some keys:

:console:

```

a
b
ArrowLeft
ArrowRight
ArrowDown
Tab

```

In that example, I typed the ‘a’ key, the ‘b’ key, some arrow keys, and the ‘tab’ key. You can see what the event model calls those keys. It turns out that the left and right arrows are called “ArrowLeft” and “ArrowRight”, which is useful to know.

To do this next part, we can either register a new event listener for key events, or we can add to the event listener that we already have. Either way is fine, but let's just add to the current "keydown" event listener to see what that looks like. Remember, the current listener looks like this:

:javascript:

```
document.addEventListener('keydown', function(event) {
  if (event.key === ' ') {
    animation.togglePaused();
  }
});
```

To add the ability to keep track of the left and right arrow keys, we will need a couple of variables that hold their current state, like this:

:javascript:

```
var keyLeft = false,
    keyRight = false;
```

Then we will start by upgrading our "keydown" handler like this:

:javascript:

```
document.addEventListener('keydown', function(event) {
  switch (event.key) {
    case ' ':
      animation.togglePaused();
      break;
    case 'ArrowLeft':
      keyLeft = true;
      break;
    case 'ArrowRight':
      keyRight = true;
      break;
  }
});
```

Remember our old friend the `switch` statement? We could have written this using `else if` statements, but it's good to get practice with `switch`, too. When we get a "keydown" event, we handle different keys differently (and leave all the rest alone). If we get a space key, we toggle the paused state as before, but now if we get a left or a right arrow key, we set it as being down. All that remains is to unset it when the key is released. For that we use "keyup":

:javascript:

```
document.addEventListener("keyup", function(event) {
  switch (event.key) {
    case 'ArrowLeft':
      keyLeft = false;
      break;
    case 'ArrowRight':
```



```

        keyRight = false;
        break;
    }
});

```

This one is very similar to the “keydown” event, but it doesn’t handle the space bar because it really doesn’t need to. If you want to see what’s happening with the `keyLeft` and `keyRight` variables during the animation, you can add some logging statements to the `move` or `draw` function, like this:

:javascript:

```

console.log('keyLeft', keyLeft);
console.log('keyRight', keyRight);

```

Of course, that quickly logs a huge amount of stuff quickly and doesn’t really do anything interesting, but you should at least see the key states change from `false` to `true` and back when you press the arrow keys while the game is running. Note, however, that you have to be careful to click in the game window first: if the console is the last thing you were interacting with, then the key events won’t fire: the handlers are registered on the document, so the document needs to be what we’re interacting with in order to capture the events.

Now we’re ready to make the paddle move. How fast should it go? Let’s say it should be able to cross the entire canvas in one second, so the speed will be `canvas.width - paddleWidth` because speed is going to be in pixels per second, and moving from the left side to the right means the left corner changes from 0 to `canvas.width - paddleWidth` (when the right side touches the right wall).

We will add the speed to our variables and call it `paddleSpeed`, like this (we could have named it like a constant, as well—sometimes it isn’t clear which way to go when you first start on an idea, and that’s okay):

:javascript:

```

var paddleWidth = 100,
    paddleSpeed = canvas.width - paddle.Width,
    paddleX = (canvas.width - paddleWidth) / 2;

```

Then we can change our `move` function to actually move the paddle (note that this is just a snippet of the `animate` call’s argument, which is much longer than this):

:javascript:

```

move: function(dt) {
    if (keyRight) {
        paddleX += paddleSpeed * dt;
    }
    if (keyLeft) {
        paddleX -= paddleSpeed * dt;
    }
}

```

```
},
```

Awesome. Let's give that a try. What happens when you start the game and press the arrow keys now? It's alive! It moves in response to your key presses!

It also doesn't stop when you reach the border. Uh-oh. That is probably not what we want. How should we fix that? Well, when we calculate the new `paddleX`, we shouldn't let it go out of bounds. Since `paddleX` is at the left edge of the paddle, that means it shouldn't be less than 0, and it shouldn't be more than `canvas.width - paddleWidth`:

```
:javascript:
```

```
move: function(dt) {
  if (keyRight) {
    paddleX += paddleSpeed * dt;
    if (paddleX > canvas.width - paddleWidth) {
      paddleX = canvas.width - paddleWidth;
    }
  }
  if (keyLeft) {
    paddleX -= paddleSpeed * dt;
    if (paddleX < 0) {
      paddleX = 0;
    }
  }
},
```

Now that's better. The paddle stops when it reaches the right wall or the left wall, depending on which way you are pushing it.

Something To Shepherd

We have a moving paddle, and we have a clock. What we lack is a ball to keep inside the canvas walls. We just learned how to animate one of those recently, though, so it should be pretty easy to add it back in. We will have ours start in the center of the canvas, and we will use the `fillCircle` function from our "animate.js" library to draw the ball.

Since we are adding yet another moving thing to our game, we need to keep track of all of its state. Here are the things we kept track of in the past, with different names to keep them separate from all of the state we are keeping track of now:

```
:javascript:
```

```
var BALL_RADIUS = 10,
    ballX = canvas.width / 2,
    ballY = canvas.height / 2,
    ballVelX = canvas.width / 2,
    ballVelY = canvas.height / 2;
```

The velocity variables could be set to anything, really, so this is just an experiment to see if we

like how fast things move with the velocity components set to half the canvas width and height (per second). At least we know the paddle can move fast enough!

We have state for the ball, so all we need to do is move it and draw it. Do you remember what you need to do in the `move` and `draw` functions to make that work? Drawing should be pretty easy, actually; just draw a ball with radius `BALL_RADIUS` at the coordinates `ballX` and `ballY`. That's a one-line call inside of `draw` somewhere:

:javascript:

```
fillCircle(ctx, ballX, ballY, BALL_RADIUS);
```

To move it, you just need to add two lines to the `move` function:

:javascript:

```
ballX += ballVelX * dt;
ballY += ballVelY * dt;
```

If you run the program with these changes, it works as before, but now the ball appears in the center of the screen and moves down to the right. It keeps on going, too. Forever.

The paddle still moves, and the clock still runs, so that's all still good. What we are missing is the test for the walls that reverses the velocity components to make the ball bounce. We had that test in our previous animation's `move` function, and it looked like this (but with different variable names):

:javascript:

```
if (ballX - BALL_RADIUS <= 0) {
  ballVelX = Math.abs(ballVelX);
} else if (ballX + BALL_RADIUS >= canvas.width) {
  ballVelX = -Math.abs(ballVelX);
}
if (ballY - BALL_RADIUS <= 0) {
  ballVelY = Math.abs(ballVelY);
} else if (ballY + BALL_RADIUS >= canvas.height) {
  ballVelY = -Math.abs(ballVelY);
}
ballX += ballVelX * dt;
ballY += ballVelY * dt;
```

Quick quiz: why don't we use `else if` between the horizontal and vertical tests? In other words, why do we check both horizontal and vertical directions every time? (HINT: it's a literal "corner case". There are metaphorical ones, too.)

Given that our playing field is a perfect square (unless you changed it, which you are perfectly welcome to do), the ball just bounces back and forth between two corners. We can move our paddle while that is going on, but it doesn't really do anything. It just moves around while the ball does its thing. That is completely expected, since we haven't programmed any interaction between the ball and the paddle. We're tackling this game one step at a time, and it's good to

make progress wherever we can.

Failure Is An Option

With the ball bouncing off the walls and our paddle moving around, we are ready to introduce the potential for a “Game Over” scenario, where the ball goes off the bottom of the screen and we lose. Right now it’s impossible to lose because the ball stays in the field all the time. It is also impossible to win, since there is no score and no way to make it go up. Opposition in all things. Let’s start with failure and move on to success from there.

How do we make the ball go off the bottom of the screen? That’s pretty easy: just remove the test for `ballY + BALL_RADIUS >= canvas.height`. Then it will be free to fly right off the bottom. Again in the move function, that leaves us with this update code:

:javascript:

```
if (ballX - BALL_RADIUS <= 0) {
    ballVelX = Math.abs(ballVelX);
} else if (ballX + BALL_RADIUS >= canvas.width) {
    ballVelX = -Math.abs(ballVelX);
}
if (ballY - BALL_RADIUS <= 0) {
    ballVelY = Math.abs(ballVelY);
}
ballX += ballVelX * dt;
ballY += ballVelY * dt;
```

If we run this now, the ball will go off the bottom when it reaches it, just like we told it to. But the game doesn’t stop when that happens. Also, it isn’t really the bottom of the screen that we need to be concerned about, but the ball passing below the top of the paddle. Once it has done that, there is no redeeming it, so the game is over. Let’s alter our checks a bit to stop the game once the ball goes below the top of the paddle, remembering that we can stop the animation by providing a `done` function that returns `true` when the game is over.

Here is the `done` section we want to add to our `animate` function call:

:javascript:

```
done: function() {
    if (ballY + BALL_RADIUS > PADDLE_Y) {
        alert("Game Over!");
        return true;
    }
    return false;
},
```

We could have made this simpler (e.g., just `done return ballY + BALL_RADIUS < PADDLE_Y;`), but it’s nice to be able to tell the user that the game is over. Note the pattern of putting an exceptional case first, then the default case last. Since the `if` statement issues a

`return`, the function will immediately be finished at that point and its result will be `true`. But, if the `if` body doesn't run, the final `return false;` will be executed. This is a useful pattern to follow because it aids readability. A person reading this function can read and understand the first part, then forget about it while focusing on what comes next. This is merely a small example of something that starts making a big difference in larger functions.

Where are we now? Now the game stops once the ball reaches the bottom, but it bounces if it hits something else first. It currently doesn't hit anything else first, unless you have exercised your freedom of expression and made the game taller than it is wide, which you are of course completely welcome to do. Go on, I'll wait.

Success Is Better

There is currently no way to actually hit the ball, since we just test for it passing the top of the paddle and we quit. That's really no fun, is it? It would be nice if we could actually hit the ball with the paddle, but we currently don't check for that. Let's do that now.

This will go into the `move` function, just like all of the rest of our collision tests. Our test for vertical collision will change to something like this:

:javascript:

```
if (ballY - BALL_RADIUS <= 0) {
  ballVelY = Math.abs(ballVelY);
} else if (ballY + BALL_RADIUS >= PADDLE_Y) {
  if (ballX >= paddleX && ballX <= paddleX + paddleWidth) {
    console.log('Hit');
    ballVelY = -Math.abs(ballVelY);
  }
}
```

See what we did there? We added the test for the top of the paddle back in, but this time we only reverse velocity if we're actually hitting the paddle, meaning that the center of the ball is within the left and right paddle edges when it touches the top of it.

If you think about this carefully, though, you will notice that there is a problem, and you'll see it in the console if you try running it. A hit is registered, but so is "Game Over", because the test for a hit is so similar to the test for a failed game! Oops. To fix that, let's have the `move` function be completely in charge of indicating whether we hit or failed, and we will store that in a variable that `done` simply returns (don't forget to add that variable to our state!).

You have probably realized by now that there are many ways of going about this. We could, for example, change the code in `done` to ensure that it only triggers when the ball goes below the paddle without hitting it, or while still traveling downward. You should feel free to try that out. Here we do it with a variable to avoid code repetition, but other ways work fine.

With that change, things look like this:

:javascript:

```
// ... somewhere above ...
var gameOver = false;

// ... somewhere in the move function ...

if (ballY - BALL_RADIUS <= 0) {
  ballVelY = Math.abs(ballVelY);
} else if (ballY + BALL_RADIUS >= PADDLE_Y) {
  if (ballX >= paddleX && ballX <= paddleX + paddleWidth) {
    ballVelY = -Math.abs(ballVelY);
  } else {
    gameOver = true;
  }
}
```

Let's go over what we've done. First, as before, we check that the ball hasn't hit the ceiling, and bounce if it has. If it hasn't, we check whether it has gone below the top edge of the paddle. This is all as it was before. But now, instead of just terminating the game unconditionally, we do some more checks.

The first thing we check is whether the ball is to the right of `paddleX`, which is the left edge of the paddle. If it is, we check that it is also to the left of `paddleX + paddleWidth`. If it is, then the ball has been hit by the paddle. Note that we don't bother with `BALL_RADIUS` here, because if the center of the bottom of the ball is within the bounds of the paddle, that's a hit. Putting the radius into that condition would make it seem like we should have succeeded when we fail, and that can be frustrating. Go ahead and try it, though; it's good to see what these things feel like and to experiment with different things to make the game interesting.

Finally, if those checks fail, it means we missed the paddle and the game is over, so we set `gameOver = true`. To make the game actually fail properly now, we will need to update the `done` function to notify the user of the failure, then return the value of `gameOver`, like this:

:javascript:

```
done: function() {
  if (gameOver) {
    alert("Game over!");
  }
  return gameOver;
},
```

Now, if you start the game, you will notice that you can actually play it! Congratulations, that's a huge step! The full listing is below. You should be pretty happy with this, because you can look at this listing and understand exactly what each part of it does, and how it all fits together. If you look back over this chapter, you will also see a very important technique for building

software: we figured out what we wanted to do, then we put it together a little bit at a time, making sure to have small successes along the way. That is not always possible, but it often is, and it's important to be able to cut an idea into pieces and build one at a time.

You will also notice that we didn't get everything right. We did some things, realized they needed adjustments, and took a couple of steps backward to finally move all the way forward. That is part of the spirit of programming; it's a world where you can experiment, learn, adjust, and experiment again until you understand enough to be completely successful.

The full listing comes up next, including our additions to "animate.js" to include configuration parameters, elapsed time, and pause toggling.

While this is the longest listing yet, you can still follow everything that is happening within it. The state variables are all at the top, the `move` function handles moving the paddle and the ball (as well as determining when the game is over), and the `draw` function just sticks things where they go. The `done` function is very straightforward as well, merely returning the state of the `gameOver` variable set inside of `move`.

All told, it's actually a small amount of code for a relatively complete game. The exercises will explore adding a couple more features to it, so make sure you're comfortable with all of the pieces here and then dive in!

Listings

animate.js

:javascript:

```
function animate(config) {
  var move = config.move || function() {},
      draw = config.draw || function() {},
      done = config.done || function() {};

  var lastTime = 0,
      frame = null,
      elapsed = 0,
      EXPECTED_DT = 1 / 60;

  function tick(t) {
    if (done()) {
      frame = null;
      return;
    }
    frame = requestAnimationFrame(tick);

    var dt = (t - lastTime) / 1000;
    if (dt > 4 * EXPECTED_DT) {
      dt = EXPECTED_DT;
    }
  }
}
```

```

    } else if (dt < 0) {
        dt = 0;
    }
    elapsed += dt;
    lastTime = t;

    draw();
    move(dt);
}

frame = requestAnimationFrame(tick);

return {
    running: function() {
        return !!frame;
    },
    elapsed: function() {
        return elapsed;
    },
    pause: function() {
        if (frame) {
            cancelAnimationFrame(frame);
            frame = null;
        }
    },
    start: function() {
        if (!frame) {
            frame = requestAnimationFrame(tick);
        }
    },
    togglePaused: function() {
        if (frame) {
            cancelAnimationFrame(frame);
            frame = null;
        } else {
            frame = requestAnimationFrame(tick);
        }
    },
};
}

function fillCircle(ctx, x, y, r) {
    ctx.beginPath();
    ctx.arc(x, y, r, 0, 2*Math.PI);
    ctx.fill();
}

```

program.html

:html:


```

<canvas id="drawing" width="300" height="300"></canvas>
<script src="animate.js"></script>
<script>
var canvas = document.getElementById('drawing'),
    ctx = canvas.getContext('2d');

var PADDLE_HEIGHT = 10,
    PADDLE_Y = canvas.height - PADDLE_HEIGHT;

var paddleWidth = 100,
    paddleSpeed = canvas.width - paddleWidth,
    paddleX = (canvas.width - paddleWidth) / 2;

var BALL_RADIUS = 10,
    ballX = canvas.width / 2,
    ballY = canvas.height / 2,
    ballVelX = canvas.width / 2,
    ballVelY = canvas.height / 2;

var keyLeft = false;
    keyRight = false;

var gameOver = false;

var animation = animate({
  move: function(dt) {
    // Move the paddle.
    if (keyRight) {
      paddleX += paddleSpeed * dt;
      if (paddleX > canvas.width - paddleWidth) {
        paddleX = canvas.width - paddleWidth;
      }
    }
    if (keyLeft) {
      paddleX -= paddleSpeed * dt;
      if (paddleX < 0) {
        paddleX = 0;
      }
    }

    // Move the ball.
    if (ballX - BALL_RADIUS <= 0) {
      ballVelX = Math.abs(ballVelX);
    } else if (ballX + BALL_RADIUS >= canvas.width) {
      ballVelX = -Math.abs(ballVelX);
    }
    if (ballY - BALL_RADIUS <= 0) {
      ballVelY = Math.abs(ballVelY);
    } else if (ballY + BALL_RADIUS >= PADDLE_Y) {
      if (ballX >= paddleX &&

```

```

        ballX <= paddleX + paddleWidth) {
            ballVelY = -Math.abs(ballVelY);
        } else {
            gameOver = true;
        }
    }
    ballX += ballVelX * dt;
    ballY += ballVelY * dt;
},
draw: function() {
    ctx.clearRect(0, 0, canvas.width, canvas.height);
    ctx.font = '20px sans-serif';
    ctx.textBaseline = 'hanging';
    ctx.fillStyle = 'black';
    ctx.fillText(Math.floor(animation.elapsed()), 2, 2);

    ctx.fillStyle = "blue";
    ctx.fillRect(paddleX, PADDLE_Y,
                 paddleWidth, PADDLE_HEIGHT);

    fillCircle(ctx, ballX, ballY, BALL_RADIUS);
},
done: function() {
    if (gameOver) {
        alert("Game Over!");
    }
    return gameOver;
},
});

canvas.addEventListener('click', animation.togglePaused);
document.addEventListener('keydown', function(event) {
    switch (event.key) {
        case ' ':
            animation.togglePaused();
            break;
        case 'ArrowLeft':
            keyLeft = true;
            break;
        case 'ArrowRight':
            keyRight = true;
            break;
    }
});

document.addEventListener('keyup', function(event) {
    switch (event.key) {
        case 'ArrowLeft':
            keyLeft = false;
            break;
        case 'ArrowRight':

```

```
        keyRight = false;
        break;
    }
});
</script>
```

Exercises

Exercise 12-1: Short Circuit Logic Operators

Solution on page 467

For all expressions below, indicate whether anything is written to the console.

- `var a = false || console.log("hi")`
- `var b = true || console.log("hi")`
- `var c = false && console.log("hi")`
- `var d = true && console.log("hi")`

Note that we are only interested in whether "hi" gets written, not in the actual value of the expressions.

Exercise 12-2: Logic Operator Evaluation

Solution on page 467

For each expression below, predict what it evaluates to:

- `1 || undefined`
- `1 && undefined`
- `0 || 1`
- `undefined || false`
- `true && 6`

Exercise 12-3: Lab Part I: Adding a Score and Increasing Difficulty

Solution on page 468

For this lab you will add some improvements to the paddle game that we hinted at while creating the first version. These are

- Add a score (at upper right) that increments for each successful hit,
- Make the paddle slightly smaller (multiply its width by 0.99) with each successful hit,

In other words, we will make this more like an actual game, one that keeps score and gets a little harder and more interesting to play as time goes on.

Note that every context has a `textAlign` property that you can set to "left", "right", or "center" and it affects how the `fillText` method decides where to draw your text. For

displaying time, we used the default value (because we didn't specify it at all), which is "left". Now that we will be changing it, we will need to specify it for time ("left", since it's on the left side), and then specify a different one for the score ("right", since it will be on the right side).

Exercise 12-4: Lab Part II: Adding Randomness and Speed Changes

Solution on page 471

To continue the lab, we will make the game more difficult by adding some randomness to how the ball bounces off of the paddle, and by making it go slightly faster every time it does. This requires a bit more vector math, so let's start with some hints about how that should work.

- Make the ball a bit faster with each successful hit, and
- Add randomness to the movement of the ball so it isn't so predictable,

Previously, we just reversed x and y velocity every time a bounce occurred. Among other things, this guaranteed that the overall speed of the ball never changes. Simply reversing one of the components of the velocity doesn't change its speed, only its direction. That is nice, but we're about to fiddle with things on a deeper level now, so we have to be extra careful about maintaining speed.

How do we know that reversing direction doesn't change speed (aside from intuition, I mean)? We know that because speed is the magnitude of the velocity vector, and that's calculated like this:

$$s = \sqrt{v_x^2 + v_y^2}$$

Or, in code, like this:

:javascript:

```
speed = Math.sqrt(ballVelX*ballVelX + ballVelY*ballVelY);
```

Do you see why the speed doesn't change when we change the sign of `ballVelX` or `ballVelY`? It's because they're squared, and that always produces a positive value.

Our goal is to change the direction of the ball in more ways than just "bouncing" it, and we want to slowly increase speed with every bounce. That formula we just used is going to come in handy. Here are some hints for how to go about making this change:

Use `Math.random` to *slightly* alter the `ballVelX` value when bouncing off of the paddle:

We already change `ballVelY` to be its negative when that happens, but now we will *also* change `ballVelX`, just a bit, just to keep things interesting. It should increase *or* decrease the value by a small amount, say up to 10%. How would you accomplish this using `Math.random`?

The first thing to do is to break this idea down into small pieces. How much will we add to (or subtract from) `ballVelX`? It's up to 10% of its current value, let's start with getting a random percentage from 0 to 10: `Math.random() * 0.1`. That will do it.

Now that we can compute a random percentage up to 10%, how do we apply that to `ballVelX` to get a velocity delta (change)? We multiply. The amount we will want to change `ballVelX` by is thus `ballVelX * 0.1 * Math.random()`.

Finally, we want to either add or subtract that value. There are a couple of ways to approach this. You can flip a coin (use `Math.random() > 0.5` for this) to decide whether to add or subtract, or you can generate a number that is randomly positive or negative and always add. Your choice (there are others, but let's keep it simple).

There are some problems with the idea presented here, and you may notice them as you play for a while. Because the randomness is multiplicative, it's possible (likely, even) to get into a state where the x velocity just keeps getting smaller over time: it's harder to increase its magnitude (because we do so by a percentage) than it is to decrease it: 10% added to a small number doesn't make up for the 10% previously subtracted from a larger one. There are better ways, but this was simple to explain and implement. You may want to try other things.

Keep track of speed, and make sure it only changes *exactly* how you want it to:

Speed can be a tricky thing. If you don't keep a careful eye on it, you could easily end up in a situation where random changes to direction result in a very large increase in overall speed, and that's not intended: we want to *control* speed in this game, not let it go haywire. That means we will need to make absolutely sure that it doesn't get messed up by our introduction of random motion.

Here's the basic idea. Right when a paddle hit is detected, do this:

- Calculate the speed of the ball and remember it (we will call this `originalSpeed`),
- Change direction and add randomness as above,
- Calculate the new speed (we will call this `newSpeed`),
- Scale `ballVelX` and `ballVelY` to make the new speed the same as the original speed.

Most of this is pretty straightforward using the speed formula. If you can calculate speed using the formula given above, then you can do most of the steps in this process. The one that might be a little new is figuring out how to scale things once you have `originalSpeed` and `newSpeed`.

The answer can be found with a little algebra, which you are encouraged to do. This, however, is not an algebra course, so here's the answer: scale the velocity components by `originalSpeed / newSpeed` after changing them. Intuitively, if the new speed is smaller than the original, this makes it bigger by the right amount, and if it is bigger, it makes it

smaller. What you end up with is the same speed you started with, but a different direction. You are “renormalizing” the speed.

You don’t want it to just stay the same, though. You want it to get slightly faster in a controlled way. Well, that should be pretty simple since we have the original speed. Instead of scaling everything by `originalSpeed / newSpeed`, we will scale it by that times some factor that makes it a bit quicker. For example, you could imagine multiplying that whole thing by `1.01` to add 1% to the speed every time, making the new scale factor `1.01 * originalSpeed / newSpeed`.

Chapter 13

Snakes On a Page

We have now created a complete, playable game. That's an accomplishment, and we did it using only basic from-scratch concepts in JavaScript and HTML. Along the way we have covered a number of interesting and useful programming ideas and techniques that apply to just about any software you may want to write. That's pretty great! You are getting to know JavaScript pretty well at this point, certainly well enough to do interesting things with it.

The language has grown a lot, starting with its first *massive* upgrade in 2015, and we haven't really covered any of the new features introduced then or in the intervening years, but the original language base is plenty powerful as well as being easier to reason about when first learning how to program. This book thus intentionally avoids some of the newer language features even though many of them are quite nice, because it's just too much for an introduction to the art and practice of programming. If what you have seen is exciting, you will definitely want to dive deeper into the language at some point. Fortunately, this text will provide a solid foundation on which to build and there are many great resources for moving on from here.

That the language has changed—even dramatically— isn't just true for JavaScript, by the way. All languages worth using tend to evolve over time. Some improve, some don't, but exceedingly few really remain both stagnant and relevant for long. Learning is a lifelong pursuit when you do this sort of thing as a career. That's one of my favorite things about it.

For now, however, let's pause our climb and take some time to practice what we have learned. There will be just a couple of new things in here, particularly where canvas coordinates are concerned, but no new fundamental language features. This chapter is all about solidifying your grasp of everything learned thus far.

We're going to use our `animate` abstraction to create a different kind of game this time: the game of snakes. In this game, food (shown green in the example image) randomly appears on the canvas, and your snake starts out as a very tiny critter. Every time it eats, it gets longer and faster. The goal is to get the snake to be as long as possible without running into itself or

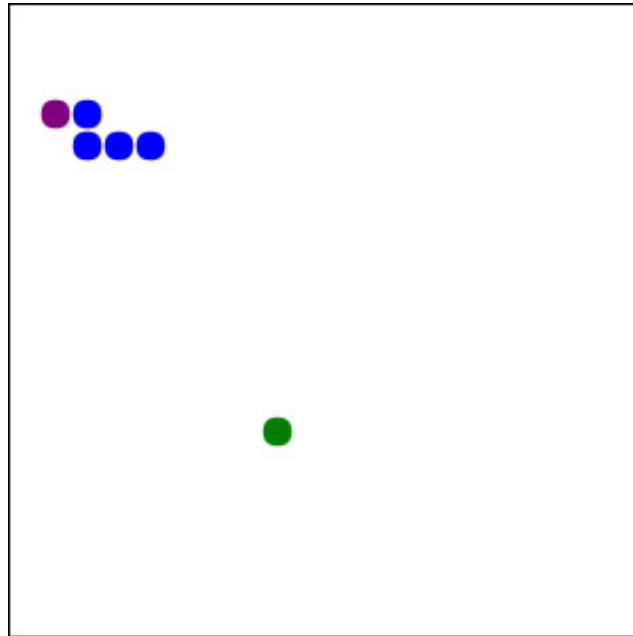


Figure 13.1: The Classic Game of Snakes

the wall.

Game Scaffolding

To begin, create a new folder and copy “animate.js” from your previous project into it (or just make a new “snakes.html” file wherever “animate.js” is). You get all of that work for free, now! That’s the beauty of a good library: once it’s done (and has its bugs worked out), you can just use it over and over again.

In your “snakes.html” file in this new folder, you will put the familiar scaffolding for the game: a canvas, the script tags for “animate.js”, and some basic animation stuff, consisting of empty functions for animating and event listeners for pausing. This should all be very comfortable at this point.

:html:

```
<canvas id="drawing" width="300" height="300" style="border: 1px solid black"></canvas>
<script src="animate.js"></script>
<script>
var canvas = document.getElementById('drawing'),
    ctx = canvas.getContext('2d');

// State variables typically go somewhere around here.

var animation = animate({
```



```

    move: function(dt) {
        // Change state based on time and events.
    },
    draw: function() {
        // Clear the canvas, draw stuff.
    },
    done: function() {
        // Quit when we lose.
    },
  });

// We pretty much always want a pause feature:
canvas.addEventListener('click', animation.togglePaused);
document.addEventListener('keydown', function(event) {
  if (event.key === ' ') {
    animation.togglePaused();
  }
});
</script>

```

As you can see, even with our `animate` abstraction, there is some repetition here. We aren't going to spend any time fixing that, since it's repetition in things that won't always be the same (all games won't use the space key for pausing, for example), but when you do see a repeated pattern and can see how it might be meaningfully reused in other circumstances, that can be good opportunity for refactoring. We did that earlier with `togglePaused`, and now our skeleton is shorter and even clearer than before.

Of course, this is just the minimal skeleton that you need to get through this chapter. You should always feel free to play around and add your own features.

Designing the Game

First things first: what state do we need to keep track of for a snake game?

In the game of snakes we have a current position, but that isn't really enough. We actually have a whole snake's worth of current positions! That sounds like an array, since they're all in order and we have to move things appropriately, so we know we will have an array of position coordinates. That's pretty easy.

We will also need to know which direction the snake's head is moving in. Each time an arrow key is pressed, we may change direction; but if nothing is pressed, the head keeps moving in the same direction. We need to keep track of that.

There will always be a bit of food around for us to eat, so we will also need to keep track of where the food is. There is only one bit of food around at a time, so we can just store that as a single set of position coordinates.

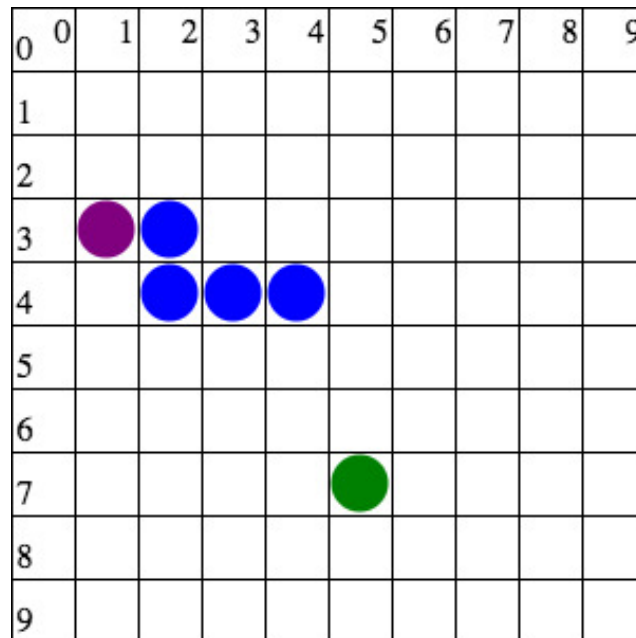


Figure 13.2: Food: [5,7], Snake: [[1,3], [2,3], [2,4], [3,4], [4,4]]

And that's basically all! There are some tricky parts, however, especially since the snake moves very differently from our ball or paddle from the previous chapter. We will get to that when the time comes, and it will be one of the biggest changes in our approach so far. The `animate` function we created can handle it, though.

Positions Are Picky

In the state section, we will set up an initial snake and some food. Before we do that, however, we kind of need to know what we mean by “position”. We can't just mean “pixel locations” anymore, because that's not how the snake game works.

The snake is sitting on a grid with (relatively) large cells, and can only move up, down, left, or right, one whole cell at a time. The figure shows an example of a valid position (solid) and an invalid position (outlined). Because each cell contains numerous pixels, we won't be moving things based on pixel units; each pixel is only a fraction of a cell. Instead, we need to position things using integer cell units.

Unit Conversions

We might as well pick a grid size. Let's make it 20 cells wide and 20 cells high, but let's do so in a way that is easy to change using suitable constants. From the grid size we can easily compute the size of each cell in pixels, which will tell us how to move our snake and where to

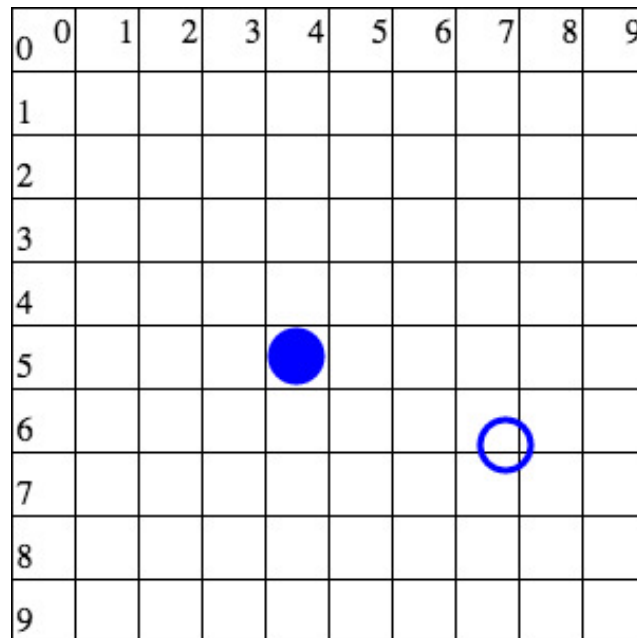


Figure 13.3: The outlined circle wouldn't be allowed.

draw its body pieces. Our position will be an index into this grid, so it can have x and y values from 0 to 19 (inclusive).

In order to go from “snake position” to “pixel location”, we’re going to need to do some unit conversion. We will start by computing how big the cells are based on the canvas size and the number of items in each direction:

:javascript:

```
var CELLS = 20, // Cells on one side.
    CELL_PIXELS = canvas.width / CELLS;
```

Note that some folks like to use really descriptive names for constants. That’s generally a good idea, and we have sort of skimmed here. `CELLS` is not really the number of cells in the grid, but the number of cells on a side. Perhaps we should have called it `CELLS_ON_A_SIDE`. But this is actually a short program, and in the context of this code, the name `CELLS` is unambiguous, so we’re going with that. A similar argument can be made for `CELL_PIXELS`. In short programs, a small comment in one place can allow for shorter names everywhere else.

There is often a balance to be struck when there is a temptation to use descriptive but super-long-and-unwieldy names. As a rule of thumb, I like to give things names that are appropriate for their scope. If a variable is used in a large program or by things outside of a library, a more descriptive name makes a lot of sense. If, however, it is in a small scope and you can see all of its uses really easily, a shorter name makes sense. In general,

make a variable as descriptive as necessary given its scope and the distance to its use, but no more descriptive than that.

We declare `CELLS` (cells along each side of our grid) to be 20. That's the number of cells on a side, so the full grid will be 20 by 20. We then compute from that and the canvas size how many pixels are on a side of each individual cell. The reasoning is this: if there are 20 cells across the full canvas, and the canvas is, say, 300 pixels wide, then each cell will be $300 / 20 === 15$ pixels wide. We can easily check that reasoning: if we have 20 cells, each 15 pixels wide, then the canvas has to be 300 pixels wide to accommodate them.

Note that we assume that the width is the same as the height, since later we use `CELL_PIXELS` for both width and height. If that assumption is wrong, things will look really strange. We will stick to a square canvas for this game.

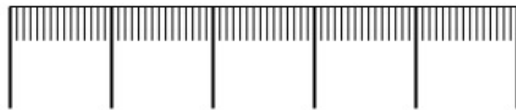


Figure 13.4: Lengths for cells (large) and pixels (small).

Armed with this information, you can start to see how we would transform grid cell coordinates to canvas coordinates. If you look at the coordinates we've been using for snake segments and food, we have been using integer values like (5, 7). To convert those to pixels, we need to scale up from cell coordinates to pixel coordinates. If each cell is 15 pixels wide, for example, cell (5, 7) would correspond to pixel (75, 105), which we obtain by multiplying each coordinate by `CELL_PIXELS`. That's a good start! If our cell coordinates are in `x` and `y`, the calculation might look something like this:

:javascript:

```
var xPx = x * CELL_PIXELS,
    yPx = y * CELL_PIXELS;
```

So far, so good, but we're going to draw a circle in that cell, which means we need to find the center of it. What we have computed is actually the upper left corner. To get to the center, we need to add half of `CELL_PIXELS` in each direction, like this:

:javascript:

```
var xPx = x * CELL_PIXELS + 0.5 * CELL_PIXELS,
    yPx = y * CELL_PIXELS + 0.5 * CELL_PIXELS;
```

And, we can make the computation a bit simpler by factoring `CELL_PIXELS` out of each equation:

:javascript:

```
var xPx = CELL_PIXELS * (x + 0.5),
    yPx = CELL_PIXELS * (y + 0.5);
```

All we did was factor the common element there, but that makes it look like we're adding half to our cell coordinates, then converting that to pixel coordinates. It turns out to be the same thing. If we are thinking about things in cell coordinates, we can find the center of any cell by adding 0.5 to it in cell coordinates. If we then scale things up to pixel coordinates, everything works. That's comforting: algebra works the same as the geometric interpretation of it.

If, every time we go to draw something, we convert to pixel coordinates in this way, we will get things exactly where they need to be in the canvas. The radius of our circle will be roughly $\text{CELL_PIXELS} / 2$ so that it fills the entire cell.

Let's make this concrete: if we have a canvas that is 300 pixels wide and 300 pixels tall, and we're using a 20-by-20 grid of cells, then there will be $300 / 20 = 15$ pixels on each side of a cell. We figured that out earlier. Now, to find the pixel location of the center of a circle in cell (3, 5), we get $xPx = 15 * 3.5$ which is 52.5 . We also get $yPx = 15 * 5.5$, which is 82.5 . The canvas doesn't have a problem with fractional pixel values (and we will see why in a moment), so it allows us to draw a circle at location (52.5, 82.5) without any trouble. The radius of our circle will be 7.5, and that lets it fill in the cell as much as possible.

All told, this isn't too bad. The computations are pretty simple, and we can make them convenient by putting them inside of functions. We can just keep track of cell coordinates until we actually need to draw something.

Scale and Translation

There is another way to go about this, though. We can have the canvas do all of these calculations for us by applying a **transform**¹ to the context.

We kind of encountered this earlier when doing that little bit of algebraic manipulation. Converting from cell to pixel coordinates involves translating (adding) by 0.5 , then scaling (multiplying) by CELL_PIXELS .

When we do this with the context, we do it in reverse because instead of changing our values from cell coordinates to pixel coordinates, we're changing the canvas from pixel coordinates to our more desirable cell coordinates. After scaling, each unit square occupies a larger space of the canvas, and then the translation by 0.5 of those units makes sense.

We accomplish this using the `scale` and `translate` functions of the canvas object. There are other transform operators as well, but these are the most straightforward, specifying how much to translate and scale in the x and y directions:

:javascript:

¹https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial/Transformations

```
ctx.scale(CELL_PIXELS, CELL_PIXELS);
ctx.translate(0.5, 0.5);
```

Once those transforms have been applied, the canvas accepts our cell coordinates directly. Thus, when we go to draw a circle at coordinates (3, 5), those will be put into exactly the place we want them: in the center of cell (3, 5). We can draw circles with radius 1/2 and they will look exactly the way we want. This simplifies life for us quite a bit!

There is one thing that gets a smidgen more complicated, though: clearing the canvas before drawing on it. There are lots of options for this, as well, including saving and restoring canvas state and resetting to a pixel transform, but look at the `clearRect` call below: we just opt to clear it in grid coordinates instead (remembering that the coordinates correspond to the centers of cells, not their corners).

Putting this together with the standard game skeleton, we can see exactly how it looks to output all of the cell circles:

:html:

```
<canvas id="drawing" width="300" height="300"
  style="border: 1px solid gray"></canvas>
<script src="animate.js"></script>
<script>
var canvas = document.getElementById('drawing'),
    ctx = canvas.getContext('2d');

var CELLS = 20,
    CELL_PIXELS = canvas.width / CELLS;

ctx.scale(CELL_PIXELS, CELL_PIXELS);
ctx.translate(0.5, 0.5);

var animation = animate({
  draw: function() {
    // Clear the canvas.
    ctx.clearRect(-0.5, -0.5, CELLS, CELLS);

    for (var x = 0; x < CELLS; ++x) {
      for (var y = 0; y < CELLS; ++y) {
        fillCircle(ctx, x, y, 0.45);
      }
    }
  },
});

// Allow pausing.
canvas.addEventListener('click', animation.togglePaused);
document.addEventListener('keydown', function(event) {
  if (event.key === ' ') {
    animation.togglePaused();
  }
});
```

```

    }
  });
</script>

```

Note that the radius is 0.45 because 0.5 doesn't actually look all that nice. It completely, uncomfortably fills the cells, and that ends up making things a bit cluttered. Go ahead and experiment with different values.

The most important thing to notice, though, is this: we still make the calculations of how big the cells will be, but when we go to draw, we just pretend that we're drawing in grid coordinates and it all works out. The `fillCircle` call is using grid cell coordinates, and the radius of the circles is roughly 1/2 instead of whatever the number of pixels should be to fill out a cell.

We can now draw circles right where we want them, using coordinates that feel natural.

Alternate Canvas Clearing Method

The `clearRect` method in the previous section is somewhat unfortunate, because we can't change our canvas transformation without also changing our clearing code. That feels wrong, because clearing a canvas shouldn't depend on how we draw on it, we just want to clear it no matter what its coordinate system looks like. When you have strange dependencies in your code like that, it's a good idea to find out whether you can get rid of them.

In this case, we definitely can. The current transformation is represented as a 3x3 matrix with scale and translation factors in it, and we can reset it by forcing it to be the identity matrix, then we can clear it using its width and height like before.

Note that you can see the current transform matrix by calling `ctx.getTransform()` on the canvas context.

In our case, resetting the transform means changing one matrix to another:

$$\begin{pmatrix} 15 & 0 & 7.5 \\ 0 & 15 & 7.5 \\ 0 & 0 & 1 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

The transform can be set using the `ctx.setTransform(a, b, c, d, e, f)` function, where the variables sit like this in the matrix:

$$\begin{pmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{pmatrix}$$

That's reasonably straightforward: we can call `ctx.setTransform(1, 0, 0, 1, 0, 0)` to reset it to identity. We can take advantage of the context's `save` and `restore` functions to keep from messing with our scaling and translation in other parts of the code, too. In fact, we can write a function to do this and put it into `animate.js`. Let's see what that looks like:

:javascript:

```
// somewhere in animate.js
function clearContext(ctx) {
  ctx.save();
  ctx.setTransform(1, 0, 0, 1, 0, 0);
  ctx.clearRect(0, 0, ctx.canvas.width, ctx.canvas.height);
  ctx.restore();
}
```

We could easily move this into the library because it no longer needs to know anything about how `draw` likes to think of canvas coordinates. It is independent of anything in any of our animation logic and will always clear the canvas no matter what state it is in.

Now you can just call `clearContext(ctx)` after importing `animate.js`.

Representations

It's time to start talking more about how to keep track of things in this game. The state will include

- All snake positions,
- The current direction of the snake's head, and
- The current location of food.

Since the position of each snake segment is needed, we will use an array. Since each segment has two values for the position (`x` and `y`), we will store those coordinates in an object with `x` and `y` keys. This declares the segments and starts the game off with only one segment (the head) in the middle of the grid. Note how `segments` is an array with one element, an object containing `x` and `y` values:

:javascript:

```
var segments = [
  {x: Math.floor(CELLS / 2),
   y: Math.floor(CELLS / 2)},
];
```

We use the `Math.floor` function because we have to work with integer cell numbers. In the case of a 20-by-20 grid, this won't do anything since the halfway point is 10, and that's already an integer. If we had an odd number of cells on a side, however, this would pick something slightly off-center to make sure it was occupying a whole cell.

The location of the food will generally be random, and that can be computed in the normal way, again making sure that we get integer cell coordinates out of it:

:javascript:

```
var food = {
  x: Math.floor(Math.random() * CELLS),
  y: Math.floor(Math.random() * CELLS),
};
```

Finally, the snake will have a random initial direction. How should directions be represented? They could be represented as numeric changes to *x* and *y*, but we will represent them as letters and interpret them as grid motion later:

:javascript:

```
var direction = "UDLR"[Math.floor(Math.random() * 4)];
```

Because strings can be viewed as arrays of characters, this selects a random character out of the "UDLR" string (one of "up", "down", "left", or "right"). When it's time to move the snake, the current direction will be used to determine where the head goes. After that's determined, the rest of the snake will follow it.

We are now ready to make something happen—as per our usual practice, we will get a simplistic version of the game working first and then start adding things until it's all done. What we have right now is enough to draw the food and the snake. Motion will come later.

Drawing the Snake and Food

The skeleton game now looks like this. Note the `scale` and `translate` functions to get the context into cell coordinates, and the use of `fillCircle` to draw the food, the snake's head, and the rest of the snake's body (currently non-existent). There should be no surprises here.

:html:

```
<canvas id="drawing" width="300" height="300"
  style="border: 1px solid gray"></canvas>
<script src="animate.js"></script>
<script>
var canvas = document.getElementById('drawing'),
    ctx = canvas.getContext('2d');

// Put things into grid coordinates.
var CELLS = 20,
    CELL_PIXELS = canvas.width / CELLS;
ctx.scale(CELL_PIXELS, CELL_PIXELS);
ctx.translate(0.5, 0.5);

var segments = [
```

```

    {x: Math.floor(CELLS / 2),
      y: Math.floor(CELLS / 2)},
  ];
  var food = {
    x: Math.floor(Math.random() * CELLS),
    y: Math.floor(Math.random() * CELLS),
  };
  var direction = "UDLR"[Math.floor(Math.random() * 4)];

  var animation = animate({
    draw: function() {
      // Clear the canvas.
      clearContext(ctx);

      // Draw the food.
      ctx.fillStyle = 'green';
      fillCircle(ctx, food.x, food.y, 0.45);

      // Draw the snake head.
      ctx.fillStyle = 'purple';
      fillCircle(ctx, segments[0].x, segments[0].y, 0.45);

      // Draw the rest of the snake a different color.
      ctx.fillStyle = 'blue';
      for (var i = 1; i < segments.length; ++i) {
        fillCircle(ctx, segments[i].x, segments[i].y, 0.45);
      }
    },
  });

  // Allow pausing.
  canvas.addEventListener('click', animation.togglePaused);
  document.addEventListener('keydown', function(event) {
    if (event.key === ' ') {
      animation.togglePaused();
    }
  });
});
</script>

```

When you run this, you will see a snake head and some food on the canvas. The food will change positions each time you reload the page because its location is random. The snake head should always be roughly in the center.

Moving the Snake

To make the snake move, the current direction determines where the head will go, and then each other segment will take the place of the one in front of it (playing follow-the-leader). Conceptually, when the snake moves, the head goes to a new cell, and every segment behind

it moves to take the place of what used to be in front of it. But we can make things easier on ourselves than that. We can just create a brand new head in the new location and throw the tail (the last segment) away. That looks exactly the same and is easier to code.

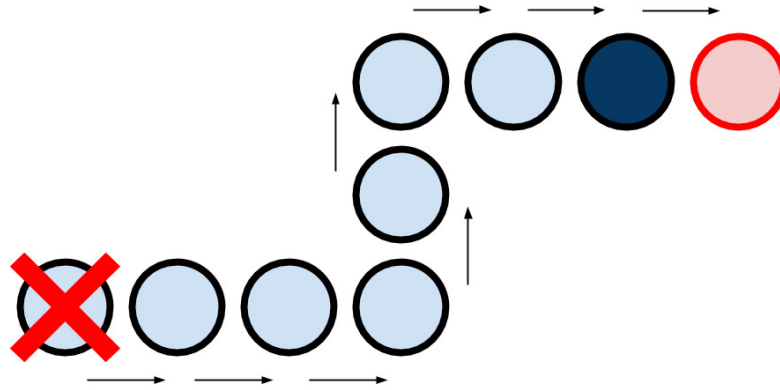


Figure 13.5: Removing a tail and adding a head is like motion.

In general, even if it isn't the most efficient approach, a simple and correct approach is the right way to start. Then, change to a more complex approach only once things are working and there is a clear need for the complexity. For example, sometimes a more complex approach is a lot faster. In our game it just isn't going to matter much, so simplicity wins.

Since we have an array of segments and we are going to be adding heads and removing tails all the time, now is a good time to talk about some useful array methods:

- `push`²: Add an element to the end.
- `pop`³: Remove an element from the end.
- `unshift`⁴: Add an element to the front (position 0).
- `shift`⁵: Remove an element from the front (position 0).

The `shift` and `unshift` operations do exactly what they say: they shift all of the other elements around to either fill in the space created with a removed element (`shift`) or make room for the new element (`unshift`). This sounds like exactly what we want. We chose 0 as the position of the head (check the code), so moving will simply create a new head based on the direction of motion from the old head, and everything else will scoot over to make room. Then the last element—the end of the tail—can be removed with `pop`:

:javascript:

²https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/push

³https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/pop

⁴https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/unshift

⁵https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/shift

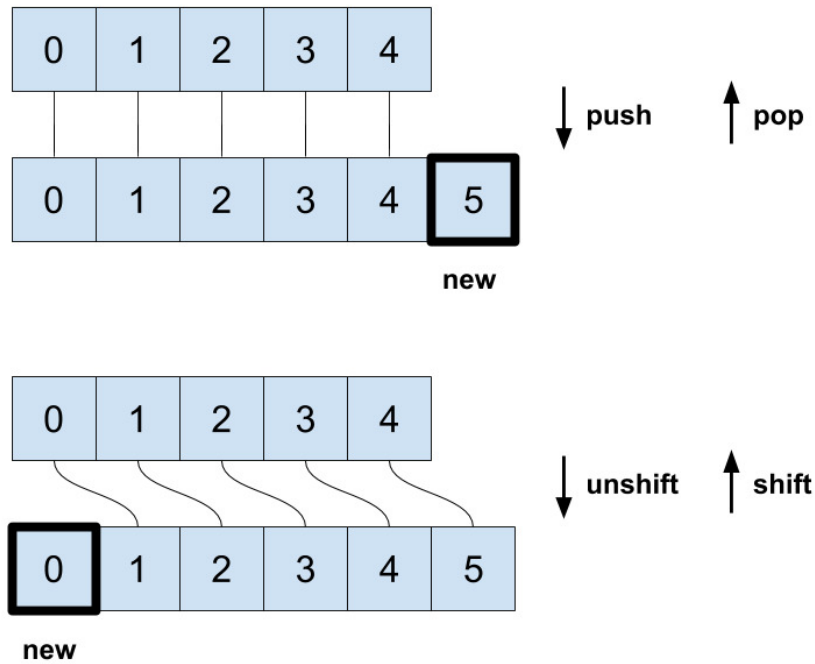


Figure 13.6: Array push/pop and unshift/shift operations.

```

move: function(dt) {
  // Create a new head, same as the old head.
  var newHead = {
    x: segments[0].x,
    y: segments[0].y,
  };

  // Then change it based on current direction.
  switch (direction) {
    case 'U': --newHead.y; break;
    case 'D': ++newHead.y; break;
    case 'L': --newHead.x; break;
    case 'R': ++newHead.x; break;
  }

  // Remove the tail.
  segments.pop();

  // Add the new head.
  segments.unshift(newHead);
},

```

That looks simple enough: copy the old head, set the copy to occupy one space over, remove the tail, and add the new head copy. That's how you move a snake.

Note that the cases each only take one line, since the test, resulting code, and `break` are all together. Sometimes that's not very readable, but I think it works pretty well in this situation, particularly since it puts the differences right next to each other and the lines are not very long. Readability is sometimes about making the right trade-offs, not about blind adherence to a rulebook (though sometimes blind adherence is the least of all available evils).

It's the principles that count.

Now, there is something pretty wrong with all of this. Before you run it, can you guess what it is?

Hint: how fast will the snake move?

Got it yet?

We aren't using `dt` at all! It isn't affecting our motion calculations in the slightest. Therefore, every single frame, 60 times per second, we advance the snake. Oops. That's *super* fast. We could slow it down by making those `++` and `--` operations instead be addition and subtraction by some velocity times `dt`, but that won't quite work for this game: the circles can't occupy two cells at once; they're either in a cell or not. They need to advance discretely, not continuously.

What we really need to do is advance only after a certain amount of time has passed, then teleport into the next square all at once. That's going to require a different kind of logic than what we have used previously. Instead of moving a smaller amount with each frame, we need to move all at once, every few frames. To do that, let's first determine our speed in terms of the grid.

Because this is all happening in a universe of our own creation, we are not constrained by physics, and we get to choose our own units! Let's talk about speed in terms of "cells per second".

Suppose we wanted to advance 5 grid cells per second. In our universe that means our speed is $5 \frac{\text{cell}}{\text{s}}$ ("s" is "seconds"). But since we aren't moving continuously, what we really want to know is how long to spend in each cell. That means we need time, not speed, so we need to flip this over (multiplicative inverse):

$$5 \frac{\text{cell}}{\text{s}} \implies 0.2 \frac{\text{s}}{\text{cell}}$$

OK, that works. If we want to move 5 cells per second, we need to stay in our current position for 0.2 seconds before moving to the next one.

So, we basically just wait between moves. Waiting sounds like something we can do! In the `move` function, if we aren't done waiting from last time, we just won't change anything. We

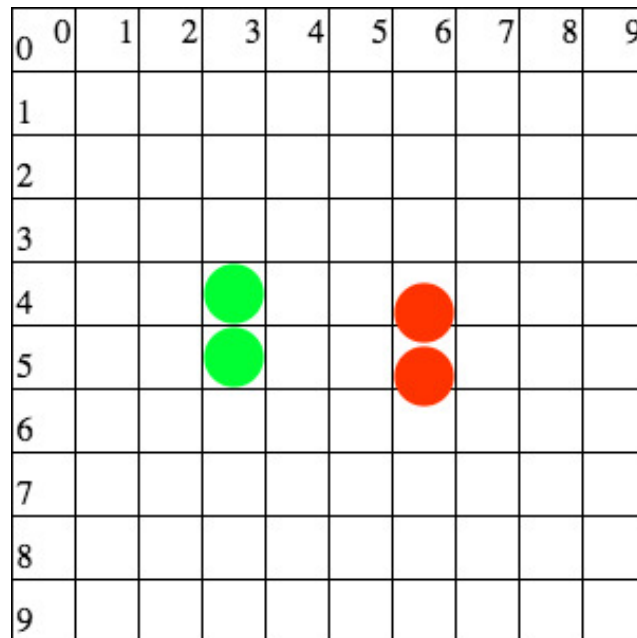


Figure 13.7: Segments on the left are okay. On the right, not so much.

can do that with an early `return`.

When we finally reach a moment when we have waited long enough to move again, we will advance the snake, figure out when the next time to move will come, and start ignoring frames once more until that time.

To make that work, we need to keep track of our next allowed time to move, then only move when we get past that new time. Add the following to the state section:

:javascript:

```
var MOVE_EVERY = 0.2,
    nextMoveTime = 0;
```

Then, in the very beginning of `move`, decide whether it is time to move based on elapsed time. If not, return immediately (no motion), otherwise compute when the subsequent move will be and go ahead and execute this one.

:javascript:

```
move: function(dt) {
  // Don't move if it isn't time, yet.
  if (animation.elapsed() < nextMoveTime) {
    return;
  }
  // Time to move - store the *next* move time for later.
  nextMoveTime = animation.elapsed() + MOVE_EVERY;
```

```
// Actual move code goes here.
},
```

Do you see how that works? We don't move at all if it isn't time yet, we just wait for another frame to try again. Then, when enough time has passed that the current time (elapsed) is past our "wait until here" time, we move and set up the next time we want to move again. We always move on the first call, since the next time to move starts out at 0.

That works!

Now if you reload the page, you will see the snake progressing, one cell at a time, at roughly 5 cells per second. Excellent. Our change worked. Now all we need to do is respond to arrow keys and we will have something we can control!

Changing Directions

The simplest thing is to let the arrow keys be the controls for the snake. When a key is pressed, it will change the current direction character. We can control this behavior with a `switch` statement, or with an `else if` pattern.

Or we can learn a new thing!

Let's look at yet another way of handling multiple cases: using an object to map the key event to the outcomes we want. Note: we will just add this event listener, not adjust the one already there that handles the space key. It's fine to have two event listeners for the same event, and this is a good chance to see that in action. After all, it is called `addEventListener`, not `replaceEventListener`, `setEventListener`, or `obliterateAllPreviousEventListeners`.

When two or more event listeners have been added for the same event, they will be run in the order added when that event occurs. In our case, that's the "keydown" event:

:javascript:

```
document.addEventListener('keydown', function(event) {
  var keyDirs = {
    'ArrowUp': 'U',
    'ArrowDown': 'D',
    'ArrowLeft': 'L',
    'ArrowRight': 'R',
  };

  var dir = keyDirs[event.key];
  if (dir) {
    direction = dir;
  }
});
```

This technique takes advantage of the fact that a missing key in an object returns the value `undefined`, which is “falsy” and therefore doesn’t set the `direction` variable. We only set the direction variable when one of the expected arrow keys is pressed, and we set it to one of the values ‘U’, ‘D’, ‘L’, and ‘R’.

This should allow you to control the snake head. The most recent press before the snake moves is honored and makes the snake change direction.

A cute way of not even creating the `dir` variable in the first place is to use `||` short-circuiting to have a sensible default. Instead of this...

:javascript:

```
var dir = keyDirs[event.key];
if (dir) {
  direction = dir;
}
```

...we can actually do this:

:javascript:

```
direction = keyDirs[event.key] || direction;
```

It’s a common-enough pattern that it is readable by sheer ubiquity, so it’s also fine to do it this way if you like.

Limiting Motion

The snake can actually go backwards through itself right now. That usually isn’t allowed in this game, so we should make sure it can’t happen. To fix it, we will need to disallow backward motion in our mapping, and we can use a ternary conditional operator `? :` to do that.

Recall that the value of an expression `condition ? trueValue : falseValue` is determined by the truthiness of the `condition`. If it is true, the whole expression evaluates to `trueValue`, otherwise it evaluates to `falseValue`. The new mapping that disallows backward motion should give an idea of how that works:

:javascript:

```
document.addEventListener('keydown', function(event) {
  var keyDirs = {
    'ArrowLeft': direction === 'R' ? 'R' : 'L',
    'ArrowRight': direction === 'L' ? 'L' : 'R',
    'ArrowUp': direction === 'D' ? 'D' : 'U',
    'ArrowDown': direction === 'U' ? 'U' : 'D',
  };

  var dir = keyDirs[event.key];
  if (dir) {
```



```

        direction = dir;
    }
});

```

The map now leaves the direction alone if we try to go backwards, effectively ignoring what we told it to do. Take a look at the 'ArrowLeft' entry to see how this works: if the left arrow is pressed, and we are already going right, we keep going right, ignoring the left press. Otherwise, we go left as requested. This isn't perfect (it is possible to change direction quickly several times before a move occurs, creating a race condition), but it is good enough for our purposes for now.

What we have done here, though, is a bit overly cute and fairly repetitive. Sure, these things fit on one line each, but they aren't exactly readable and easy to follow. If you were coming at this fresh, not having seen it for a while, would you understand what was going on? No? Me neither. In fact, I had exactly this problem when editing this book; I could not remember why this code worked! That is a danger sign if ever there was one, and we will talk more about how to improve it in the exercises.

Eating

The snake never gets bigger in this game. In our `move` function, we always pop the tail, and we never check for a collision with the food. Let's change that.

Recall that the `move` function has this statement near the bottom:

:javascript:

```

// Remove the tail.
segments.pop();

```

Instead of unconditionally removing the tail, we should only remove it if the snake has not just eaten. If the snake has just eaten, a new head will be added (part of moving), and we can just leave the tail there, effectively expanding the snake by one segment:

:javascript:

```

// Eat or remove the tail.
if (food.x !== segments[0].x || food.y !== segments[0].y) {
    segments.pop();
}

```

Here we took advantage of De Morgan's Laws, though it might not be quite obvious. What we want to say is this: "If the snake's head is not on the food, pop the tail." The head is on the food when the `x` and `y` coordinates of `segments[0]` are both equal to those of `food`. That would most directly translate to the following:

:javascript:

```
if (!(food.x === segments[0].x &&
    food.y === segments[0].y))
```

But that's kind of cumbersome. We can take advantage of De Morgan and distribute that ! (NOT) operator into the inner parentheses. It will cause && to become || when it does so, and it will invert each of the interior expressions, producing

:javascript:

```
if (food.x !== segments[0].x || food.y !== segments[0].y)
```

In other words, if either the x or y coordinate is not equal between food and segments[0], then remove the tail (because the head can't be on the food—at least one of the coordinates doesn't match).

Sometimes using these laws can really help readability. Here, though, both approaches are pretty ugly and not terribly easy to read. When that sort of thing happens, it is probably time to write a function with a helpful name. Consider this:

:javascript:

```
// ...somewhere in the file...
function sameCell(p1, p2) {
    return p1.x === p2.x && p1.y === p2.y;
}
```

:javascript:

```
// Back in the move function:
if (!sameCell(food, segments[0])) {
    segments.pop();
}
```

That reads much better. Now it's clear what we are testing for: we want to pop the tail if the first segment (the head) is not in the same place as the food.

One thing remains: the food needs to be moved after the snake eats. That means we need to add an else clause to our new test:

:javascript:

```
// Eat or remove the tail.
if (!sameCell(food, segments[0])) {
    segments.pop();
} else {
    food.x = Math.floor(Math.random() * CELLS);
    food.y = Math.floor(Math.random() * CELLS);
}
```

When we only needed to know whether to remove the tail or not, the !sameCell version of this test made sense, but now it kind of feels backwards: we test for “not hitting the food”, and then when we get an else we move the food.

It sure seems like it would be clearer to do things the other way, now:

:javascript:

```
// Eat or remove the tail.
if (sameCell(food, segments[0])) {
  food.x = Math.floor(Math.random() * CELLS);
  food.y = Math.floor(Math.random() * CELLS);
} else {
  segments.pop();
}
```

There is a moral to this story: sometimes the expression of a condition is clearer one way with one action, but clearer the other way when there is an `else` body. So it goes. Programming is all about making things work, but maintainable programming is all about communicating to humans, and that is part art and part skill. The skill part is to remember to consider your audience. Read through the code and see if the logic makes more sense with fresh eyes one way or the other. The art part is in picking the clearest way whenever there is a choice. Quite often clarity translates to “fewer negations”.

Crash

And with that, we have food that moves. What’s left? Oh, yes, a way to lose the game. That is currently not possible. The way you lose in a real game of snakes is to crash the snake, either into a wall or into itself. That sounds like a bunch of tests in the `done` function. Let’s add them. First, the walls:

:javascript:

```
done: function() {
  var head = segments[0];
  if (head.x < 0 || head.x >= CELLS ||
      head.y < 0 || head.y >= CELLS) {
    alert("Game Over: wall crash");
    return true;
  }

  // Self-collisions here?

  return false;
},
```

That was easy! How do we test for crashing into the body? Well, if the head’s `x` and `y` coordinates overlap with any of the other segments, that’s a crash. That sounds like a loop where we test each non-head segment to see if we have crashed into it. Since the head is at position 0, we will just loop through everything starting at 1. Note that we can use our `sameCell` function again, which is a nice touch and a sign that it was a good function to write. This code is also inside of the `done` function:

:javascript:

```

for (var i = 1; i < segments.length; ++i) {
    var pos = snake.positions[i];
    if (sameCell(head, pos)) {
        alert("Game over: self crash");
        return true;
    }
}

```

The entire done function looks like this:

:javascript:

```

done: function() {
    var head = segments[0];

    if (head.x < 0 || head.x >= CELLS ||
        head.y < 0 || head.y >= CELLS) {
        alert("Game over: wall crash");
        return true;
    }

    for (var i = 1; i < segments.length; ++i) {
        var pos = segments[i];
        if (sameCell(head, pos)) {
            alert("Game over: self crash");
            return true;
        }
    }

    return false;
},

```

That's all there is to it. If we find any segment that overlaps with the head, we are done. We quit. The game ends.

We now have a playable snakes game!

There are several things that we did not do in this game that we did in other contexts, such as showing the pause state in a button, or showing the clock or a score (which could be the length of the snake, here). There are also a few interesting bugs that you might find amusing to try to solve. For example, if you are moving up, then you quickly hit the right arrow key followed immediately by the down arrow key before the snake has a chance to react, you will crash into yourself. True story. That probably should not be possible in a real game, but it is in this one, and it requires a bit of thought to get the directions to only change when the snake can actually move.

We also have no indication that the game has ended, and we don't speed up as time goes on. All of these are things you can implement yourself with the skills and knowledge that you have right now.

Listings

animate.js

This one gained the ability to clear canvas contexts, so it's the same as before, with one more function in it:

:javascript:

```
function animate(config) {
    var move = config.move || function() {},
        draw = config.draw || function() {},
        done = config.done || function() {};

    var lastTime = 0,
        frame = null,
        elapsed = 0,
        EXPECTED_DT = 1 / 60;

    function tick(t) {
        if (done()) {
            frame = null;
            return;
        }
        frame = requestAnimationFrame(tick);

        var dt = (t - lastTime) / 1000;
        if (dt > 4 * EXPECTED_DT) {
            dt = EXPECTED_DT;
        } else if (dt < 0) {
            dt = 0;
        }
        elapsed += dt;
        lastTime = t;

        draw();
        move(dt);
    }

    frame = requestAnimationFrame(tick);

    return {
        running: function() {
            return !!frame;
        },
        elapsed: function() {
            return elapsed;
        },
        pause: function() {
            if (frame) {
                cancelAnimationFrame(frame);
            }
        }
    };
}
```

```

        frame = null;
    },
    start: function() {
        if (!frame) {
            frame = requestAnimationFrame(tick);
        }
    },
    togglePaused: function() {
        if (frame) {
            cancelAnimationFrame(frame);
            frame = null;
        } else {
            frame = requestAnimationFrame(tick);
        }
    },
};

function fillCircle(ctx, x, y, r) {
    ctx.beginPath();
    ctx.arc(x, y, r, 0, 2*Math.PI);
    ctx.fill();
}

function clearContext(ctx) {
    ctx.save();
    ctx.setTransform(1, 0, 0, 1, 0, 0);
    ctx.clearRect(0, 0, ctx.canvas.width, ctx.canvas.height);
    ctx.restore();
}

```

snakes.html

:html:

```

<canvas id="drawing" width="300" height="300"
        style="border: 1px solid black"></canvas>
<script src="animate.js"></script>
<script>
var canvas = document.getElementById('drawing'),
    ctx = canvas.getContext('2d');

// Put things into grid coordinates.
var CELLS = 20,
    CELL_PIXELS = canvas.width / CELLS;
ctx.scale(CELL_PIXELS, CELL_PIXELS);
ctx.translate(0.5, 0.5);

var segments = [

```

```

    {x: Math.floor(CELLS / 2),
      y: Math.floor(CELLS / 2)},
  ];
  var food = {
    x: Math.floor(Math.random() * CELLS),
    y: Math.floor(Math.random() * CELLS),
  };
  var direction = "UDLR"[Math.floor(Math.random() * 4)];

  var MOVE_EVERY = 0.2,
      nextMoveTime = 0;

  // Indicates whether p1 and p2 cell locations
  // are the same.
  function sameCell(p1, p2) {
    return p1.x === p2.x && p1.y === p2.y;
  }

  var animation = animate({
    draw: function() {
      // Clear the canvas.
      clearContext(ctx);

      // Draw the food.
      ctx.fillStyle = 'green';
      fillCircle(ctx, food.x, food.y, 0.45);

      // Draw the snake head.
      ctx.fillStyle = 'purple';
      fillCircle(ctx, segments[0].x, segments[0].y, 0.45);

      // Draw the rest of the snake a different color.
      ctx.fillStyle = 'blue';
      for (var i = 1; i < segments.length; ++i) {
        fillCircle(ctx, segments[i].x, segments[i].y, 0.45);
      }
    },
    move: function(dt) {
      if (animation.elapsed() < nextMoveTime) {
        return;
      }
      nextMoveTime = animation.elapsed() + MOVE_EVERY;
      // Create a new head, same as the old head.
      var newHead = {
        x: segments[0].x,
        y: segments[0].y,
      };

      // Then change it based on current direction.
      switch (direction) {

```

```

    case 'U': --newHead.y; break;
    case 'D': ++newHead.y; break;
    case 'L': --newHead.x; break;
    case 'R': ++newHead.x; break;
  }

  if (sameCell(food, segments[0])) {
    food.x = Math.floor(Math.random() * CELLS);
    food.y = Math.floor(Math.random() * CELLS);
  } else {
    segments.pop();
  }

  // Always add the new head.
  segments.unshift(newHead);
},
done: function() {
  var head = segments[0];

  if (head.x < 0 || head.x >= CELLS ||
      head.y < 0 || head.y >= CELLS) {
    alert("Game Over: wall crash");
    return true;
  }

  for (var i = 1; i < segments.length; ++i) {
    var pos = segments[i];
    if (sameCell(head, pos)) {
      alert("Game Over: self crash");
      return true;
    }
  }

  return false;
},
});

// Allow pausing.
canvas.addEventListener('click', animation.togglePaused);
document.addEventListener('keydown', function(event) {
  if (event.key === ' ') {
    animation.togglePaused();
  }
});
document.addEventListener('keydown', function(event) {
  var keyDirs = {
    'ArrowLeft': direction === 'R' ? 'R' : 'L',
    'ArrowRight': direction === 'L' ? 'L' : 'R',
    'ArrowUp': direction === 'D' ? 'D' : 'U',
    'ArrowDown': direction === 'U' ? 'U' : 'D',
  }

```



```
};

var dir = keyDirs[event.key];
if (dir) {
    direction = dir;
}
});
</script>
```

Exercises

Exercise 13-1: Lab: Improving the Snake Game

Solution on page [474](#)

For this lab, several improvements will be made. Some of them will come from things we did in the ball and paddle game, and some of them will be new. Here they are:

- Add a score to the game (the length of the snake minus 1).
- Increase the speed of the snake every time it eats.
- Fix it so the snake can never go backward, not even if you go sideways and backward really fast.

The first two should be pretty easy to make work, given that we've done things like them in the previous chapter. The third one might require some more thought. Some possibilities for fixing it might include

- Keep track of all direction changes and process them in order, or
- Keep track of the current direction separate from the requested direction, and only change the current direction when moving, if it's allowed.
- Simply don't let the snake head hit the very next segment. It would not be possible in normal play anyway.

The last of these is easiest. Feel free to toy around with the others, but a valid solution can also be an easy solution.

Chapter 14

Abstractions With Classes and Objects

Until this point we have made good use of the abstractions in “animate.js”. They tend to allow us to get down to the business of designing our games, freeing us from thinking about low-level details that are always the same, like how to manage time, and what pausing should accomplish.

Closures are a powerful mechanism for creating these abstractions, and we have made abundant use of them. Not only do we pass closures into the `animate` function to get the behavior that we want, but it also passes closures back to us so that we can control things while they’re happening and get information out of our animations.

We have also used **objects** (in the sense of “maps from keys to values”) to collect similar bits of data together, like the `x` and `y` coordinates of snake segments. In this chapter, we are going to look at a different way of specifying these object and function abstractions using **classes**. This is part of what is called “Object-Oriented Programming”, although JavaScript’s version of it is fairly unique.

We will cover the fundamental language features that make classes possible, and then we will introduce the newer ECMA 2015 (ES 6) syntax for classes that makes things a bit cleaner. Understanding both the old and the new ways is unfortunately necessary for now, because there is still a tremendous amount of code out there using the original approach, because online documentation still references `prototype` methods, and because the new syntax is just that: syntax. It does the same kinds of things inside.

Let’s begin with a review of objects as data.

Objects as Data Containers

As mentioned in the introduction, this next topic is going to touch on a few intermediate-to-advanced features of JavaScript, and has some brand new concepts in it.

These concepts can themselves be somewhat abstract, so it may require a bit of head-scratching and a lot of playing around to get through this section. That's okay. These concepts usually require multiple exposures to get right; if this is your first time, it's not expected that things will just click. That said, I will make every effort to ease your way.

We are going to talk more about **objects** now, and we are going to introduce special objects that are also **instances** of **classes**. There is some new vocabulary in here, and we will start to understand it in detail as we continue on.

Let's start with the basics. Take another look at the state that we keep track of in our paddle game from a couple of chapters back:

:javascript:

```
var PADDLE_HEIGHT = 10,
    PADDLE_Y = canvas.height - PADDLE_HEIGHT;

var paddleWidth = 100,
    paddleSpeed = canvas.width - paddleWidth,
    paddleX = (canvas.width - paddleWidth) / 2;

var BALL_RADIUS = 10,
    ballX = canvas.width / 2,
    ballY = canvas.height / 2,
    ballVelX = canvas.width / 2,
    ballVelY = canvas.height / 2;
```

That's quite a number of variables and constants! If you squint at it, you can see that we sort of grouped things that belong together. Paddle values are all in one place and ball things are in a different place. That was for sanity's sake. We could easily have mixed them all up and moved them around so long as their initial values came before the code that uses them, but then it would be hard to keep track of what is happening.

Unfortunately, this grouping doesn't help us much when trying to pass values around our program. If, for example, we want to pass information about a ball into a function in a library, we have to pass a lot of variables into it. If we later add more data about the ball, we have to remember to change all of the places that we send ball data and add yet another parameter. And, with a lot of parameters, it's easy to forget their order when programming.

We already know how to group things, though, and to name them so it's easier to pass them into functions: we pass an object. Let's change the variables in our game program into objects, one per "thing" in the game:

:javascript:

```
var paddle = {
  height: 10,
  width: 100,
};
paddle.speed = canvas.width - paddle.width;
```

```

paddle.x = (canvas.width - paddle.width) / 2;
paddle.Y = canvas.height - paddle.height;

var ball = {
  radius: 10,
  x: canvas.width / 2,
  y: canvas.height / 2,
  velX: canvas.width / 2,
  velY: canvas.height / 2,
};

```

What do you notice about the above code? Here are some of my observations:

- The internal names of things are shorter because they “belong” to something that helps to differentiate them. We don’t need to say “ballX” because we will be using it as `ball.x`, which is plenty descriptive.
- Sometimes you can create an object all at once, as with `ball`, but sometimes an object’s values are derived from other values, so we set them later, as with `paddle`.
- We haven’t really saved much space, yet, and this still looks fairly messy.
- There are only two variables, now, instead of whatever ridiculous number there were before.

While it is true that it still looks fairly messy, we have actually greatly simplified our lives and made our code more flexible. If we want a function to do something with a ball and a paddle (like test whether we have a hit), we can have it accept only 2 arguments and tell us the answer. Let’s see what that would look like:

:javascript:

```

function hit(ball, paddle) {
  if (ball.y + ball.radius >= paddle.Y) {
    if (ball.x >= paddle.x &&
        ball.x <= paddle.x + paddle.width) {
      return true;
    }
  }
  return false;
}

```

That’s pretty nice, and we only gave it two parameters. Each of those parameters has everything in it that we need. They also have more than we need, but that’s okay—we just use what we need and don’t worry about the rest.

If we had needed to pass in all necessary variables for `hit` before, it might have looked like this instead:

:javascript:

```

function hit(ballX, ballY, paddleX, paddleWidth) {
  ...
}

```

```
}
```

Now imagine calling that. There are several parameters (even assuming some constants don't need to be passed), and order is important. We could, of course, accept a parameter object like we now do with `animate`, but that just points us back to our new approach: passing objects instead of long parameter lists. The new idea here is that things are already objects when we pass them to (the first version of) `hit`.

This idea of grouping data together is powerful and you should make liberal use of it. It's easy to see that grouping bits of data together can keep things clean and clarify how things are related.

But things don't just have state, they also have behavior. How do we group behaviors in with state? It's not so easy to motivate this question right now, so you might be wondering why we even care. The reason is the same as for grouping variables: many functions that we write in our program only operate on one kind of data. They are not really useful for anything else, so they might as well go with the data they operate on. Stick with this and it will become more concrete very soon.

We already know that functions and data can go together, because when we created the `animate` function, we had it pass out closures that operate on the data in their outer scopes, allowing us to control behavior.

This was very useful—the `animate` function gave us back an object that had state and provided behaviors that manipulated that state. The trouble with using closures to do this, however, is that the interpreter has to create them from scratch every time you make an object.

Consider how many behaviors the `String`¹ type exports via its prototype: creating all of those functions every time we make a new string would take a nontrivial amount of time and would be very wasteful of computer memory; functions take up space just like data, particularly when they are closures. It's particularly wasteful because functions like `split` or `toUpperCase` always do the same thing no matter what string they're applied to; the only thing that changes between string objects is the data being manipulated, not the behavior of the manipulation.

We touched on this very briefly at the beginning of the course when talking about how functions like `"hello".toUpperCase()` can operate on the thing to the left of the dot, and now we're going to dig in and see more about how to make our own functions do the same thing without relying on closures.

We will do this by creating a brand new `Animation` class that has the same functionality as our previous `animate` function (though we will have to exercise more care than before when using things like `animation.togglePaused`; stay tuned).

¹https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String

Animation Class

When we call `animate`, it returns an object that contains functions we can call to work with only that animation. We call this a “class by closure”. It contains data and behaviors, and it exposes them using closures. As mentioned before, there is nothing wrong with that in most cases, and it is perfectly fine to leave that class alone. We are only poking at it because it is a good case study for learning a language feature.

Constructor

Let’s turn it into a class. Every class needs what is called a **constructor**, meaning a function that sets it up and gets it ready to go. It also returns a new **instance** of the class. An instance is an object that is “patterned after” the class, and is created before the constructor body is entered.

A common metaphor for a class is a “blueprint”. The blueprint details how a house is to be built, including doors, windows, roofing, walls, etc. It also details how water and electricity and ductwork interact, which is part of the house’s “behavior” (faucets, light switches, outlets, and thermostats are points of interaction). In this ages-old metaphor, a blueprint is to a **class** as the actual house is to an **instance**. Calling a constructor creates an instance of our class.

Here is a very basic constructor for our new `Animation` class. It just stores the things passed into it:

:javascript:

```
function Animation(config) {
  this._move = config.move || function() {};
  this._draw = config.draw || function() {};
  this._done = config.done || function() { return false };

  this._EXPECTED_RATE = 1/60;
  this._lastTime = 0;
  this._elapsed = 0;
  this._frame = null;
}
```

Let’s start by getting the trivial bits out of the way. We are naming implementation details like `elapsed` and `lastTime` using a leading underscore—this signals to people looking at our class that they should not be accessing those things directly, because they are not part of what we expect them to be using. It’s just a convention, but it’s a fairly standard one. JavaScript allows you to access all sorts of things no matter how they are named, but this at least sends humans a message that some of your variables are not safe to rely on from the outside because they are implementation details that might change. They aren’t part of the “contract” for the class.

The most important thing to note is the use of the special `this` variable, which is available to

all standard functions as a hidden parameter (the first one, in fact), and of particular interest in functions defined on classes. You can think of `this` as “*this instance* of this class”. It is a special object that we can add data to, and we do that inside of our constructor. It sort of magically appears for us when the constructor is called with the special keyword `new`:

:javascript:

```
var animation = new Animation({...});
```

The `new` keyword signals to the interpreter that `Animation` should not just be called, but that it should be treated like a constructor and it should return an instance of the `Animation` class. It dutifully creates one, sets it as `this` in the `Animation` function, and then calls it, returning the new object it just created.

For most intents and purposes, `this` is just a plain old JavaScript object, so you can do all of the expected things to it: setting, getting, and deleting properties.

Methods (Prototype Functions)

So far, so good. We now have an overly complicated way of creating an object using a constructor, but there are no behaviors on it, yet. In object-oriented parlance, functions that operate on an instance are called **methods**. We add methods to a class using the constructor’s **prototype** object, so it is now time to talk a little bit more about what that object does.

Every object in JavaScript has a `prototype` property. We will not get into all of the details of what that means, but we will make use of it, because prototype objects are the key to JavaScript classes and implementing common behaviors that are relatively space-efficient.

If you have looked at any of the documentation for JavaScript, you have seen evidence of `prototype` objects already. For example, the string method `toUpperCase` is documented as `String.prototype.toUpperCase`. That means if you have an instance of type `String` (that is its class name), you have to get its prototype first, and then get the `toUpperCase` function from that. But you never really use it that way. Why is it that you can call it like `"hello".toUpperCase()` and not `"hello".prototype.toUpperCase()`?

The answer has to do with how JavaScript searches for functions on objects, and what it does when it finds them on a prototype. Consider a call like this:

:javascript:

```
"hello".toUpperCase()
```

The JavaScript interpreter knows to look at the object itself (the string `"hello"`) for something called `toUpperCase` and, when it can’t find it there, it knows to search the object’s `prototype` for it. There’s one critical feature of this search that makes it all work: the interpreter will set `this = "hello"` if it finds `toUpperCase` on the `prototype`

object. Otherwise it does something far less useful that we will talk about later. The important thing to note is that `prototype` functions have `this` set to the instance of the class when called directly on it, like `"hello".toUpperCase()`.

This feature of prototype search and automatic `this`-setting gives us just what we need to add common behaviors to our classes that are shared across all instances.

Our First Prototype Function

We can create our own prototype functions (methods). To do this, get `Animation.prototype` and add a new item to it, like this:

:javascript:

```
Animation.prototype.elapsed = function() {  
    return this._elapsed;  
};
```

Remember when we said that semicolons were important and should never be left off? This is one of those places where they are routinely forgotten and can in some circumstances cause a lot of debugging pain. When creating prototype functions, you really want to have semicolons. This is *particularly* true in the browser, where scripts are frequently included *verbatim* and your missing semicolon can cause a bug to appear to be in *someone else's code* because it came after yours. It's sad but true. There are ways to mitigate this (like always using an immediate function to contain your code) and those tricks are common for a reason. You will see them used all over the web if you know what to look for.

Semicolon reminders are always worth repeating: you really need them here, and this is a place where they are routinely forgotten. A prototype function is set using an anonymous function, which is an expression, and needs to be terminated with a semicolon.

This simple function will now exist for any animation we create using `new Animation()`. Again, when trying to find functions, JavaScript searches the current object, and then searches its prototype if not found, setting `this` accordingly. This means that functions on the prototype will be called even though they aren't strictly part of the object itself. We can therefore now do this:

:javascript:

```
var a = new Animation();  
console.log(a.elapsed());
```

Even though `elapsed` is not part of the instance stored in `a`, it is still found and called correctly because it is part of its prototype. Since `a` is on the left of the dot, it will be the value of `this` when `a.elapsed()` is called.

Of course we could also have just done this:

:javascript:

```
var a = new Animation();
console.log(a._elapsed);
```

But that would be terribly impolite and perhaps even a bit dangerous. The author of `Animation` decided that `_elapsed` was an implementation detail, and therefore that it might disappear or change meaning any time. It is not part of the **public interface** to the class, so we should not give in to the temptation to use it.

A Rather Large Wart

Let's create a function for requesting an animation frame and use it in a `start` function, since we will be needing one of those:

:javascript:

```
Animation.prototype._requestFrame = function() {
    var that = this;
    this._frame = requestAnimationFrame(function(t) {
        that._tick(t);
    });
};

Animation.prototype.start = function() {
    if (!this._frame) {
        this._requestFrame();
    }
};
```

The `start` function looks straightforward enough, but what is going on with the `_requestFrame` function? That seems overly complicated for something that used to just be `requestAnimationFrame(tick)`, doesn't it?

This is where a rather large wart on JavaScript becomes evident. There is an unfortunate convergence of facts that requires the `var that = this;` idiom above (a very, very common one):

- **Every** function declared using the `function` keyword has a `this` variable. Every single one, even anonymous functions.
- A function does not know about its instance unless directly called on it through dot notation.

Thus, if we tried to say `requestAnimationFrame(this._tick)`, it would not work. That function is found on the prototype, but `this` is assigned when it is called, not when it is created, which means that the browser will (eventually, during an animation frame event) call it without an instance, and it therefore, quite unhelpfully, gets the global `window` object as `this` instead of what we want it to have.

Not only that, but we have to save the outer `this` into `that` because the inner anonymous function gets its very own `this` that shadows the outer one. Double trouble.

Variables that are set as arguments at *call time* and *implicitly present* within a function are said to be in the **dynamic scope**, which as you can see, is a lot more magical and confusing than the **lexical scope** that we are used to. You can't tell where they are set by visually examining the surrounding scopes of the function, but instead have to know *how it was called*. There is a good reason that most languages went to lexical scoping after Lisp's brief and ill-fated flirtation with default dynamic scopes: they are hard to reason about. Anything that is hard to reason about is a potential source of subtle bugs.

This trips people up all the time and is sadly not really fixable because this ugly behavior is actually relied upon in many situations. There have been multiple bandages put onto it, like the `bind` method of functions, but they are just that: bandages. The patient is sadly chronically and, to some extent, incurably ill.

That gives rise to the `that = this` pattern and the use of a closure where it seems unnecessary. Here is the above code, heavily annotated with comments:

:javascript:

```
Animation.prototype._requestFrame = function() {
  // Save our "this" into the lexical scope,
  // under a name that won't be shadowed.
  var that = this;
  // Create an anonymous function that closes
  // over "that". Pass that closure into
  // requestAnimationFrame.
  requestAnimationFrame(function(t) {
    // Since we're calling this *directly* on
    // "that", the _tick function will have
    // its own "this" set to our "that", which
    // is the "this" of _requestFrame, and
    // is thus exactly what we want.
    that._tick(t);
  });
};
```

But there is hope with ECMAScript 2015: **arrow functions** do not have an automatic `this` variable, so they are saner to use (no dynamic scope!). They are also more concise, so they are popular and nice to work with. We are not going to get into much of ECMAScript 2015 or later because wow, it adds a lot to the language and we're just trying to learn the basics, here (it does make the language far nicer to use, though). But here is how you would avoid using the `var that = this;` idiom above in modern browsers, which all support arrow functions:

:javascript:

```
Animation.prototype._requestFrame = function() {
  this._frame = requestAnimationFrame(
    t => this._tick(t));
};
```

The gist of this syntax is function parameters go on the left of the arrow (use parentheses if there are zero arguments, or more than one), and what the function does (and returns) on the right, where braces can be used if more than one line is needed in the body. It is reminiscent of the function boxes that we showed earlier: we take inputs and follow the arrow to function outputs. The entire arrow function expression is just this part: `t => this._tick(t)`. That's the arrow function. Parameters on the left, body on the right.

And with that, requesting an animation frame is nice and simple, so we could just inline it instead of having a special `_requestFrame` method, like this:

:javascript:

```
Animation.prototype.start = function() {
  if (!this._frame) {
    this._frame = requestAnimationFrame(
      t => this._tick(t));
  }
};
```

There is more to these kinds of functions, but we will stop here. We will use them to avoid the “dynamic `this`” trap going forward, though. They are too nice to ignore for that.

The Rest of the Class

With that out of the way, you should be able to understand what the rest of `this` stuff is doing. Let's look at a larger piece of it in context:

:javascript:

```
function Animation(config) {
  this._move = config.move || function() {};
  this._draw = config.draw || function() {};
  this._done = config.done || function() { return false };

  this._EXPECTED_RATE = 1/60;
  this._lastTime = 0;
  this._elapsed = 0;
  this._running = false;
}

Animation.prototype._tick = function(t) {
  if (this._done()) {
    this._frame = null;
    return;
  }
}
```

```

    this._frame = requestAnimationFrame(
      t => this._tick(t));

    var dt = (t - this._lastTime) / 1000;
    if (dt > 4 * this._EXPECTED_RATE) {
      dt = this._EXPECTED_RATE;
    } else if (dt < 0) {
      dt = 0;
    }
    this._elapsed += dt;
    this._lastTime = t;

    this._draw();
    this._move(dt);
  };

  Animation.prototype.elapsed = function() {
    return this._elapsed;
  };

  Animation.prototype.running = function() {
    return !!this._frame;
  };

  Animation.prototype.start = function() {
    if (!this._frame) {
      this._frame = requestAnimationFrame(
        t => this._tick(t));
    }
  };

  Animation.prototype.pause = function() {
    if (this._frame) {
      cancelAnimationFrame(this._frame);
      this._frame = null;
    }
  };

  Animation.prototype.togglePaused = function() {
    if (this.running()) {
      this.pause();
    } else {
      this.start();
    }
  };

```

There are no surprises here, hopefully. Instead of passing closures out of a function, we define functions on a prototype, and rely on `this` to get us access to the object we care about.

Using Instances With Events

To use our new `Animation` class, we create it with `new` and start calling stuff on it. For example, to make a simple animation, our code might look like this:

:html:

```
<canvas id="drawing" width="600" height="300"></canvas>
<script src="animate.js"></script>
<script>
var canvas = document.getElementById('drawing'),
    ctx = canvas.getContext('2d');

var x = 0, vx = 100;

var animation = new Animation({
  move: function(dt) {
    x += vx * dt;
  },
  draw: function() {
    ctx.clearRect(0, 0, canvas.width, canvas.height);
    ctx.fillRect(x, canvas.height/2, 15, 15);
  },
  done: function() {
    return x >= canvas.width;
  },
});

canvas.addEventListener('click',
  () => animation.togglePaused());

animation.start();
</script>
```

Again, it is very important that we use `new Animation()` instead of just calling `Animation()` without `new`: that makes it create an instance of our class.

Our animation object looks much the same as it ever did (but it doesn't start automatically—so we call `animation.start()` manually). We can call `animation.pause()` and other functions on it just as before. Where things get weird is when we try to use `togglePaused` in an event listener:

:javascript:

```
canvas.addEventListener('click',
  () => animation.togglePaused());
```

It used to be that we could just say `canvas.addEventListener('click', togglePaused)`, passing a function directly into the listener registration, but now it is a prototype function that depends on `this`, so that will not work for all of the reasons we've already discussed.

Again, the arrow function has parameters on the left (none), body on the right: `() => animation.togglePaused()`. That expression creates an actual function.

The dynamic `this` variable is the reason you will often see arrow functions that only do one thing. You need to get the instance (here it is `animation`) into a closure to call it properly (it has to be on the left of a dot during the call), or it is lost. Be wary of that anytime you need to pass a prototype function into another function, always wrap it in a closure, and you might as well get comfortable doing so as an arrow function: it provides `this` safety as well as being more succinct.

There is yet another approach to doing this because all functions have a `bind` method that binds `this` properly (among other things):

:javascript:

```
canvas.addEventListener(  
  'click', animation.togglePaused.bind(animation));
```

That isn't necessarily clearer and it does essentially the same thing as the arrow function. It is basically creating a new closure with `this` **bound** to the thing we pass into `bind`. The thing we just referenced on the left of the dot. It also stutters. Just be aware that it exists so that you understand it when you see it.

But Why?

With all of the extra effort we have to go to just to get events working right when using prototype-based classes in JavaScript, why should we even do that instead of just passing out closures like we did for most of this course? We have, after all, talked an awful lot about favoring clarity over efficiency, and this seems like a place where that argument has a lot of merit.

There are various reasons given for preferring prototypes for behavior rather than an object full of closures, including things like enumerability, memory efficiency, tooling to aid the developer, a mechanism for inheritance, and others. The main reason I keep using prototype-based objects is actually a lot simpler than all of these reasons: it is **standard**.

There is more than one dimension along which clarity matters. We have focused a lot on obviousness and clarity through the use of homogeneous structures for our code, and that really does make a lot of sense: once you have learned what a function does, you can suddenly use it to create incredibly interesting and complex software and you really don't need much else. That is the basic idea behind a lot of Lisp and Lisp-inspired languages (including Racket, which is even simpler underneath), and there is undeniably power in that way of thinking.

However, while the prototype-based approach in JavaScript has power of its own that we have not explored in this course, the main reason to become acquainted with and make liberal use of it where it makes sense is that anyone familiar with standard JavaScript idioms will look

at your prototypes and immediately know what they are doing. Because everyone does it this way, using this approach lends a certain amount of readability to your code by default. Every language has its syntax and vocabulary, but a huge chunk of communication is accomplished through agreed-upon cultural idioms. That is also true of programming languages.

Furthermore, as alluded to earlier, the rise of ES6 (ECMAScript 2015) and later provides language support for classes. That support is basically a wrapper around the more manual prototype definition that we have shown here, so it is worth getting at least a little bit comfortable with it.

Here, for your viewing pleasure, is our class defined using the new syntax, including object unpacking with default empty arrow functions. You will immediately notice that there is a lot less repetition:

:javascript:

```
class Animation {
  constructor({move=()=>{}, draw=()=>{}, done=()=>false}) {
    this._move = move;
    this._draw = draw;
    this._done = done;
    this._EXPECTED_RATE = 1/60;
    this._lastTime = 0;
    this._elapsed = 0;
    this._frame = null;
  }

  _tick(t) {
    if (this._done()) {
      this._frame = null;
      return;
    }
    this._frame = requestAnimationFrame(
      t => this._tick(t));

    var dt = (t - this._lastTime) / 1000;
    if (dt > 4 * this._EXPECTED_RATE) {
      dt = this._EXPECTED_RATE;
    } else if (dt < 0) {
      dt = 0;
    }
    this._elapsed += dt;
    this._lastTime = t;

    this._draw();
    this._move(dt);
  }

  elapsed() {
    return this._elapsed;
  }
}
```



```

    }

    running() {
      return !!this._frame;
    }

    start() {
      if (!this._frame) {
        this._frame = requestAnimationFrame(
          t => this._tick(t));
      }
    }

    pause() {
      if (this._frame) {
        cancelAnimationFrame(this._frame);
        this._frame = null;
      }
    }

    togglePaused() {
      if (this.running()) {
        this.pause();
      } else {
        this.start();
      }
    }
  }
}

```

That really is a lot clearer, isn't it? And it is essentially the same—with benefits—as what we did before; it just makes prototype-based classes a lot nicer to create and read.

Keep it Classy

When developing code, I usually start with closures because I don't often have the whole picture in mind when I begin. As it becomes clear that an object is getting complex, and there is more and more data to keep track of, I switch to a class.

It is perfectly fine to not know how to organize a program from the very beginning. This is essentially a creative process, and as such it will evolve as your understanding changes. Many programs end up being object-oriented at some level eventually, though, and class syntax makes it pretty easy to take things in that direction, keeping related data and behaviors together.

That's really what this is about. Functions are a wonderful, powerful abstraction, but an abstraction is only as useful as the data it operates on. If a function really is specific to a particular set of data, there is no reason for it to exist outside of that environment. Classes

make expressing that idea straightforward and natural.

Exercises

Exercise 14-1: Lab: Create a Segment Class For the Snake Game

Solution on page 478

We just finished making our snake game work really well, and now it's time to change the implementation without hurting game play. For this lab, you will create a new `Segment` class that accepts `x` and `y` coordinates and keeps track of them. Each instance of this class will have three methods:

- `move(dir)`: moves the segment in the direction specified,
- `copy()`: returns a copy of the segment, and
- `collides(other)`: returns `true` if this instance is at the same position as `other`.

Once done, make all segments into instances of this class and change your code to use these new methods where possible.

Chapter 15

Practical Web Programming

We have come a long way!. We covered a lot of the surface area of JavaScript, including the most important syntax, structures, and even a handful of useful program design techniques. We wrote simple animations and games together, and in the process we explored some ideas about how to write library code and generate useful abstractions.

We have admittedly been writing code in a way that is not very common. Our HTML files, for example, do not have any of the usual tags in them. Our scripts are essentially running inside of the document body, and we are relying on browser default behavior. That is not often the best plan.

In this section we are going to get things a little bit more mainstream and introduce some good practice with document structure and organization, as well as hit a few miscellaneous bits that we have missed along the way.

You can write working programs without the tools in this part of the course, but you will be swimming upstream if you continue with some of the simplifying practices we have embraced until now. There is no reason to do that with the knowledge you have!

A More Complete Document

HTML documents have two major sections, and we have generally only been using one (implicitly): the body. The other section is called the “head”, and it contains setup for the document. Included in this setup is nearly always the JavaScript that the page will be using.

What we have been doing, with our bare script tags (often after a canvas or button tag), has basically been this:

:html:

```

<!DOCTYPE html>
<html>
<head>
</head>
<body>
  <canvas id="drawing" width="600" height="300"></canvas>
  <script src="animate.js"></script>
  <script>
    // Our program goes here.
  </script>
</body>
</html>

```

All of those other tags, including the `html` and `head` tags, have been implied. Browsers try to be pretty lenient with what they accept, but if you don't specify what the document contains, sometimes you will be surprised at how the browser interprets it. We have been very careful to keep things simple enough in this course that the default interpretation stayed predictable, but it wouldn't have been difficult to stray into dangerous territory (where “dangerous” means “every browser does it differently” or “behavior is unspecified and may change with another browser update”).

For example, if we had not used an HTML `canvas` tag in our game programs, instead relying on JavaScript to create and insert the tag (via `document.createElement` and `document.body.appendChild`), they would not have worked at all. When there is no tag other than a `script` tag, the browser assumes you meant it to go in `head` instead of in `body`, and that means that there is no body to append a canvas to when the script runs. More is said about this process below.

Hopefully you didn't run into that, since all of the examples did it as shown above, with an actual `canvas` tag, but you can see how this could get brittle if we weren't explicit about things. From now on, it's best practice to specify *all* of these essential tags in your document.

Head Scripts

Script tags often go inside of the head, rather than the body, but moving them there introduces a bit of a wrinkle: they run *before the document elements exist*! That means, for example, that the following code will simply fail:

:html:

```

<!DOCTYPE html>
<html>
<head>
  <script src="animate.js"></script>
  <script>(function() {
    // THIS WILL NOT WORK!
    var canvas = document.getElementById('drawing'),

```

```

        ctx = canvas.getContext('2d');
    }());</script>
</head>
<body>
    <canvas id="drawing" width="600" height="300"></canvas>
</body>
</html>

```

Can you see why? The script that is asking the document for the “drawing” element is running before that element has been created, below. Therefore, the `getElementById` function will just return `undefined`. The `undefined` value has no member called `getContext`, so you will get an error in the console to that effect. Oops. We will talk about how to fix this using an event, later on.

We do have one safety feature in here, and that is the **immediate function** in the inline script: a function that is defined and called in one step. We can also avoid global variable mistakes by putting `"use strict";`¹ at the beginning of our immediate function. It’s a special directive that tells the interpreter that we should only ever allow assigning to existing variables, either in the global scope via `window` or something we have declared with `var`. That’s a nice safety feature in case we forget `var` somewhere; we will get an error instead of silently polluting the global scope.

:html:

```

<script>(function() {
    "use strict";
    // Our code goes here.
})();</script>

```

For our “animate.js” file, we should probably use something like that. Then we can just create one global object that contains all of our functions. At the bottom we could explicitly export only the stuff we want to, by setting properties on `window` (the global browser object), which is what `this` is set to when not called on a class instance:

:javascript:

```

// animate.js
(function() {
    "use strict"; // do not allow undeclared variables

    // functions from earlier animate.js go in here

    // Note: global scope variable on purpose.
    this.AnimateJS = {
        animate: animate,
        fillCircle: fillCircle,
        Animation: Animation,
    }
})();

```

¹https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode

```
};
})(); // end immediate function
```

This takes our existing functions `animate` and `fillCircle` and puts them into an `AnimateJS` object. It also puts the `Animation` constructor in there. Now when we load our script, it will expose just one global variable: `AnimateJS`. We can then access all of the needed functionality by calling its fields, e.g., `AnimateJS.animate(...)`. Several popular libraries do exactly this, like `jQuery`, which puts much of its functionality into a global variable named `$`. Similarly, `underscore` (or `lodash`) uses the `_` variable to hold its functionality.

One thing to pay attention to: we use the `this` variable inside our immediate function because that variable is always the global scope object when not called on a method (or when inside an HTML tag event handler string). Our immediate function is definitely not a method: there's no dot for something to be to the left of, when it is called.

Why not use `window`? Well, you can, and it will work fine in the browser. There are other places that JavaScript can run, however (like `Node.js`²), and there might not be a `window` object. No matter, what else happens, though, if a naked function is called without an object context, `this` is assigned the global scope object.

We are starting to overlap a bit with the idea of modules in JavaScript. The current ecosystem is full of highly opinionated libraries that all work a bit differently, unfortunately. I will refrain from expressing my own opinions, other than to say that modules are useful and so are private scopes.

All of this is like strapping in before driving, but we still have a problem that keeps us from getting anywhere: our “main” code will still run before the document has a body. In order to solve this while still leaving our code in the `head` of our HTML document, we need to be notified when the document has finished loading. We need the `DOMContentLoaded` event:

:html:

```
<html>
<head>
  <script src="animate.js"></script>
  <script>(function() {
    "use strict";

    function main() {
      // After the document loads, we can see the canvas!
      var canvas = document.getElementById('drawing'),
          ctx = canvas.getContext('2d');

      ctx.fillRect(10, 10, 20, 20);
```

²nodejs.org

```

        AnimateJS.fillCircle(ctx, 50, 50, 10);
    }

    // Capitalization matters. This is the event's name.
    document.addEventListener('DOMContentLoaded', main);
}());</script>
</head>
<body>
    <canvas id="drawing" width="600" height="300"></canvas>
</body>
</html>

```

Well, gosh. If we’re doing it that way, we can actually move our entire program into another file like we did with “animate.js”. Then we have this:

:html:

```

<html>
<head>
    <script src="animate.js"></script>
    <script src="program.js"></script>
</head>
<body>
    <canvas id="drawing" width="600" height="300"></canvas>
</body>

```

And our “program.js” file would look like this:

:javascript:

```

// program.js

(function() { // private scope
    "use strict";

    function main() {
        var canvas = document.getElementById('drawing'),
            ctx = canvas.getContext('2d');

        ctx.fillRect(10, 10, 20, 20);
    }

    // Get started right away once the document loads.
    document.addEventListener('DOMContentLoaded', main);
}());

```

This is a common pattern for libraries to use when they need to do some setup. In general, it’s nice to keep things separated into different files like this. It keeps things easier to manage. JavaScript code generally goes into “.js” files, HTML code goes into “.html” files, and CSS (which we are kind of staying away from for this course, but they are loaded with `link`³ tags)

³<https://developer.mozilla.org/en-US/docs/Web/HTML/Element/link>

goes into “.css” files. JavaScript is for behavior, HTML is for basic layout and structure, and CSS is for styling. The lines between these things get blurry sometimes, but having this in mind can really help as you design your programs.

In real life, of course, we would probably not hard-code the word “drawing” in our function. We would instead provide a configuration function or constructor that would allow callers to tell us which element to draw on.

Adding Elements Dynamically

Until now we have been relying on the `canvas` HTML tag to create a canvas for us, but it can actually be created in JavaScript and added dynamically (speaking of blurring the lines). This can be useful if you’re creating an animation widget and you just want the user to tell you where to put it. Then you have a lot of control over how it is created and used.

To create a new element, you use the `document.createElement` function, and then you use the `appendChild` function of the element you want it to live inside of (there are also `before` and `after` functions for inserting your new element next to another one in the DOM). For example, to create a canvas like what we have been using, you could use the following JavaScript:

```
:javascript:
```

```
var canvas = document.createElement('canvas');
canvas.width = 300;
canvas.height = 300;

document.body.appendChild(canvas);
```

From that point on, the canvas is visible in the page and you can access it like you would have if you had specified it in HTML and gotten it by (for example) ID.

This is what we did for the “fake console” setup near the beginning of the course: create a `div`, append it to the document, and create a `console.log` function in the global scope that writes text to that `div`.

This works with any of the HTML elements. It is a useful thing to know about. You will see it used by other libraries, but we obviously don’t use it in this course. The key thing to remember is that the canvas created by `document.createElement` *is not in the document until you add it with `appendChild`*. You can change settings and draw to it and everything, but it will not be visible until it is part of the document.

Countdown

Let's put these new ideas into practice by building a countdown timer to whatever your favorite date is. We will allow it to be specified in a text box using an `input` tag, and we will display the result in a `span` tag that shows the countdown changing every second.

First things first, we will create a basic `index.html` file in a new folder:

:html:

```
<html>
<head>
  <title>Countdown</title>
  <script src="countdown.js"></script>
</head>
<body>
  Target day (mm-dd):
  <input type="text" id="date" size="5" maxlength="5"> :
  <span id="counter"></span>
</body>
</html>
```

The HTML shown here adds a text input⁴ field and a `span`⁵ that we will use to display the number of seconds left until the big day, whatever day that is. Now for the “countdown.js” file we haven't written, yet:

:javascript:

```
(function() { // private scope
  "use strict";

  function showCounter(elTarget, elCounter) {
    // TODO:
    // Compute the seconds left until the value in
    // element elTarget and display it in elCounter.
  }

  function main() {
    var elTarget = document.getElementById('date'),
        elCounter = document.getElementById('counter');

    // Set a default value for it (next year):
    elTarget.value = "01-01";

    // Every second, show the counter.
    setInterval(() => {
      showCounter(elTarget, elCounter);
    }, 1000);
  }
})
```

⁴<https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input>

⁵<https://developer.mozilla.org/en-US/docs/Web/HTML/Element/span>

```
document.addEventListener('DOMContentLoaded', main);
})();
```

The `main` function that starts when the "DOMContentLoaded" event fires is found at the bottom, and we put all of the supporting code above it.

We know we will need something that happens every second, so we use `setInterval`⁶ to call a function once per second. That function is not yet written, but it will be, shortly:

The `main` function simply sets a default for our target month and day, and then starts the interval ticking that will figure out what date we care about and display the time remaining.

Dates

To get the number of seconds until a particular date, we first need to convert the text in the target input into a `Date`⁷ object. Then we can use that to make calculations. There are many ways to create a date in JavaScript, but first let's talk about how to get the month and day out of the text field.

We are going to be pretty brittle with this, because parsing dates is hard in general and you usually just want to use someone's library for that. Since we are explicitly asking it to be specified as "mm-dd" format, we will just assume that's exactly what we have. With a "mm-dd" string (like "04-05" for April 5), we can get a full date by adding a year in front of it, like this: "2050-04-05". The `Date` constructor knows how to parse that kind of a date, so all we have to do is provide it with one.

We need a year. Fortunately, that's easy: if we call `new Date()` with no arguments, it returns the current date, from which we can get the 4-digit year like this:

:javascript:

```
var now = new Date();
var target = new Date(now.getFullYear() +
    '-' + elTarget.value);
```

This gives us the current time in `now` and the target time this year in `target`.

There is one more wrinkle that we need to figure out before we can move on, and that is the fact that the date might be in the past. If today is July 1 (07-01), and our target is January 5 (01-05), then putting the current year on it will give us a day that already happened. We can hardly use that as a calculation. Fortunately, if we treat our date as a number (say, by prefixing it with +), then we will get the number of milliseconds since January 1, 1970 UTC. We can use that to determine if our date is in the past. If it is, we will just add 1 to its year.

⁶https://developer.mozilla.org/en-US/Add-ons/Code_snippets/Timers

⁷https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date

When we are finished, we can get the seconds by subtracting `now` from `target`, dividing by 1000 to get seconds instead of milliseconds, and taking the floor. Then we display it as the `innerText` of our counter `span` element:

:javascript:

```
function showCounter(elTarget, elCounter) {
  var now = new Date();
  var target = new Date(now.getFullYear() +
                        '-' + elTarget.value);
  // Is our target in the past? Add one to the year.
  // That will fix it.
  if (+now > +target) {
    target.setFullYear(target.getFullYear() + 1);
  }

  var seconds = Math.floor((target - now) / 1000);
  elCounter.innerText = seconds;
}
```

This actually works. If you run the program now, you will see a counter of seconds remaining until next January 1. You can change the date in the box, and things will kind of get upset while it is not valid, but once you enter a real date, it will work fine. You might have noticed that it shows `NaN` if you have an invalid date. We can actually detect that using the built-in⁸ `isNaN` function⁹ and just not display anything, like this:

:javascript:

```
var seconds = Math.floor((target - now) / 1000);
if (isNaN(seconds)) {
  seconds = '';
}
elCounter.innerText = seconds;
```

Note, though, that the `Date` object is forgiving about missing days. It will allow you to specify only a month, if you want.

With that, we are done, right? Except that the display is pretty hard to read and quite possibly not what you had in mind when you thought of a countdown timer. Perhaps it would be better to show something like days, hours, minutes, and seconds, separated by colons. That takes a little more doing, but it is absolutely possible. Let's create a new function to calculate it (and since we are going to be doing a bunch of time calculations anyway, we will just give it two dates).

To compute minutes and seconds, what we want is the number of minutes in a large number of seconds, and then we want the remaining seconds. If, for example we have 92 seconds, that should translate to "1 minute, 32 seconds". To get 1 minute, we just divide by 60 and

⁸https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects

⁹https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/isNaN

take the floor. That gives us the number of minutes that fit into our total seconds. But then we want the remainder to get the number of seconds remaining. That means we want the `%` modulus operator. Here's an example using the console:

:console:

```
> Math.floor(92 / 60)
< 1
> 92 % 60
< 32
```

What we will do is apply this same logic over and over again to get all of the components we want:

- Divide seconds by 60 to get minutes, mod by 60 to get remaining seconds.
- Divide minutes by 60 to get hours, mod by 60 to get remaining minutes.
- Divide hours by 24 to get days, mod by 24 to get remaining hours.

And then we are done. Here is what that looks like:

:javascript:

```
function friendlyDuration(from, to) {
  var seconds = Math.floor((to - from) / 1000);
  if (isNaN(seconds)) {
    return '';
  }

  // We now have seconds, and need to calculate all
  // of the days/hours/minutes/seconds components.
  var minutes = Math.floor(seconds / 60);
  seconds %= 60;
  var hours = Math.floor(minutes / 60);
  minutes %= 60;
  var days = Math.floor(hours / 24);
  hours %= 24;

  // And now we have days, hours, minutes, and seconds.
  return days + ":" + hours + ":" + minutes + ":" + seconds;
}
```

There is a bit of a problem, though. If we have 3 days, 2 hours, 20 minutes, and 5 seconds, it will display like this: "3:2:20:5". That's a bit jarring. What we would expect would be something more like "3:02:20:05". The leading zeros are missing.

Or what about this: 3 minutes, 25 seconds. That will currently read as "0:0:3:25". We really don't need the days and hours in there for this short a time. They should not display at all! It looks like we need more logic in our string-formatting code. Let's fix this by creating a small function in here that pads numbers with leading zeros if necessary, and only adds them if needed. The change is at the bottom of the now-familiar value computation:

:javascript:

```

function friendlyDuration(from, to) {
  var seconds = Math.floor((to - from) / 1000);
  if (isNaN(seconds)) {
    return '';
  }

  // We now have seconds, and need to calculate all
  // of the days/hours/minutes/seconds components.
  var minutes = Math.floor(seconds / 60);
  seconds %= 60;
  var hours = Math.floor(minutes / 60);
  minutes %= 60;
  var days = Math.floor(hours / 24);
  hours %= 24;

  function pad(val) {
    val += ''; // convert to a string
    while (val.length < 2) {
      val = "0" + val;
    }
    return val;
  }

  // Now create a format string with all values padded.
  // This gives us a string like 03:02:04:25.
  var output = pad(days) + ":" +
    pad(hours) + ":" +
    pad(minutes) + ":" +
    pad(seconds);

  // Check for and remove all leading instances of "00:".
  // This removes any leading zero values completely,
  // changing something like "00:00:03:13" to "03:13".
  while (output.slice(0, 3) === "00:") {
    output = output.slice(3);
  }

  // Finally, remove all leading zeros on the leftmost
  // segment. This would change "03:13" to "3:13".
  while (output[0] === "0") {
    output = output.slice(1);
  }

  return output;
}

```

Tricky, right? Making this human-friendly typically is, but it is well worth the effort. If you run this program, you will see a countdown timer exactly like you might expect. It shows days, hours, minutes, and seconds, and it never has a leading zero on the left side, but it always shows leading zeros on the internal segments. That is kind of fun!

We used a string method called `slice`¹⁰ (yes, using a regular expression¹¹ would make some of this easier. No, I really did not want to include a chapter on those) to accomplish some of this. It returns a smaller part of the string, starting at the first index and going up to (but not including) the second index. Thus `"01234567".slice(3, 6)` will evaluate to `"345"`. If no second index is specified, it goes all the way to the end of the string, so `"0123456789".slice(5)` evaluates to `"56789"`.

Summary and Conclusions

Well, that was quite the adventure! There is a lot of trail ahead, of course. The world of HTML and JavaScript alone is huge and full of interesting things to do, and there is no way we could even cover a significant fraction of what is possible in a single introductory course. The goal of this course has been to go from zero knowledge to writing programs over the course of a full-credit semester, and I hope that you have found it useful and mildly entertaining at times. If you have made it this far, have done the exercises and the labs, and can basically understand all of the listings at the ends of the recent chapters, you have really crossed an important threshold: you can write your own software.

JavaScript is not a perfect language, but it really is everywhere. It runs in the browser, it runs on servers, and it is hard to avoid in general. There are other wonderful and even much better languages out there (depending on what you are doing), and many of them also run in the browser now (commonly by transpiling either to JavaScript or WebAssembly), so even if all you have is a machine that accesses the internet, you can try a lot of them out.

The world is yours. Go explore!

Listings

Full listings follow. The HTML is much shorter than usual because now we are loading our code from a JavaScript file that registers with the “DOMContentLoaded” event. Is this always the right thing to do? No, it’s merely a tool for your growing toolbox. There are always tradeoffs, and the best way to learn about them is to tinker and to explore other people’s code. There is a lot of it out there!

But this is ours:

index.html

:html:

¹⁰https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/slice

¹¹https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp

```

<html>
<head>
  <title>Countdown</title>
  <script src="countdown.js"></script>
</head>
<body>
  <input type="text" size="5" maxlength="5" id="date"> :
  <span id="counter"></span>
</body>
</html>

```

countdown.js

:javascript:

```

(function() { // private scope
"use strict";

function friendlyDuration(from, to) {
  var seconds = Math.floor((to - from) / 1000);
  if (isNaN(seconds)) {
    return '';
  }

  // We now have seconds, and need to calculate all
  // of the days/hours/minutes/seconds components.
  var minutes = Math.floor(seconds / 60);
  seconds %= 60;
  var hours = Math.floor(minutes / 60);
  minutes %= 60;
  var days = Math.floor(hours / 24);
  hours %= 24;

  function pad(val) {
    val += ''; // convert to a string
    while (val.length < 2) {
      val = "0" + val;
    }
    return val;
  }

  // Now create a format string with all values padded.
  var output = pad(days) + ":" +
    pad(hours) + ":" +
    pad(minutes) + ":" +
    pad(seconds);

  // Check for and remove all leading instances of "00:".
  while (output.slice(0, 3) === "00:") {
    output = output.slice(3);
  }
}

```

```

    }

    // Finally, remove all leading zeros.
    while (output[0] === "0") {
        output = output.slice(1);
    }

    return output;
}

function showCounter(elTarget, elCounter) {
    var now = new Date();
    var target = new Date(now.getFullYear() +
                           '-' + elTarget.value);
    // Is our target in the past? Add 1 to the year.
    if (+now > +target) {
        target.setFullYear(target.getFullYear() + 1);
    }

    elCounter.innerText = friendlyDuration(now, target);
}

function main() {
    var inputTarget = document.getElementById('date'),
        elCounter = document.getElementById('counter');

    // Set a default value for it (next year):
    inputTarget.value = "01-01";

    // Every second, show the counter.
    setInterval(function() {
        showCounter(inputTarget, elCounter);
    }, 1000);
}

document.addEventListener('DOMContentLoaded', main);
}());

```

Exercises

Exercise 15-1: Lab: More Interesting Counters

Solution on page 482

For this final lab, you will add 2 or 3 more counters to the page and allow them to be named using input fields. All of them should work simultaneously.

For this, a few little details might trip you up.

First of all, if you are in a loop and create a closure based on variables in that loop, you are going to have a bad time. Check this out:

:javascript:

```
for (var i = 0; i < 3; i++) {  
  setInterval(function() {  
    console.log("I'm the " + i + " value");  
  }, 1000);  
}
```

What is this attempting to do? It's trying to set up three intervals, each of which outputs something different to the console. If you run it, you will notice that every single one of them says "I'm the 3 value". Oops, that's not what we wanted!

The problem is this: our little function closes over the `i` variable, but that variable just changes value each time through the loop. It is not a new variable. So, by the time our functions are run (one second later), that value is 3 for all of them. They all close over the same variable.

Ugh. How do we fix that? Well, you can use `let` instead of `var` if you are in a browser (or other environment) that supports it. That has much saner behavior in these circumstances and creates new, scoped variables in a more natural way. If you don't have that option, you can create a function that does this registration for you, effectively giving your closure new variables to close over:

:javascript:

```
function register(i) {  
  setInterval(function() {  
    console.log("I'm the " + i + " value");  
  }, 1000);  
}  
  
for (var i = 0; i < 3; i++) {  
  register(i);  
}
```

That works. The reason is that the `register` function is called immediately each time through the loop, and each function call gets its own variables to play with. Thus, our completely unchanged closure now works properly because it is closing over new variables each time.

Why do you need this? You might not. There are plenty of ways to do this lab without knowing this rather subtle detail, but at least one of the ways, the one that I'll show in the answer key, needs to handle this case properly.

Final Exam

Write your own game! You have all of the tools. Imagine something simple and fun, and write it.

Chapter 16

Answer Key

Solutions

Chapter 1 Solutions

Exercise 1-1: Create a new programming session

Practice starting a new programming session from scratch. Close all of your program tabs, then

- Start your text editor.
- Write “Testing my sanity” or some other recognizable and pithy comment, and save it as an HTML file.
- Load that file in the browser.

Answer 1-1

Following the instructions above should lead you to a page that shows your pithy comment. If you made it unique (you have never used it before in the previous examples), then you can be sure that you have done the exercise correctly.

For example, if your comment was “this is a brand new file, just for the exercises”, then you should see exactly that text in the browser window.

If you do not see it, here are a few things to consider:

- Can you find your HTML file in the file manager?
 - If you can see your file, what happens when you select it to open it? (Often this is done by double-clicking on it.) If it opens in the browser and you see the expected text, you’re all set.
 - If you cannot see it HTML file, there are a couple of possibilities.
 - * Perhaps the file was not saved. Try going back to your editor and saving the file.

- * The file might not have been saved *where you thought it was*. Try searching for it. If you gave it a unique name, it will be easier to find.
- If you cannot find your new folder in the file manager, you might need to start over and try again. Find a place for the new file that is memorable for you. Create something where you can find it easily next time and give it another shot.

Exercise 1-2: Practice loading the console

Once you have successfully loaded your new program (with your new statement inside of it), open the Developer Tools and find the console.

Answer 1-2

Each browser is different in this respect. Take a close look at the instructions for your particular browser, given earlier in this chapter.

The console has a place where you can type JavaScript expressions and have them evaluated immediately when you press the return/enter key. If you see a relatively blank area and a place to type things (either at the bottom of it, as is the case with Firefox, or right inside of it, like in Chrome), then you have found it. Try typing in a command to be sure, something like `2 + 2`. If it gives you the answer, you have definitely found the console.

Exercise 1-3: Find the source viewer

When you write the text of a program, you are writing what is called “source code”. This means your code is the “source” of all of the instructions that the computer will execute on your behalf. Find the developer tool that lets you view the source of the file that the browser has currently loaded.

Note that this is not the “View Source” menu item that you might be familiar with. Rather, this is the *developer tool* that shows you the source and gives you tools to work with it.

Answer 1-3



If you successfully found the console above, then you are probably seeing a window with a bunch of tabs at the top or along the side. One of these will either say “Sources” (Chrome) or “Debugger” (Firefox). If you select the appropriate tab, you will see the actual text you entered into your file, along with some tools to work with it.

We will not be doing anything with those tools just yet, but they will appear in later chapters. The key is to know how to get to them.

Exercise 1-4: Faster developer tools access

Developer tools are incredibly important and useful, particularly for this course, so we are going to want to be able to get to them quickly. Find the combination of key strokes that opens (and closes) the console and practice them. Memorize them.

Answer 1-4

To find the magic keystrokes, you can look in the menu that lets you find the developer tools in the first place. For example, for Chrome on a Mac, the keys that open the console are **Option** + **Command** + **J**. This can be found by going to the **View** > **Developer** menu and looking at “JavaScript Console”, which in my case has the symbols “  J” next to it, indicating **Option** + **Command** + **J**.

On Chrome OS, you can find the key combination by bringing up the keyboard shortcut viewer (either in **Settings** > **Keyboard** > **Shortcut Keys**, or by typing the three keys **Ctrl** + **Alt** + **?** together) and pressing modifier keys until you see “JavaScript” console appear on the keyboard. Hint: try pressing **Ctrl** + **Shift** and you will see it on the “J” key. Therefore, the combination there is **Ctrl** + **Shift** + **J**.

On Windows and Linux the combination is usually similar to that on Chrome OS.

On Firefox, you can find the key combination by opening up the developer tools and hovering over the “Console” tab, where you will see a key combination that works there (on my system it shows up as **Option** + **Command** + **K**, meaning you hold down the option and command keys while pressing K on a Mac.

Whatever you find, it makes sense to practice it and memorize it. The console will be an old friend by the time we are through this course.

Exercise 1-5: The console as a calculator

Use the console to find the value of this expression: $1 + 2 - 3 + 4 - 5 + 6 - 7 + 8 - 9 + 10$.

Answer 1-5

Once the console is open and you have typed that expression in and hit the enter key, your session will look something like this:

```

:console:
> 1 + 2 - 3 + 4 - 5 + 6 - 7 + 8 - 9 + 10
< 7

```

Exercise 1-6: Definitions

Define the following:

- syntax highlighting
- JavaScript console
- text editor
- variable
- expression
- evaluate
- assign
- delimit
- escape character
- literal
- concatenate

Answer 1-6

- syntax highlighting: a feature of a code editor, where parts of your code are colored or styled differently based on what kinds of things they represent in the language. For example, strings may be colored differently than numbers.
- JavaScript console: a developer tool where you can type short one-line expressions and see their effects immediately.
- text editor: a program that allows you to write plain text, save it, and edit it later.
- variable: a special place in memory that you can access by name. It can hold any kind of value.
- expression: something that can be *evaluated*, that is, something that represents a computation that can be turned into a value.
- evaluate: to compute an expression, producing a value
- assign: the process of putting a value into the memory referenced by a variable. This can also create the variable at the same time, if it doesn't already exist.
- delimit: to “set apart” something, like using quotes to set apart strings, parentheses to set apart sub expressions, and semicolons to separate statements.
- escape character: the backslash \ character, used to signal special characters in strings that cannot usually be typed directly.
- literal: loosely speaking, a value that looks like what it is. A string literal is characters inside of quote marks; it is a string value sitting right there in front of you. A number literal is just a number. There are others.
- concatenate: to join two things together, as in using + to join two strings into one.

Exercise 1-7: Assignment practice

Show code that causes the variable `x` to contain the value 42.

Answer 1-7

:javascript:

```
x = 42
```

Exercise 1-8: Evaluation

Write the result of evaluating $(x - 12) * 3 + 9$ where `x` is 42 without using a computer - use your brain to figure out what the answer will be. Show your work.

Answer 1-8

The answer, as found by just typing it into the console, is this:

:console:

```
> x = 42
< 42
> (x - 12) * 3 + 9
< 99
```

That is the answer we are going for. A complete answer will show the steps taken, such as variable assignment and probably the relevant substitution for `x`, like this (use your own discretion about how much work shown is enough - understanding order of operations and variable evaluation is the key here, not pedantry):

:javascript:

```
x = 42
(x - 12) * 3 + 9
(42 - 12) * 3 + 9
30 * 3 + 9
90 + 9
99
```

Exercise 1-9: Using variables

Use a variable to compute the value of the polynomial $x^2 - 2x + 1$ where $x = 3$. Hint: you can use `x*x` to mean “x squared”.

Answer 1-9

A successful console session might look like this:

:console:

```
> x = 3
< 3
> x*x - 2*x + 1
< 4
```

What happened there? We just created a variable x by assigning $x = 3$. The console helpfully told us that we just assigned something to be 3. Then, we typed in the polynomial expression, which references x . The console computed it using our value for x and output the final answer for $3*3 - 2*3 + 1$, which is 4.

Exercise 1-10: Approximate Euler's constant

If you are using Chrome or Firefox (as they support exponentiation using the `**` operator), approximate the value of e by computing $(1 + 1/1000)^{**1000}$. What value do you get?

Answer 1-10

A successful console session might look like this:

:console:

```
> (1 + 1/1000)**1000
< 2.7169239322355203
```

A more accurate value for e is actually 2.718281828459045, but we got it to two decimal places!

Exercise 1-11: A non-console program

Write a program (in your text editor, not the console) that uses string concatenation to show "Hello!" using only strings that each contain a single character.

Answer 1-11

The general steps for creating a new program apply. The student should have done the following:

- Created a new folder,
- Written a file `index.html` into that new folder, and
- Added a program there, something like the following:

:html:

```
<script>
var msg = "H" + "e" + "l" + "l" + "o" + "!";
alert(msg);
</script>
```

The basic critical elements for this to work are the use of opening and closing `script` tags (signaling to the browser that you are going to write JavaScript) and the use of `+` to concatenate tiny strings together to make a full message.

The message doesn't need to be stored in a variable first. A perfectly acceptable solution will have zero or more variables in it. For example, this would also be valid code within the tags:

:javascript:

```
alert("H" + "e" + "l" + "l" + "o" + "!");
```

Any combination is suitable so long as the following criteria are met:

- Only one-character strings are used, and
- Semicolons are always used at the end of statements.

Exercise 1-12: Escapes in strings

Write a program to create a string that shows this emoticon `\o/` (backslash, 'o', slash) and alert it.

Answer 1-12

Because the `\` character is an *escape* character in strings, you cannot insert one into a string without doing something special. A correct answer will escape the `\` character to make it literal, like this:

:html:

```
<script>
alert("\\o/");
</script>
```

The key is the double `\\` inside the string. The first one is the escape character, which indicates that the next character will be treated *differently than usual*. The *usual* use of `\` is to escape other characters, so when it is preceded by a `\`, it is treated *literally*. Thus, while `"\\"` is *written* with two backslash characters, it actually *contains* just one, and that is what shows up in the alert.

Two backslashes `\\` are how we *communicate* to the interpreter that we want a string with a `\` inside of it, and that is what this exercise is meant to reinforce.

Chapter 2 Solutions

Exercise 2-1: Define terms

Define the terms

- formal parameter
- argument
- substitute
- evaluate
- recurrence relation
- recursive function

Answer 2-1

- formal parameter: the variable names representing things passed into a function. When defining a function like $f(x) = x^2$, the x in $f(x)$ is a *formal parameter*. It is a placeholder for a particular value of x that will be used later when evaluating the function.
- argument: the value given when evaluating a function, taking the place of one of its formal parameters. For example, in $f(x, y) = x + 3y$, the formal parameters are x and y , but when evaluating the function, say at $f(2, 3)$, the *arguments* are $x = 2$ and $y = 3$. The arguments are the actual values given when evaluating a function.
- substitute: replace a variable with its value, also known as “evaluate”.
- evaluate: produce a value from a computation definition, when all variable values are known.
- recurrence relation: a definition of a computation that references itself in some way, with different arguments. For example, a doubling series can be represented as $f(1) = 1$ and $f(n) = 2f(n - 1)$. The definition of f references itself, but with different arguments. This usually represents a series, but they can be more complicated than that. Recurrence relations are generally not complete without base cases, at least one value that is produced without self-reference.
- recursive function: a function that references itself, as in a recurrence relation.

Exercise 2-2: Writing functions

Using pencil and paper, write a function using algebra notation that represents *all of the steps* of the following computation:

1. Square x ,
2. Multiply y by 6,

3. Add them together,
4. Subtract 7 from that, and
5. Divide the whole thing by 2.

Answer 2-2

If we follow all of the steps above, we should end up with a function that looks like this:

$$\frac{x^2 + 6y - 7}{2}$$

You can get there by adding things to the computation one at a time, e.g.,

1. x^2 : (square x)
2. $6y$: (multiply y by 6)
3. $x^2 + 6y$: (add them together)
4. $x^2 + 6y - 7$: (subtract 7 from that)
5. $(x^2 + 6y - 7)/2$: (divide the whole thing by 2)

Exercise 2-3: Defining functions

Write a function f that accepts a *non-negative integer* x (in other words, you do not need to worry about negative values in your answer) and gives instructions to

1. Produce the value 1 if $x < 2$
2. Otherwise add $f(x - 1)$ to $f(x - 2)$.

Answer 2-3

There are several acceptable answers. This can be written as two functions, as piecewise functions, or as code. Here are some possibilities that can at least give an idea of what will work:

$$\begin{aligned} f(0) &= 1 \\ f(1) &= 1 \\ f(x) &= f(x - 1) + f(x - 2) \end{aligned}$$

In this variation, we have specified how to compute $f(0)$ and $f(1)$, then specifies all other values in terms of the previous two. This is valid because we know we will not get negative inputs, so converting the idea of “less than 2” to “either 0 or 1” is fine. You might recognize this as the Fibonacci Sequence.

Another variation uses piecewise function notation:

$$f(x) = \begin{cases} 1 & \text{for } x < 2 \\ f(x-1) + f(x-2) & \text{otherwise.} \end{cases}$$

This defines the function in terms of its possible inputs. It is generally more widely accepted than the previous approach. Other variations include using set notation, e.g. $x \in \{0, 1\}$ instead of inequalities.

Finally, code is also acceptable, though we have not really gotten to that yet:

:javascript:

```
function f(x) {
  if (x < 2) {
    return 1;
  }
  return f(x-2) + f(x-1);
}
```

Here we define a JavaScript function called `f` that accepts formal parameter `x`. If it determines that `x` is less than 2, it returns the value 1 immediately, otherwise it computes the value as shown.

Exercise 2-4: Recursive evaluation, basic

Given the function $h(x) = x + h(x-1)$ where $h(0) = 0$, evaluate $h(5)$ showing all steps. Do not completely evaluate any subexpressions until all of their parts are available. Note that as a result of this, you will have multiple levels of brackets during parts of the computation.

Answer 2-4

Annotations are optional (horizontal braces), but the process should be clear. The student should show that they understand how recursive evaluation occurs (substitute, expand, evaluate). This is an example of how to expand $h(5)$ using annotations to clarify the answer:

$$\begin{aligned}
h(5) &= 5 + \underbrace{h(4)} \\
&= 5 + \underbrace{[4 + \underbrace{h(3)}]} \\
&= 5 + [4 + \underbrace{[3 + \underbrace{h(2)}]]] \\
&= 5 + [4 + [3 + \underbrace{[2 + \underbrace{h(1)}]]]] \\
&= 5 + [4 + [3 + [2 + \underbrace{[1 + \underbrace{h(0)}]]]]] \\
&= 5 + [4 + [3 + [2 + [1 + \hat{0}]]]] \\
&= 5 + [4 + [3 + [2 + \hat{1}]]] \\
&= 5 + [4 + [3 + \hat{3}]] \\
&= 5 + [4 + \hat{6}] \\
&= \underbrace{5 + \hat{10}} \\
&= \hat{15}
\end{aligned}$$

Note that order of operations is pretty important, here. It is true that you can collapse some of the addition steps on the left first if you take advantage of the associativity of addition, so partial credit can absolutely be given for that, but full credit should show the full set of steps, at least closely resembling the above.

Again, annotations are optional so long as the steps are clear. This may feel like an unnecessary amount of detail to the student, and for a problem like this it definitely is, but learning to think about things in this way will make a big difference when it comes time to understand what a program is doing, particularly when something goes wrong. This is all about building a mental model of the process using a simple example.

Exercise 2-5: Recursive evaluation, more advanced

Using pencil and paper, evaluate $f(6)$ using the definition below (this is the Fibonacci Sequence). Show all steps, and do not do any work early (do not take advantage of the associativity of addition, for example, just systematically complete the innermost expressions first).

$$f(x) = \begin{cases} 1 & \text{for } x < 2 \\ f(x-1) + f(x-2) & \text{otherwise.} \end{cases}$$

Answer 2-5

There might be some variation in how many operations are done at once and in which order. This is the order in which a computer might do it (without optimizations, etc.) if the recursive definition were written in straightforward code. Note that horizontal brace annotations are optional; the key is that the student can systematically produce an answer by doing recursive evaluation. So long as the steps are obvious, the specifics of how they are written are less important. An answer that follows the pattern outlined in this chapter is shown below:

$$\begin{aligned}
 f(5) &= \underbrace{f(3)} + f(4) \\
 &= \overbrace{[f(1) + f(2)]} + f(4) \\
 &= [\hat{1} + \underbrace{f(2)}] + f(4) \\
 &= [1 + \overbrace{[f(0) + f(1)]]} + f(4) \\
 &= [1 + [\hat{1} + \underbrace{f(1)}]] + f(4) \\
 &= [1 + [1 + \hat{1}]] + f(4) \\
 &= \underbrace{[1 + \hat{2}]} + f(4) \\
 &= \hat{3} + \underbrace{f(4)} \\
 &= 3 + \overbrace{[f(2) + f(3)]} \\
 &= 3 + \overbrace{[[f(0) + f(1)] + f(3)]} \\
 &= 3 + [[\hat{1} + \underbrace{f(1)}] + f(3)]
 \end{aligned}$$

...breathe, and continue...

$$\begin{aligned}
f(5) &= 3 + [\underbrace{[1 + \hat{1}]}_{\hat{2}} + f(3)] \\
&= 3 + [\hat{2} + \underbrace{f(3)}_{\hat{3}}] \\
&= 3 + [2 + \underbrace{[f(1) + f(2)]}_{\hat{3}}] \\
&= 3 + [2 + [\hat{1} + f(2)]] \\
&= 3 + [2 + [1 + \underbrace{[f(0) + f(1)]]}_{\hat{2}}]] \\
&= 3 + [2 + [1 + [\hat{1} + f(1)]]] \\
&= 3 + [2 + [1 + \underbrace{[1 + \hat{1}]]}_{\hat{2}}]] \\
&= 3 + [2 + \underbrace{[1 + \hat{2}]]}_{\hat{3}}] \\
&= 3 + \underbrace{[2 + \hat{3}]}_{\hat{5}} \\
&= \underbrace{3 + \hat{5}}_{\hat{8}} \\
&= \hat{8}
\end{aligned}$$

That is basically how the computer would do it. A person who knows to take advantage of the associativity of addition might do it a different way, collapsing some of the addition problems on the left before expanding functions on the right. Partial credit can be given for answers that involve that kind of early simplification, but it should be clarified to the student that this involves some mathematical manipulation that is usually beyond the capabilities of a computer to do, and the above expansion would be closer to reality in that environment.

Exercise 2-6: Match terms

Match the following terms

- substitution
- evaluation
- function
- call
- recursion
- break point
- watch
- step

with their definitions:

- A definition of a process of computation.
- The process of replacing a variable with its value.

- The process of performing a computation to produce a value.
- Supplying a function with an argument and getting a result.
- A computation that is defined in terms of itself.
- In a debugger, execute one statement of code and then stop.
- In a debugger, show the current value of a variable.
- In a debugger, where a program should pause for inspection.

Answer 2-6

substitution The process of replacing a variable with its value

evaluation The process of performing a computation to produce a value

function A definition of a process of computation

call Supplying a function with arguments and getting a result

recursion A computation that is defined in terms of itself

break point In a debugger, where a program should pause for inspection.

watch In a debugger, show the current value of a variable.

step In a debugger, execute one statement of code and then stop.

Exercise 2-7: Pencil and Paper Debugging

Given the program below, pretend that you have entered it into the computer, set a debug watch on x and y , and are stepping through it one line at a time, starting at the top. What are the values of x and y at each step? Show them in a table like this:

x	y
—	—
—	—
—	—
—	—
—	—
—	—

Hint: y will be undefined until it has been assigned something.

```
x = 10;
x = x + 1;
x = x * 3;
y = 10 + 3 * x;
x = x - 3 + y;
```

Answer 2-7

x	y
10	undefined
11	undefined
33	undefined
33	109
139	109

- First we set $x = 10$, so that is the first value of x in the table. As y has not been assigned yet, it is `undefined`.
- Next, we compute $x + 1$ and assign it back to x , so x is now 11, and y has not changed.
- Then we compute $x * 3$ and assign it back to x , so x is now 33, and again, y has not changed.
- Now we leave x alone and set y to be $10 + 3 * x$. Because multiplication has precedence over addition, this computes $3 * 33$ first, then adds 10, giving us 109, which is assigned to y .
- Finally, we compute $33 - 3 + 109$ to get 139, and assign that to x .

If you were to type this program in and load it into the debugger using the instructions in this chapter, you would be able to set watches on x and y , set a breakpoint on the first line, reload the page, and step through to see each row of this table in turn. Doing this in your head is an important skill to develop, as most bugs can be caught before a program is run this way. Many bugs that people experience in their software come from an incomplete mental model of what is going on, and a pencil can be a useful tool for exposing those gaps.

Chapter 3 Solutions

Exercise 3-1: Parameters and Arguments

What is the difference between a “parameter” and an “argument” when talking about functions?

Answer 3-1

parameter Also known as a “formal parameter”, this is a *variable* in the function’s *definition* that can be used during evaluation.

argument An argument is a *value* passed into a function *call*.

Note that *sometimes* these are used interchangeably in informal settings, but we are very careful to keep them separate in this course. It helps immensely when learning about the different contexts in which functions appear: definition and evaluation.

Exercise 3-2: Function Anatomy

In a function definition like this one,

```
function A(B) {  
  // C  
}
```

which parts of the function (A, B, and C) correspond to

- the name
- the parameter(s)
- the body

Answer 3-2

A is the function’s name, B is its one parameter, and C is in a comment located in the body of the function.

Exercise 3-3: Return

What does the `return` keyword do in a function?

Answer 3-3

The `return` keyword produces the final value of a function call. Bonus points for mentioning that it also causes the function to exit immediately, even if there is code below it. The main

thing for students to get comfortable with here, though, is that the value of a function call is determined by what is `returned`.

Exercise 3-4: Function Definition

Define (and write down below) a function called `mid` with two formal parameters `low` and `high` that returns the number right in the middle of the two. Hint: this number will be the *average* of `low` and `high`.

When finished, test your function in a real program by passing it the following values and alerting the results. Then fill in the answers:

low	high	mid(low, high)
4	6	
2	10	
-5	2	
17	37	

Answer 3-4

One possible implementation for the function is this:

:javascript:

```
function mid(low, high) {
    return (low + high) / 2;
}
```

This uses the averaging technique hinted at above. Some students will miss the meaning of that hint, and that is fine so long as they have something that works. For example, a student might want to subtract `low` from `high` first, find the middle, then add `low` back in, like this:

:javascript:

```
function mid(low, high) {
    return low + (high - low) / 2;
}
```

There are actually sometimes reasons to do things in this slightly more complicated way that have to do with numerical stability, but that is sort of beside the point. The point is that anything that correctly computes the number directly between `low` and `high` will do fine.

To compute the table, the student might have written a full program like this:

:html:

```

<script>
function mid(low, high) {
  return (low + high) / 2;
}

alert(mid(4, 6));
alert(mid(2, 10));
alert(mid(-5, 2));
alert(mid(17, 37));
</script>

```

It is not required that the student do it precisely this way, but it is a useful way to go about it, and that might be an opportunity to instruct or inspire with the idea that tests are just programs.

The resulting table should look reasonably close to this:

low	high	mid(low, high)
4	6	5
2	10	6
-5	2	-1.5
17	37	27

Exercise 3-5: Scopes

In the following program snippet, fill in the table with the scope of all variables. These will be either “global” or “local”. Recall that global variables can be seen both outside *and* inside of functions, and local variables are only visible inside and during a call.

Also, remember what `var` does. It’s kind of important.

```

x = 10;

function f(a, b, c) {
  z = 3 * c;
  var d = a + b - z;
  return d * d;
}

y = 15;

alert(f(x, y, 5));

```

variable	scope
x	
y	

variable	scope
z	
a	
b	
c	
d	

Answer 3-5

Variables declared using `var` in a function, or declared as the function's formal parameters, are **local** to that function. If the `var` keyword is left off, or the variable appears outside of any function, it is **global**. Thus, the table should be filled out like this:

variable	scope
x	global
y	global
z	global
a	local
b	local
c	local
d	local

Exercise 3-6: Anonymous Functions and Timers

Write code to set a timer to `alert("Ding!")` after 5.5 seconds using `setTimeout`. Use two different methods, one with a named function, and one with an anonymous function. Recall that `setTimeout` is called like this: `setTimeout(functionToCall, delayMilliseconds)`.

Answer 3-6

For this, the student can write a single program or two different programs. Example solution code is here:

:javascript:

```
// Call setTimeout with a named function.
function ding() {
    alert("Ding!");
}
setTimeout(ding, 5500);
```

```
// Call setTimeout with an anonymous function.
setTimeout(function() {
    alert("Ding!");
}, 5500);
```

The first calls `setTimeout` after defining a function called `ding`. That function alerts the required message as expected, and `setTimeout` is called with it and a value of 5500 milliseconds, which corresponds to 5.5 seconds.

The second call to `setTimeout` packages the alert function up in an anonymous function and also passes the value 5500 as the number of milliseconds to delay.

Exercise 3-7: A Tiny Game

Write a program that, each time you reload the page, prompts for a number and alerts its signed distance from the number 5. See if you can get someone else to play it and figure out what it is doing without telling them.

Hint: the signed distance of the number x from 5 is $x - 5$.

Answer 3-7

This program uses both `prompt` and `alert`. Each time the browser page is loaded, the program will run, so the steps the program takes are

- Get a number
- Subtract 5 from it
- Alert the value.

This can be done with or without functions. The point of this exercise is to get a little more experience with `prompt`. Here is some example code:

:html:

```
<script>
num = prompt("Enter a number");
dist = num - 5;
alert("The distance is " + dist);
</script>
```

Many variations on that theme are fine, including those with no explanatory text. The smallest such program might well be this one, since it is valid to not give any message to `prompt`:

:html:

```
<script>
alert(prompt() - 5);
</script>
```

In fact, given that it is so short, the student could easily just use the console to enter and run it.

Chapter 4 Solutions

Exercise 4-1: Object Creation

Show how to create **objects** with the following characteristics:

1. Empty
2. One property with key "name" and the value being your own name (or you can make the value just be the string "me").
3. One property with a key that is a number, like 5, and a value of anything you like.
4. The following properties, with string values that correspond to your favorite book:
 - "Title"
 - "Author"
 - "Year Published"

Answer 4-1

The three things requested look like this (variable assignments are optional, and are here to clarify which part of the problem is being answered):

:javascript:

```
empty = {};  
myName = {name: "me"};  
numeric = {5: "arbitrary value"};  
book = {  
  Title: "Where the Red Fern Grows",  
  Author: "Wilson Rawls",  
  "Year Published": 1961,  
};
```

A few things are of note here. First, to make an empty object you just use empty curlyes, like this {}. There can be space between them, since space is ignored except where needed to separate things (or inside of strings, of course).

Second, for the "name" answer, the student's own name can be in the value instead of "me", and it is also fine to put quote marks around the word "name". The quotes are optional because name is a valid JavaScript identifier (the biggest tip-off is usually that it does not start with a number, and contains no spaces, dots, or dashes).

Third, for the numeric key, you just use a number. The student might have interpreted the number to be a string, so you might also see something like {"5": "some value"}, and that is also fine.

Finally, for the book entry, any book title, author, and year are fine. The keys can be in quotes or not, except for "Year Published", which contains a space and therefore **must** be

quoted. The year can either be a number as given or a string with quote marks around it. Either is fine.

Exercise 4-2: Object Property Retrieval (and Setting)

Show how to get (or set) values in an object stored in the variable `obj` for properties with the following names:

1. `"title"`
2. `10`
3. `"something amazing"`

Do any of these allow you to use dot notation? Why can you not use dot notation on the others?

Answer 4-2

Here is a set of possible answers for each of these:

```
                                :javascript:

// title
obj["title"]
obj.title

// 10
obj[10]

// something amazing
obj["something amazing"]
```

The answer to the “why” question is this:

- The number `10` (and the string `"10"` if that is the way the student understood it) is not a valid identifier because it is (or starts with) a number.
- The text `"something amazing"` is not a valid identifier because it contains a space.

Thus, for both `10` and `"something amazing"`, the only way to access those properties is with bracket notation.

Exercise 4-3: Non-existent Object Keys

What happens if you try to get a property out of an object that does not exist? What value do you get instead?

Answer 4-3

You get the special value `undefined`, but otherwise things keep plodding along. No errors occur, and the code does not stop running just because you tried to get a value that was not there.

Exercise 4-4: Deleting Object Properties

How do you delete a property from an object? Hint: you cannot just assign `undefined` to it.

Answer 4-4

The answer here is to use the `delete` statement, like this:

:javascript:

```
delete obj.someProperty;  
// Or alternatively:  
delete obj["someProperty"];
```

That deletes the property called `"someProperty"` from the object.

Exercise 4-5: Logging

How do you log something to the console from code in a file (as opposed to just entering it directly in the console)? How do you log *multiple* things at once?

Answer 4-5

Use the `console` object and one of its logging methods. The most common of these is used like this:

:javascript:

```
console.log("whatever you want to log",  
           "and maybe something else");
```

If the student has any variation on `console.log` or `console.debug` or similar, then it is fine. The two critical pieces are the use of a `console` logging function, and the understanding that multiple things can be logged at the same time by specifying them all as arguments to the logging function. The example above has two things logged at once, so they will show up in the log together.

That can be very useful when trying to figure out what your program is doing, because you can add information about what you are looking at, like this:

:javascript:

```
console.log("myVar:", myVar);
```

This logs the literal string `"myVar: "`, followed by the *value* (or contents) of the variable `myVar`.

A rather odd (and discouraged) but still technically valid approach is to use bracket notation to get the log function:

:javascript:

```
console["log"]("myVar:", myVar);
```

Again, that is uncommon and therefore discouraged. It does work, though, even with methods (which is sort of surprising: `"hi there"["toUpperCase"]()` does, in fact, produce the value `"HI THERE"`, but goodness does it look odd).

Exercise 4-6: Standard Library

What is a language's "standard library"?

Answer 4-6

The standard library is a set of functions that are always available to be used by programmers.

That is a fine answer. A longer answer might include something along these lines.

The standard library functions are always there, ready to go. In JavaScript, the standard library is mostly contained in a bunch of global objects like `Math` and `Date`, etc., and when working within the browser, it includes objects like `document` and `window`, which contain a lot of built-in functionality.

Exercise 4-7: Math Practice

Without the aid of the computer, write down how you would compute a few things using the `Math` library:

1. 2^{12}
2. $\sqrt{3}$
3. $|-56|$
4. $\lfloor 6.8 \rfloor$
5. $\lceil 2.3 \rceil$

Note that $|x|$ indicates the "absolute value" (positive value of x), $\lfloor x \rfloor$ indicates the "floor" (nearest integer at or below x), and $\lceil x \rceil$ indicates the "ceiling" (nearest integer at or above x).

Answer 4-7

1. 2^{12} : `2**12` or `Math.pow(2, 12)` are both fine, but the latter is preferred because it follows instructions more closely ("using the `Math` library").
2. $\sqrt{3}$: `Math.sqrt(3)` or `Math.pow(3, 0.5)` both work fine.
3. $|-56|$: `Math.abs(-56)`
4. $\lfloor 6.8 \rfloor$: `Math.floor(6.8)`

5. `[2.3]:Math.ceil(2.3)`

Exercise 4-8: String Practice

Strings have a bunch of standard methods built in, as well as the standard `length` property. Solve this without the aid of a computer: given the string `s = " This string is mine. "`, what will the following methods produce? Hint: the spaces at the ends of it are part of the string because they are within the quote delimiters.

1. `s.length`
2. `s.toUpperCase()`
3. `s.toLowerCase()`
4. `s.trim()`
5. Bonus: `s.toLowerCase().trim()`

Answer 4-8

The results of these expressions can be seen in the console to check the student's answers:

:console:

```
> s = " This string is mine. "
< " This string is mine. "
> s.length
< 22
> s.toUpperCase()
< " THIS STRING IS MINE. "
> s.toLowerCase()
< " this string is mine. "
> s.trim()
< "This string is mine."
> s.toLowerCase().trim()
< "this string is mine."
```

The last one might feel a little bit like a trick, which is why it is a bonus problem (and a chance to learn something):

- We first get `s`, which is the string `" This string is mine. "`.
- We then call `toLowerCase()` on `s`, which evaluates to `" this string is mine. "` (all lower case).
- On that *new lower-case string* we then call `trim()`, which evaluates to `"this string is mine."` (leading and trailing spaces removed).

Basically, the interpreter reads left to right just like we do. It evaluates `s`, then calls `toLowerCase()` on it to produce a new value, then calls `trim()` on *that* to produce a final value. This sort of “method chaining” is pretty common in JavaScript and similarly-structured languages.

Exercise 4-9: Canvas Practice

Write a program that draws three squares (using `fillRect`) on a 300-by-300-pixel canvas. The squares will have the following properties:

- Each square is 100 pixels on a side,
- Each is a different color, one of red, yellow, and green.
- The first square is in the upper left corner, the second is in the middle, and the third is in the lower right corner.

Answer 4-9

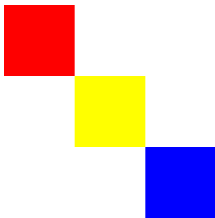
There is not a lot of room to wiggle in this assignment, but the colors can be in any order. This answer provides a fairly literal interpretation of the color requirements, putting them in the order given from left to right.

A correct answer will be a full working program. The student should be able to show the HTML source and the output in a browser window. The ID of the canvas can be anything; the important thing is that the student can create the appropriate canvas tag and find it in code.

:html:

```
<canvas id="drawing" width="300" height="300"></canvas>
<script>
  var canvas = document.getElementById('drawing');
  var ctx = canvas.getContext('2d');

  ctx.fillStyle = 'red';
  ctx.fillRect(0, 0, 100, 100);
  ctx.fillStyle = 'yellow';
  ctx.fillRect(100, 100, 100, 100);
  ctx.fillStyle = 'blue';
  ctx.fillRect(200, 200, 100, 100);
</script>
```



Chapter 5 Solutions

Exercise 5-1: Comparators and Boolean Values

Every expression in mathematics that involves a *comparator* has a boolean type. A comparator is something like $=$, $<$, $>$, \leq , \geq , etc.

1. What is the value of $7 > 10$?
2. What is the value of $3 = 3$?
3. What about the value of $3 = 3 \vee 7 > 10$?
4. What is the value of $3 = 3 \wedge 7 > 10$?
5. If $x = 2$, find two values of y that make this false, and two values of y that make it true: $x < 2y \wedge y < x + 4$.

Answer 5-1

These are relatively straightforward if the student remembers that numeric comparators have precedence over boolean operators.

The student can use any notation to answer these. Any of true, T , or \top are valid for true values, and any of false, F , or \perp are valid for false values. For the answers here we will use T and F :

1. $(7 > 10) = T$
2. $(3 = 3) = T$
3. $(3 = 3 \vee 7 > 10) = T$
4. $(3 = 3 \wedge 7 > 10) = F$
5. Since $x = 2$, we basically have $2 < 2y \wedge y < 6$, which can be manipulated to be $1 < y \wedge y < 6$. But since both of those must be true in an expression that evaluates true (because of the \wedge), you can also write it more familiarly as $1 < y < 6$. That should make it easy to find answers that make it true and answers that make it false. For example, $y = 4$ or $y = 5$ both make the expression true, while $y = 6$ and $y = 1$ both make it false.

Exercise 5-2: Boolean Connectives

- What is the symbol for OR?
- What is the symbol for AND?
- What is the symbol for NOT?
- What is their precedence order (highest to lowest)?

Answer 5-2

- OR: \vee

- AND: \wedge
- NOT: \neg
- Precedence order: \neg, \wedge, \vee

A little more explanation might be in order here. The operators OR (\vee), AND (\wedge), and NOT (\neg) are very similar in spirit to the mathematical operators ADD, MULTIPLY, and NEGATE, particularly if anything above 1 is clipped to 1. You can kind of see it if you treat T and F as 1 and 0 instead, and start applying the math operators to them. Check out the similarities in these tables. First for \vee :

A	B	$A + B$	Clipped to $[0, 1]$	$A \vee B$
0	0	0	0	0
0	1	1	1	1
1	0	1	1	1
1	1	2	1	1

This works similarly for \wedge :

A	B	AB	Clipped to $[0, 1]$	$A \wedge B$
0	0	0	0	0
0	1	0	0	0
1	0	0	0	0
1	1	1	1	1

For \neg things are a bit weird, since negating zero doesn't do anything. There is an analogue to it, but it is a bit beyond this course to discuss all of the interesting ways that $-$ and \neg are related. The multiplication and addition analogies are useful, though, because the corresponding boolean operators AND and OR have the same precedence relationship: multiplication before addition, and AND before OR.

Exercise 5-3: Truth Tables

Compute a truth table for the expression $\neg A \vee B$. It will have 4 rows. You may use 0/1 or T/F:

A	B	$\neg A \vee B$
0	0	
0	1	
1	0	
1	1	

Answer 5-3

$\neg A \vee B$ is true when either A is false or B is true (or both), so that leaves us with this:

A	B	$\neg A \vee B$
0	0	1
0	1	1
1	0	0
1	1	1

Exercise 5-4: More Truth Tables

Fill out the truth table for the expression $A \wedge B \vee \neg C$. Remember the order of operations!

A	B	C	$A \wedge B \vee \neg C$
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Answer 5-4

This question tests the student's understanding of operator precedence. The \wedge expression has to be computed before the \vee expression. Similarly, negation has the highest precedence in this expression, so it would be parenthesized as shown below:

A	B	C	$(A \wedge B) \vee (\neg C)$
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Since you have something ORed with $\neg C$, that means that any time C is false, the expression is true. Why is that?

Because if *either* side of an OR (\vee) expression is true, the whole expression is true. That means you can immediately fill out all of the rows where $C = F$ with true values, like this.

A	B	C	$(A \wedge B) \vee (\neg C)$
0	0	0	1
0	0	1	
0	1	0	1
0	1	1	
1	0	0	1
1	0	1	
1	1	0	1
1	1	1	

That's a big help, and now we don't have to think about $\neg C$ anymore - we have used all it has to offer. That leaves us with $A \wedge B$, which is only ever true if *both* A and B are true. That only happens in one place that hasn't already been marked, so we now have this:

A	B	C	$(A \wedge B) \vee (\neg C)$
0	0	0	1
0	0	1	
0	1	0	1
0	1	1	
1	0	0	1
1	0	1	
1	1	0	1
1	1	1	1

The rest of them have to be false, because C is true and one of A and B is false in all of those cases.

Exercise 5-5: Most Truth Tables

Create the truth table for $A \wedge B \vee C \wedge \neg D$. Hint: it will have 16 rows of values. You can use any representation of true and false you like, including ones and zeros.

Answer 5-5

The first thing to do when tackling this sort of problem is to figure out where the parentheses

should go (which operations to do first). NOT (\neg) comes before AND (\wedge), and AND comes before OR \vee . That leaves us with this expression:

$$(A \wedge B) \vee (C \wedge (\neg D))$$

The next step is to figure out how to fill out the 16 rows of values in a systematic way. That is done by counting in binary, and the resulting table looks like this:

A	B	C	D	$(A \wedge B) \vee (C \wedge (\neg D))$
0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	
0	1	0	0	
0	1	0	1	
0	1	1	0	
0	1	1	1	
1	0	0	0	
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	
1	1	0	1	
1	1	1	0	
1	1	1	1	

You can see the pattern here in the ones and zeros. The right column alternates every time, the next column to the left alternates every other time, the column to the left of that alternates twice as slowly still, etc.

With that set up, you can now fill in the answers. Let's look at that expression again, but let's just focus on the first part, since it's ORed with the second; any time the first part is true, the whole expression has to be true because of that \vee operator joining the two halves. We can look at it like this:

$$(A \wedge B) \vee \text{doesn't matter what else}$$

With that, we can look at every place where both A and B are true, and our expression will be true there. That covers the last four rows in our table:

A	B	C	D	$(A \wedge B) \vee (C \wedge (\neg D))$
0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	
0	1	0	0	
0	1	0	1	
0	1	1	0	
0	1	1	1	
1	0	0	0	
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Now we can ignore the *first* part of the expression and focus our attention on the *second* part in the same way:

$$\text{whatever} \vee (C \wedge \neg D)$$

In this case, C has to be true while D is false for this part of the expression to be true, since we first negate D , then AND it with C . Filling in all of the places where C is true and D is false gives us this:

A	B	C	D	$(A \wedge B) \vee (C \wedge (\neg D))$
0	0	0	0	
0	0	0	1	
0	0	1	0	1
0	0	1	1	
0	1	0	0	
0	1	0	1	
0	1	1	0	1
0	1	1	1	
1	0	0	0	
1	0	0	1	
1	0	1	0	1

A	B	C	D	$(A \wedge B) \vee (C \wedge (\neg D))$
1	0	1	1	
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Note that there is some overlap: the last 1 in the table was already set. This is fine.

The rest of the values will be zeros, since there are no other parts of the expression to make this true:

A	B	C	D	$(A \wedge B) \vee (C \wedge (\neg D))$
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Exercise 5-6: De Morgan

Use De Morgan's Law to transform the following expressions (move the outer negation into the expression). Hint: when there is no outer negation, you can first pretend that there are *two* outer negations instead, like this: $A \wedge B = \neg\neg(A \wedge B)$ (since one undoes the other), and you can move one of them into the expression.

1. $\neg(A \wedge B)$
2. $\neg(A \vee B)$
3. $\neg(A \wedge B \vee \neg C)$
4. $\neg(\neg A \vee B \wedge \neg C)$

Answer 5-6

De Morgan's Law is a pair of relations:

$$\neg(x \wedge y) = \neg x \vee \neg y$$

$$\neg(x \vee y) = \neg x \wedge \neg y$$

If you are careful with what you take x and y to be, you can convert lots of expressions into this form. A common way to remember it is to distribute the \neg inside the parentheses, and flip whatever operators you find there.

With that, we can tackle these four problems. Note that if you are not careful with parentheses, they will reach out and bite you, so we do everything one step at a time. The first two are straightforward:

1:

$$\neg(A \wedge B) = \neg A \vee \neg B$$

2:

$$\neg(A \vee B) = \neg A \wedge \neg B$$

The next two are a little more involved, requiring multiple steps:

3:

$$\begin{aligned}\neg(A \wedge B \vee \neg C) &= \neg((A \wedge B) \vee \neg C) \\ &= \neg(A \wedge B) \wedge C \\ &= (\neg A \vee \neg B) \wedge C\end{aligned}$$

4:

$$\begin{aligned}\neg(\neg A \vee B \wedge \neg C) &= \neg(\neg A \vee (B \wedge \neg C)) \\ &= A \wedge \neg(B \wedge \neg C) \\ &= A \wedge (\neg B \vee C)\end{aligned}$$

Exercise 5-7: JavaScript and Equality

Give the true or false value for each of the following expressions:

1. `3 === "3"`
2. `1.6 !== 1.6`
3. `"hi" === 'hi'`
4. `"hi" !== "hello"`
5. `3 > 6`
6. `2 <= 2`
7. `7 >= 6`
8. `!(9 >= 9)`
9. `10 < 8 || 8 < 9`
10. `9 === 9 && 2 !== 3`

Answer 5-7

As a reminder, these operations have the following meanings:

Operator	Meaning
<code>===</code>	Equal
<code>!==</code>	Not equal
<code><</code>	Less than
<code><=</code>	Less or equal
<code>></code>	Greater than
<code>>=</code>	Greater or equal
<code>&&</code>	AND
<code> </code>	OR
<code>!</code>	NOT

An additional reminder can help with this answer:

- Strings are tested for equality based on *contents*, not *delimiters*, so `"abc" === 'abc'` is true (delimiters are not part of the string; they just tell us where it is in the code).

The answers:

1. `3 === "3": true`
2. `1.6 !== 1.6: false`
3. `"hi" === 'hi': true`
4. `"hi" !== "hello": true`
5. `3 > 6: false`

6. `2 <= 2`: true
7. `7 >= 6`: true
8. `!(9 >= 9)`: false
9. `10 < 8 || 8 < 9`: true
10. `9 === 9 && 2 !== 3`: true

Exercise 5-8: Converting Strings to Numbers

When prompting for an answer using `prompt`, the value returned is a string representing the characters the user typed. For example, if you run this:

```
var val = prompt("Please enter a number:");
```

The user is shown a window with a text box into which they can type. There is nothing in that box that *forces* them to type a number, other than your message, which is really just advice.

Suppose, however, that the user does indeed type a string representing a number, like `11`. What should you do to `val` to ensure that you can do numeric things (like addition, subtraction, multiplication, etc.) to it?

Answer 5-8

This is just a memorization question to ensure that a common and useful idiom is practiced. Taking a string that contains a number, something like `"11"`, and applying the *unary plus* operator turns it into a number, like this:

```
:javascript:
```

```
var val = prompt("Please enter a number:");
var num = +val;

// You can also do it all at once, like this:
var num = +prompt("Please enter a number:");
```

In general, JavaScript will try really hard to do the conversion for you, so you can actually do lots of number things to strings to force the conversion, like `5 < "49"` - that will convert `"49"` to a number first and then do the comparison.

That also happens with `==`, but that sort of makes it dangerous because equality testing is used for much more than numbers. Therefore, we typically avoid using `==` and its sibling `!=` because they are just a bit too generous with their `true` values to be safe. Many errors can be caught early by using the `===` and `!==` operators instead, but it's at least good to know what the shorter versions (that are also very commonly, and one might even argue inappropriately, used) do.

Exercise 5-9: Conditional Expressions

It's possible to create an expression in JavaScript that is one thing for one situation, and something else for another. This is done using the ternary conditional operator: `? : .` What are the values of the following expressions?

Note that we threw a little surprise in here, but it's not too hard to make use of it: the number `0` is falsy, and *any other number* is truthy. Similarly, the empty string `' '` is falsy, but every other string is truthy. (Note that the concepts of “truthy” and “falsy” come up in the discussion of `if/else`).

1. `true ? "hello" : "good-bye"`
2. `5 < 6 ? "smaller" : "not smaller"`
3. `'hi' === 'hello' ? "yes" : "no"`
4. `5 ? "truthy" : "falsy"`
5. `0 ? "truthy" : "falsy"`
6. `' ' ? 1 : 0`

Answer 5-9

1. `true ? "hello" : "good-bye": "hello"`
2. `5 < 6 ? "smaller" : "not smaller": "smaller"`
3. `'hi' === 'hello' ? "yes" : "no": "no"`
4. `5 ? "truthy" : "falsy": "truthy"`
5. `0 ? "truthy" : "falsy": "falsy"`
6. `' ' ? 1 : 0: 0`

Exercise 5-10: If and Else

Write a program that does the following using `if` and `else` (don't forget your braces!):

- Ask the user for a number,
- If less than zero, output “negative”,
- Otherwise, output “non-negative”.

Answer 5-10

There are several ways to do this, but this assignment asks specifically for `if` and `else`, so the code will be required to look something like this:

:javascript:

```
var num = +prompt("Enter a number");
if (num < 0) {
    alert("negative");
} else {
    alert("non-negative");
}
```

Also, braces are explicitly required here, too, so don't accept code that is missing curly braces.

Exercise 5-11: More If and Else

For this program, use two techniques: the first will use nested `if/else` blocks, and the second will use an `else if` chain.

Write a program that outputs “negative” for a number less than zero, “zero” for a number equal to zero, and “positive” for a number greater than zero. Get the number using `prompt` and output either to the console or via `alert`.

Answer 5-11

As a reminder, always look out for curly braces in conditional statements.

The first program is to use nested `if/else` blocks. There are several ways of going about it. Here is one of the most straightforward:

:javascript:

```
var num = +prompt("Enter a number");
if (num < 0) {
  console.log("negative");
} else {
  if (num === 0) {
    console.log("zero");
  } else {
    console.log("positive");
  }
}
```

Of course, the order of tests can be changed and the program will still be perfectly valid (they are all mutually exclusive). For example, this works just as well:

:javascript:

```
var num = +prompt("Enter a number");
if (num === 0) {
  console.log("zero");
} else {
  if (num < 0) {
    console.log("negative");
  } else {
    console.log("positive");
  }
}
```

Finally, the second program should just collapse these into an `else if` chain, like this (using the most recent example above as the starting point):

:javascript:

```

var num = +prompt("Enter a number");
if (num === 0) {
  console.log("zero");
} else if (num < 0) {
  console.log("negative");
} else {
  console.log("positive");
}

```

Exercise 5-12: Output Tables

Make an output table for the “negative/zero/positive” program in the previous problem.

Answer 5-12

The output table is constructed following the pattern in the text. Because this is transformable into an `else if` chain, it will likely have that kind of structure. Drawing the table with all `false` at the top as is customary (and using `true` and `false` instead of 1 and 0 - anything similar is acceptable), we get this:

num === 0	num < 0	Output
false	false	“positive”
false	true	“negative”
true	?	“zero”

The last row has the “don’t care” symbol. The actual table for your student may vary, but at least one row will have a “don’t care” in it, since a true value for the first condition causes the others to not matter. Note that the table can be rearranged (turn it upside-down) to have `true` in the diagonal (not including the row of all `false` values), with `false` in the lower left and `don't care` in the upper right. That is helpful for conditions that can be expressed as `else if` chains, and is a good sanity check for the student’s answer.

Exercise 5-13: More Output Tables

Consider the guessing game problem from this chapter, where different outcomes are displayed for guesses that are too small, too large, *much* too large, and correct. Now we are going to change it slightly. For the following altered code, build the output table and see if you can use it to spot the problem in our new implementation:

```

var answer = 6;
var guess = +prompt("Guess a number");
if (guess === answer) {
  alert("You win!");
} else {

```

```

if (guess > answer) {
  alert("Too high");
} else {
  if (guess > answer + 10) {
    alert("Way too high");
  } else {
    alert("Too low");
  }
}
}

```

Answer 5-13

The test for `guess > answer` was swapped with the test for `guess > answer + 10` in this exercise. To build the output table, we do as before, creating a column for each of `guess === answer` and the other two already mentioned. The table will look like this:

=== answer	> answer	> answer + 10	Output
0	0	0	"Too low"
0	0	1	<i>impossible</i>
0	1	?	"Too high"
0	1	?	"Too high"
1	?	?	"You win"
1	0	1	<i>impossible</i>
1	1	0	<i>impossible</i>
1	1	1	<i>impossible</i>

Collapsing the like rows in the table and removing the impossible, we get this:

=== answer	> answer	> answer + 10	Output
0	0	0	"Too low"
0	1	?	"Too high"
1	?	?	"You win"

Aha. We are missing an outcome! There is no row for "Way too high", which indicates a flaw in our logic. We can see that as we study the "Too high" row: if `guess > answer`, we shouldn't have a "don't care" in the `guess > answer + 10` spot, because we *do* care about that, and therefore we need to test for it. Looking at things in this way can help to avoid bugs in conditional logic, and they do tend to creep in quite often, even for seasoned programmers. This is particularly true when chains of conditions get long and have overlapping truth values.

Exercise 5-14: Basic types and outputs

In the following code, fill in the missing values (look for '?'). Also say what *type* the result has. What kind of a thing is it?

:console:

```
> 10 + 5
< ?
> 3 * 12
< ?
> "hello" + " " + "world"
< ?
```

Answer 5-14

In the answer, the values are shown where the console would put them, and the types of those values are indicated in comments. Any sort of indication is fine.

:console:

```
> 10 + 5
< 15 // type = number
> 3 * 12
< 36 // type = number
> "hello" + " " + "world"
< "hello world" // type = string
```

Exercise 5-15: Variables and evaluation

Fill in the output of the console below where they are missing ('?'). Remember to be careful about assignment - a variable only changes value when assigned.

:console:

```
> x = 10
< ?
> x + 15
< ?
> x / 5
< ?
> y = 'hello'
< ?
> y + ' there'
< ?
```

Answer 5-15

:console:

```

> x = 10
< 10
> x + 15
< 25
> x / 5
< 2
> y = 'hello'
< "hello"
> y + ' there'
< "hello there"

```

Note that it's easy to forget that $x + 15$ doesn't *change* x , it just *outputs a value based on it*. That's why the output of $x / 5$ is just 2 and not 5: x never changed to be 25. It was 10 the whole time.

Exercise 5-16: Show the steps for evaluating a numeric expression

Using the expansion technique demonstrated earlier, show the steps of evaluation for the following expression:

$$3(2 + 4) - (3/(12 + 2))$$

Answer 5-16

Evaluating things inside-out, left to right means that we basically move to the right when we get stuck, and move inside whenever we can. Then we start over again on the left with the new expression, moving to the right when stuck, and inside whenever we can, repeating until we have a single value:

$$\begin{aligned}
 & 3(\underbrace{2 + 4}) - (9/(\underbrace{1 + 2})) \\
 & \underbrace{3(\widetilde{6})} - (9/(\underbrace{1 + 2})) \\
 & \widetilde{18} - (9/(\underbrace{1 + 2})) \\
 & 18 - (\underbrace{9/\widetilde{3}}) \\
 & \underbrace{18 - \widetilde{3}} \\
 & \widetilde{15}
 \end{aligned}$$

Exercise 5-17: Show the steps for evaluating a different numeric expression

Do the same as before, recalling that the \cdot symbol stands for multiplication and the $/$ symbol stands for division:

$$3 + 5 \cdot 2 + 6/3 - 4$$

Answer 5-17

This one requires careful attention to be paid to operator precedence rules. Multiplication and division always need to be done before addition and subtraction when there is a choice to be made. Moving from left to right, you immediately notice that you cannot perform the first addition because the next number is part of a multiplication. So you have to do that *first*, and *then* do the addition. This theme repeats throughout the evaluation process.

$$\begin{aligned} & 3 + \underbrace{5 \cdot 2} + 6/3 - 4 \\ & \underbrace{3 + 10} + 6/3 - 4 \\ & 13 + \underbrace{6/3} - 4 \\ & \underbrace{13 + 2} - 4 \\ & \underbrace{15} - 4 \\ & \underbrace{11} \end{aligned}$$

Exercise 5-18: Show the steps for evaluating a function with variables

The function $f(x, y)$ is given below. Using the expansion technique demonstrated earlier, show all of the steps of how you would evaluate $f(25, 3)$:

$$f(x, y) = \frac{x}{5} + 3 + y^3$$

Answer 5-18

$$\begin{aligned}
 f(x, y) &= \frac{x}{5} + 3 + y^3 \\
 f(25, 3) &= \frac{x}{5} + 3 + y^3 \\
 &= \frac{\widehat{25}}{\widehat{5}} + 3 + y^3 \\
 &= \widehat{\widehat{5}} + 3 + y^3 \\
 &= \widehat{8} + y^3 \\
 &= 8 + \widehat{\widehat{3^3}} \\
 &= \widehat{8 + \widehat{27}} \\
 &= \widehat{35}
 \end{aligned}$$

Exercise 5-19: Recursion in algebraic evaluation

Evaluate $f(4)$:

$$f(x) = \begin{cases} 0 & \text{for } x \leq 0 \\ x - f(x - 1) & \text{otherwise.} \end{cases}$$

Answer 5-19

Just like with regular evaluation, when you get stuck you go deeper, or move to the right. This leads to an expansion like the following:

$$\begin{aligned}
 f(x) &= \begin{cases} 0 & \text{for } x \leq 0 \\ x - f(x-1) & \text{otherwise.} \end{cases} \\
 f(4) &= \overbrace{4 - f(3)} \\
 &= 4 - \overbrace{(3 - f(2))} \\
 &= 4 - (3 - \overbrace{(2 - f(1))}) \\
 &= 4 - (3 - (2 - \overbrace{(1 - f(0)))) \\
 &= 4 - (3 - (2 - \underbrace{(1 - \tilde{0})})) \\
 &= 4 - (3 - \underbrace{(2 - \hat{1})}) \\
 &= 4 - \underbrace{(3 - \hat{1})} \\
 &= \underbrace{4 - \hat{2}} \\
 &= \hat{2}
 \end{aligned}$$

Exercise 5-20: Calling JavaScript Functions

Write a one-line program that shows a pop-up window with the text “Hello!” in it. Use the `alert` function.

There is no need to show the surrounding HTML—just show the relevant JavaScript code.

Answer 5-20

Calling functions in JavaScript involves naming the function and then adding parentheses that contain the function’s arguments. To show a pop-up window as described in the problem, you would write a program like this:

:javascript:

```
alert("Hello!");
```

If HTML is given (it is not required), it might look something like this:

:html:

```
<script>
alert("Hello!");
</script>
```

Exercise 5-21: Prompting For Values

Using the `prompt` function, ask the user for their name, and display “Hello <name>!”, where <name> is replaced by whatever the user types. For example, if I were to type “Chris” in your prompt box, you would display “Hello, Chris!”.

The correct answer need not contain surrounding HTML tags, though it can. The JavaScript that displays the prompt and the final message is the important part of the answer.

Hint: you will call two functions in a correct program, and only one of them will be `prompt`.

Answer 5-21

The task is to

- Get input from the user, and
- Display something to the user that contains that input.

There are a couple of ways to go about it. One of the more straightforward ways to think about it is to store the result of `prompt` in a variable, then compose a message from that input, then display the message, like this:

:javascript:

```
var name = prompt("Please type your name:");
var message = "Hello, " + name + "!";
alert(message);
```

Note that we are using string concatenation (with the `+` operator) to make the message from known text and unknown text.

Of course, this can also be done without any variables at all, if you just use the result of functions calls in place of the variables above:

:javascript:

```
alert("Hello, " + prompt("Please type your name:") + "!");
```

That’s not terribly readable, but it is acceptable as an answer. Sometimes it is useful to use variables even when not strictly necessary because they clarify intent for humans.

Exercise 5-22: Writing Functions

You will write two function definitions for this exercise:

- A niladic function (a function that accepts no parameters) called `getName` that prompts the user for a name and returns the value, and
- A function called `showGreeting` that accepts a name and displays “Hello <name>!”, as in the previous exercise.

A correct set of function definitions will allow you to write the previous answer as

```
var name = getName();  
showGreeting(name);
```

or even

```
showGreeting(getName());
```

Answer 5-22

There are two function definitions needed for a complete answer. First, a function that prompts the user for a name and returns the value. Recall that functions can be defined using the `function` keyword:

:javascript:

```
function getName() {  
    return prompt("Please type your name:");  
}
```

The second function shows a greeting based on the name passed into it. Note that the variable `name` can be called anything. All that matters is that the same parameter is used throughout:

:javascript:

```
function showGreeting(name) {  
    alert("Hello, " + name + "!");  
}
```

With these two functions, the program can be written as described in the problem. It's often useful to define functions like these even when they're small because they remove clutter. Removing clutter using well-named functions can clarify intent, since it's very obvious what this does. It shows a greeting using the obtained name:

:javascript:

```
showGreeting(getName());
```

Exercise 5-23: Events

Write a short program that causes "Hello!" to be displayed in a pop-up window after 5 seconds have passed. Use `setTimeout` to accomplish this.

Answer 5-23

To make something happen after a certain amount of time has passed, you can use `setTimeout` to call a function that you specify after the number of desired milliseconds. So, we first define a function that does what we want, i.e., displays "Hello!":

:javascript:

```
function hello() {  
    alert("Hello!");  
}
```

The function can be called anything, and since it needs no input, it is niladic (no formal parameters).

Once that is defined, we can call `setTimeout` with it, with a delay of 5000 milliseconds:

:javascript:

```
setTimeout(hello, 5000);
```

The full program is this:

:javascript:

```
function hello() {  
    alert("Hello!");  
}  
setTimeout(hello, 5000);
```

It is also possible (and common!) to use an anonymous function, like this:

:javascript:

```
setTimeout(function() {  
    alert("Hello!");  
}, 5000);
```

Here the function is defined and passed into `setTimeout` all at once.

Chapter 6 Solutions

Exercise 6-1: Grid Recursion

Use recursion to implement the `hLines` function. It works like `vertLines`, but it draws horizontal lines instead by marching along the `y` coordinate instead of the `x` coordinate. Run our program with both horizontal and vertical lines and verify that it works properly.

Answer 6-1

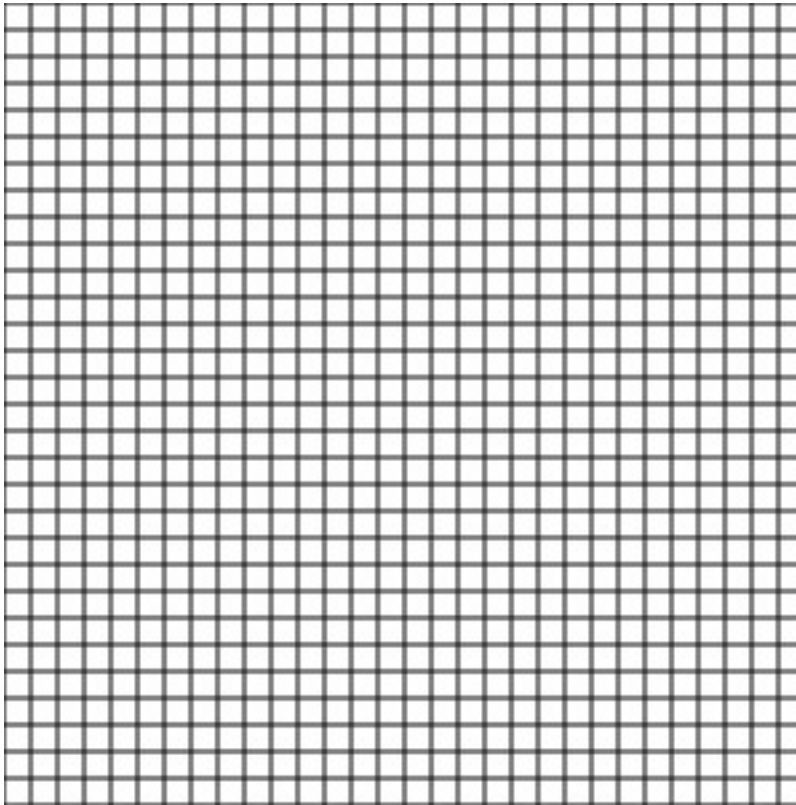
The complete program will look something like this:

:html:

```
<canvas id="grid" width="300" height="300"></canvas>
<script>
function vertLines(ctx, upTo) {
    if (upTo < 0) {
        return;
    }
    ctx.moveTo(upTo, 0);
    ctx.lineTo(upTo, ctx.canvas.height);
    vertLines(ctx, upTo - 10);
}

// NEW:
function hLines(ctx, upTo) {
    if (upTo < 0) {
        return;
    }
    ctx.moveTo(0, upTo);
    ctx.lineTo(ctx.canvas.width, upTo);
    hLines(ctx, upTo - 10);
}

var canvas = document.getElementById('grid');
var context = canvas.getContext('2d');
// NEW:
vertLines(context, canvas.width);
context.stroke();
</script>
```



Exercise 6-2: Partial Rays

Change the arguments passed initially to `rays` (the first time it is called by the interpreter) to end on something other than 360 degrees, and make the lines closer together (the original used 10 degrees, pick something smaller, like 5). Use this to make a sort of “sun on the horizon”, where only the top half of the circle is drawn.

Answer 6-2

To do this, we note that zero degrees is directly to the right, so we just change the final angle to 180 instead of 360: we will just go halfway around.

The previous call to `rays` looked like this:

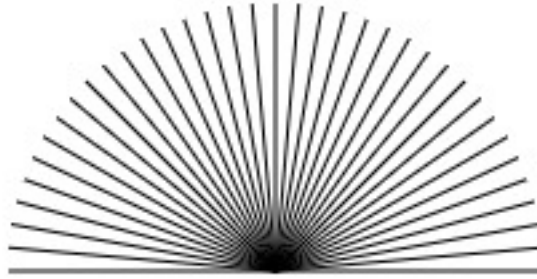
:javascript:

```
rays(ctx, 10, 360);
```

The right answer just changes it to this (the first number can be anything less than 10):

:javascript:

```
rays(ctx, 5, 180);
```



Exercise 6-3: Circle-like Things

The `rays` function basically draws a bunch of straight lines emanating out from a central point, and it does it in order, starting at zero degrees (pointing to the right), then going around counterclockwise until it reaches the final angle. It does this by repeatedly calling `moveTo` to get to the central point, then calling `lineTo` to get away from that point.

Note that, without a `moveTo` function, the first `lineTo` behaves as though it were a `moveTo`. This is in the documentation for the context object, but in case you don't have access to that, you're welcome.

In this exercise, copy the `rays` function somewhere and rename it to be `polygon`. Then see what happens if you don't call `moveTo` at all. Before running your program, answer this question:

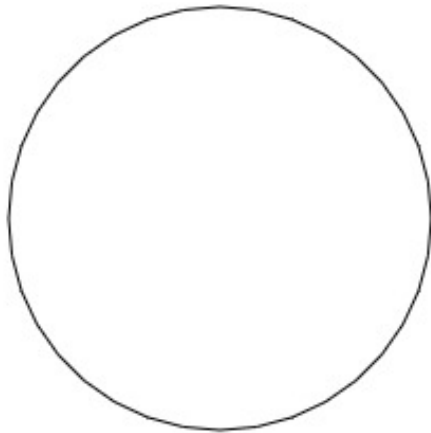
- What will happen if you don't call `moveTo` between line drawings? Hint: what was it doing before?

Once done, increase the `byDegrees` value in the first call to `polygon` to be 120. Before running the program again, answer these question:

- What shape will that draw?
- Why?

Answer 6-3

Without moving to the central point over and over again, the lines will all be connected end-to-end, because the `lineTo` function just draws a line from wherever it left off. That means it will go around in a circle that surrounds the central point, drawing lines connected to one another. In short, it draws a circle.



The correct code for the rest of this exercise will look a great deal like the original `rays` function, but everywhere you see the word `rays` will need to be replaced with the new function name, including inside of it and the call to it. This will be most obvious if the student creates a fresh program for this as suggested. If not, they might end up calling the old `rays` function and not seeing any difference, and wondering why.

When troubleshooting this with the student, the names of things should be the first thing checked.

The code is annotated below with “NEW” to indicate where the `rays` function is changed to make it into the `polygon` function. As you can see, the name of the function changed in both its definition and in the two places where it is called, and the call to `moveTo` was deleted. In addition, the initial call is now `polygon(ctx, 120, 360)`, which the student should indicate will draw a triangle (equilateral, for bonus points) because it hits three different coordinates at 0 degrees to 120 degrees, 120 degrees to 240 degrees, and finally 240 degrees to 360 degrees, which is back at 0.

:html:

```
<canvas id="polygon" width="300" height="300"></canvas>
<script>
```



```
// NEW: function name changed
function polygon(ctx, byDegrees, toDegrees) {
  if (toDegrees < 0) {
    return;
  }

  // Find the center, convert degrees to radians, etc.
  var angle = toDegrees * Math.PI / 180,
      size = ctx.canvas.width / 3,
      ox = ctx.canvas.width / 2,
      oy = ctx.canvas.height / 2;

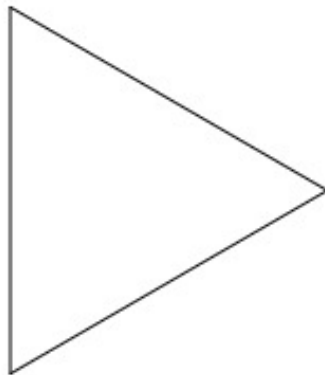
  // NEW: removed moveTo

  ctx.lineTo(ox + size * Math.cos(angle),
             oy - size * Math.sin(angle));

  // NEW: function name changed
  polygon(ctx, byDegrees, toDegrees - byDegrees);
}

var canvas = document.getElementById('polygon');
var context = canvas.getContext('2d');

// NEW: function name changed
polygon(context, 120, 360);
context.stroke();
</script>
```



Exercise 6-4: Time to Fly

Write a program that draws n rectangles of decreasing height, starting at the left edge and finishing at the right edge. The first rectangle's height will be the height of the canvas, and the last rectangle's height will be `canvas.height / n`. Use a recursive function name `bars` to draw these.

Once complete, demonstrate that it works for any n by plugging in different values and showing how the shape changes.

Answer 6-4

There are several ways to go about this. The student may start at the left and go right, or may start at the right and go left. This example shows the left-to-right approach.

Note that there is nothing wrong with displaying the rectangles upside-down. It is easier to do that, in fact, so I have shown that answer here. If the student displays the right-side up (all of them flush against the bottom of the canvas), that is worth bonus points, because it takes more work to figure out that the rectangle needs to be drawn with a different formula:

:javascript:

```
ctx.fillRect(left, ctx.canvas.height - height,
             width, height)
```

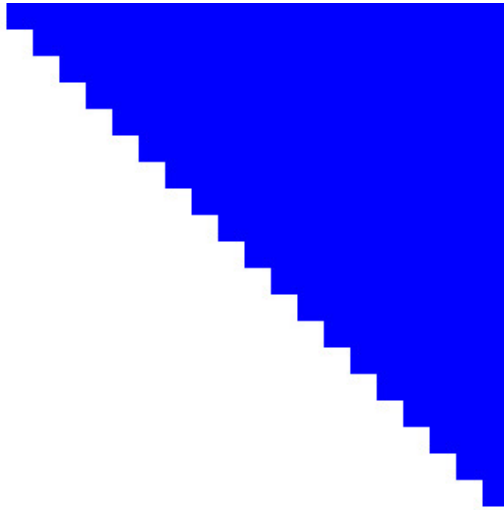
:html:

```
<canvas id="bars" width="300" height="300"></canvas>
<script>
function bars(ctx, curr, n) {
  if (curr === n) {
    return;
  }
  var width = ctx.canvas.width / n,
      left = curr * width,
      height = curr * ctx.canvas.height / n;

  ctx.fillRect(left, 0, width, height);
  bars(ctx, curr+1, n);
}

var canvas = document.getElementById('bars');
var ctx = canvas.getContext('2d');

ctx.fillStyle = 'blue'; // optional, just for fun
bars(ctx, 0, 20);
</script>
```



This example draws 20 rectangles, but it could easily be any number. Also, the example above demonstrates a method that assigns each rectangle a number from 0 to $n-1$, then calculates the position and height from that number. The student might instead approach it by keeping track of the position and height separately without any counting, and that is also valid. To do it that way, the code might end up looking something like this:

:html:

```
<canvas id="bars" width="300" height="300"></canvas>
<script>
function bars(ctx, width, heightChange, x, height) {
  if (x >= ctx.canvas.width) {
    return;
  }
  ctx.fillRect(x, 0, width, height);
  bars(ctx, width,
        heightChange, x + width,
        height - heightChange);
}

var canvas = document.getElementById('bars');
var ctx = canvas.getContext('2d');

ctx.fillStyle = 'blue'; // optional, just for fun
// Calculate width and height change:
var n = 15;
bars(ctx, canvas.width / n, canvas.height / n, 0, canvas.height);
</script>
```

The things of note here are that the function only knows the width and how much to change the height each time, as well as the current x coordinate and the current rectangle height. Then it

- Checks that it hasn't gone off the end,
- Draws the rectangle, and
- Calls itself with the settings for the new rectangle (only the last two parameters change).

Chapter 7 Solutions

Exercise 7-1: Array Basics

Write a small program that performs the following steps in order. Before running the program, answer the questions in the steps below and put your answers in the comments:

1. Create an empty array,
2. Push the numbers 1, 2, 3, and 5 onto the array,
3. Add a comment indicating the length you think the array has at this point,
4. Alert the value of `pop`,
5. Add another comment indicating the length you think the array has at this point,
6. Push the values 4 and 5,
7. Add final comment indicating the length you think the array has at this point, and
8. Alert the array.

Answer 7-1

There is very little wiggle room in this assignment, since it is fairly explicit. The student can choose any name for the variable used to hold the array, but that is about it. Here is a working program (just the JavaScript parts - there will be some surrounding HTML as is always the case so far):

:javascript:

```
var ar = [];  
  
ar.push(1);  
ar.push(2);  
ar.push(3);  
ar.push(5);  
// length: 4  
  
alert(ar.pop());  
// length: 3  
  
ar.push(4);  
ar.push(5);  
// length: 5  
  
alert(ar);
```

One allowable variant might have the value of `pop` assigned to a variable first, like this snippet shows:

:javascript:

```
var val = ar.pop();  
alert(val);
```

Of course, the format of the comments can be pretty much anything so long as the lengths are listed and are correct.

Exercise 7-2: Slicing Arrays

A slice of an array is a copy of a contiguous part of it. For the array `ar` below, write down the value of the following expressions:

```
var ar = [0, 1, 1, 2, 3, 5, 8, 13, 21];
```

1. `ar.slice(0)`
2. `ar.slice(1)`
3. `ar.slice(1, 3)`
4. `ar.slice(4, 5)`
5. `ar.slice(3, ar.length-1)`

Answer 7-2

1. `ar.slice(0) === [0, 1, 1, 2, 3, 5, 8, 13, 21]`: the entire array is sliced, because the starting element is 0 (the first element) and the ending element is omitted, implying “go to the end”.
2. `ar.slice(1) === [1, 1, 2, 3, 5, 8, 13, 21]`: this is the familiar “rest of the array” call, where everything except element 0 is returned.
3. `ar.slice(1, 3) === [1, 1]`: here the slice consists of elements starting at 1 inclusive, and ending at 3 exclusive. Therefore it contains values at indices 1 and 2 in the original array, which gives us the answer here.
4. `ar.slice(4, 5) === [3]`: just one element, since it has to include index 4 but exclude index 5. Note that you can always tell how many items will be in the new array by subtracting the first argument from the second. That can be handy.
5. `ar.slice(3, ar.length-1) === [2, 3, 5, 8, 13]`: here we start at 3 and go to one before the end, as shown.

Exercise 7-3: Finding elements in arrays

After running the “find an element in the array” function below, what is `i` when the value is found? What would it be if we were looking for `'e'` instead of `'c'`?

```
var values = ['a', 'b', 'c', 'd'];  
var i = 0;  
while (i < values.length) {  
  if (values[i] === 'c') {  
    break;  
  }  
}
```

```

    }
    i++;
}

```

Answer 7-3

After running the code as is, the value of `i` is going to be what it was when `break` was called, which will be the index of the value `c`. Since array indices start with 0, that means `i === 2`.

If you change the `if` statement to search for the value `'e'`, the loop will never call `break` at all. That means it will eventually increment `i` to be equal to `values.length`, which will cause the loop to exit without finding anything. Thus, when the value is not found, `i === values.length`, which is 4 in this case.

Exercise 7-4: Addition and Subtraction Assignment Operators

There are various ways to add and subtract values from things. After each line of the following program, show the value for `i`.

```

var i = 1;
i = i + 13;
i = i - 1;
i += 5;
i -= 2;
i++;
i--;

```

Answer 7-4

Line	Value of <code>i</code>
<code>var i = 1;</code>	1
<code>i = i + 13</code>	14
<code>i = i - 1</code>	13
<code>i += 5</code>	18
<code>i -= 2</code>	16
<code>i++</code>	17
<code>i--</code>	16

Exercise 7-5: Horizontal Lines Using While Loops

Write a program that draws a grid using `while` loops. To do so, take the `vertLines` function that uses a `while` loop and use that as a template to create a `hLines` function that looks similar. Put these together to make a grid on a canvas.

Answer 7-5

The program will have three major components: a `vertLines` function that is basically copied from the text, a `hLines` function that looks a lot like it, and the code to call both of them to compute and stroke the full grid. Those three elements are shown in the following code:

```

:html:
<canvas id="drawing" width="300" height="300"></canvas>
<script>
function vertLines(ctx) {
    var x = ctx.canvas.width;
    while (x >= 0) {
        ctx.moveTo(x, 0);
        ctx.lineTo(x, ctx.canvas.height);
        x = x - 10;
    }
}

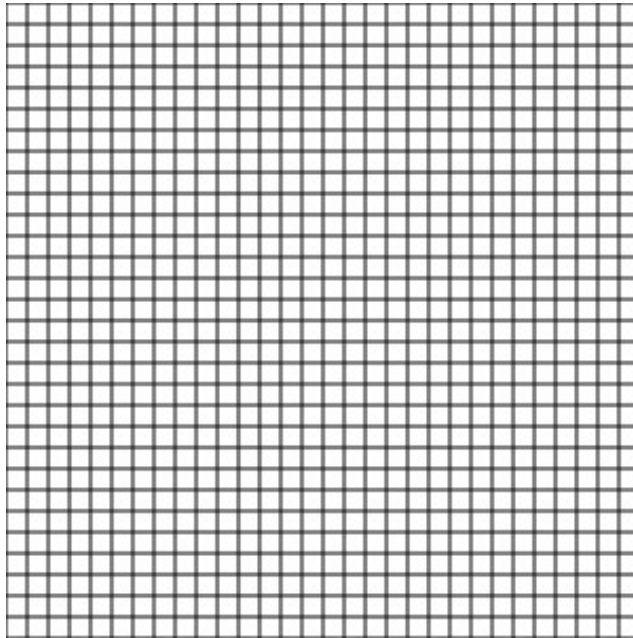
function hLines(ctx) {
    var y = ctx.canvas.height;
    while (y >= 0) {
        ctx.moveTo(0, y);
        ctx.lineTo(ctx.canvas.width, y);
        y = y - 10;
    }
}

var canvas = document.getElementById('drawing'),
    ctx = canvas.getContext('2d');

vertLines(ctx);
hLines(ctx);
ctx.stroke();
</script>

```

Note how the `hLines` function now keeps track of the current `y` coordinate, it appears in the second argument of `moveTo` and `lineTo` instead of the first.



It is also valid for the student to use `x -= 10` instead of `x = x - 10`, and similarly for `y`, if desired. Basically, if it works and is not overly convoluted, it is a good answer. Loops are an essential component of the right answer, though, and the student should be able to change the size of the canvas without breaking the program.

Do ensure that making the canvas bigger in either direction still produces a functioning program.

Exercise 7-6: While Loops and Arrays

Write a function that, using a `while` loop, computes the sum of all elements in a given array. The function can be called whatever you want, but it should accept exactly one argument: the array.

Answer 7-6

There are various good answers and each has its own variations. An efficient implementation that does not change the underlying array, but uses two variables to track progress, is this:

:javascript:

```
function sumArray(ar) {  
  var i = 0,  
      sum = 0;  
  while (i < ar.length) {  
    sum += ar[i];  
    i++;  
  }  
}
```

```
    return sum;
}
```

You can see that we start out at index `i = 0` and add one to it each time through the loop. The loop terminates when it is no longer true that `i < ar.length`. Each iteration simply adds the current value to `sum`, and the final value of `sum` is returned.

Another approach would be to use `pop`, but this changes the array that the caller passes in, so the student should be made aware of that fact. The solution is valid, but it might be an important teaching moment.

:javascript:

```
function sumArray(ar) {
    var sum = 0;
    while (ar.length > 0) {
        sum += ar.pop();
    }
    return sum;
}
```

This works because calling `pop` reduces the length of the array by 1 each time - it returns *and* removes the last element.

Yet another approach that leaves the original intact, and uses the parameter `ar` as a mutable variable, is this:

:javascript:

```
function sumArray(ar) {
    var sum = 0;
    while (ar.length > 0) {
        sum += ar[0];
        ar = ar.slice(1);
    }
    return sum;
}
```

This does *not* change the array for the caller because it does nothing that mutates the actual array, but it *does* reassign the local variable `ar` to smaller and smaller slices over time. Since `ar` is local to the function `sumArray`, that is safe and does not mess with values passed in. This is not a very efficient function (and neither is the one that uses `pop`) compared to the first, but it suffices as an answer.

Exercise 7-7: While Loop and Graphing Functions

A really interesting thing to do with loops is function graphing. When you go to graph a function like $f(x) = x^2 - 3$, what do you usually do? You probably make something like table of x and $f(x)$ values, draw dots at all of those locations, and then connect them with lines. It turns out that computers are really good at that.

To start, here is some skeleton code. Your task will be to implement the function that plots $f(x) = x^2 - 3$. We have appropriately transformed the canvas coordinates to make this sensible (positive y values go *up*, and the origin is in the *middle* now), but without explanation. If you want to learn more about canvas transforms, we will get to that near the end of the course.

Notes and requirements:

- The function is $f(x) = x^2 - 3$.
- The function should be plotted for integer values $-30 \leq x < 30$.
- Use `lineTo` to draw line segments (remember that the first `lineTo` acts like a `moveTo`, which is helpful here).

```
<canvas id="graph" width="300" height="300"></canvas>
<script>
var canvas = document.getElementById("graph"),
    ctx = canvas.getContext("2d");

function graph(ctx) {
    // YOUR LOOP GOES HERE
}

// Move origin to center, flip y.
ctx.translate(canvas.width / 2, canvas.height / 2);
ctx.scale(1, -1);

graph(ctx);
ctx.stroke();
</script>
```

Answer 7-7

The program should look something like what follows. Students are of course encouraged to explore and experiment, but the requested program is below, with its output displayed:

:html:

```
<canvas id="graph" width="300" height="300"></canvas>
<script>
var canvas = document.getElementById("graph"),
    ctx = canvas.getContext("2d");

function graph(ctx) {
    var x = -30;
    while (x <= 30) {
        var y = x*x - 3;
        ctx.lineTo(x, y);
        x++;
    }
}
```

```
ctx.translate(canvas.width / 2, canvas.height / 2);  
ctx.scale(1, -1);  
  
graph(ctx);  
ctx.stroke();  
</script>
```



Note that the student's code is entirely inside of the `graph` function. The code is small, but might take some thinking on the student's part. The flow is this:

- Start `x` at `-30`, then increment it by `1` each time through the loop (a different increment might also be used, and makes for a nice experiment).
- Each time through the loop, compute `y`, then draw a line from wherever we last were to that new point.

If the student got this basic structure down and the program works, that passes.

Exercise 7-8: While With Break

Without using the computer, what does the following code display? Remember that alerting an array converts it to a comma-separated list of its contents.

```
var i = 0,  
    vals = [];  
while (true) {  
  if (i > 10) {  
    break;  
  }  
}
```

```
    vals.push(i);  
    i++;  
}  
alert(vals);
```

Answer 7-8

The loop condition is always true, so we have to figure out what makes it stop. There is a `break` in there that triggers if `i > 10`, so we should see values in our output for everything up to but not including 11:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Exercise 7-9: While With Continue

Without using the computer, what does the following code display? Remember that `%` is the “modulus” or “remainder” operator, so for example, `5 % 3` is 2: the remainder of dividing 5 by 3.

As a second part to this question, what happens if the `i++` immediately before `continue` is removed?

```
var i = 0,  
    vals = [];  
while (i < 10) {  
    if (i % 2 === 0) {  
        i++;  
        continue;  
    }  
    vals.push(i);  
    i++;  
}  
alert(vals);
```

Answer 7-9

The loop stops when `i` is at 10, so we know we will not have any values at 10 or higher. But, before we push anything onto our `vals` array, we check to see whether `i % 2 === 0`. When does that happen? It happens whenever `i` is even, because that is when it divides evenly by 2 (no remainder). Thus, we should only get *odd numbers*: every even number causes a `continue` to happen. The alert therefore shows

1, 3, 5, 7, 9

The second part to this question is pretty interesting. If the `i++` before the `continue` is removed, then `i` does not change, but the condition is checked again. Therefore, once we hit our first even number (0), the loop *never terminates*. It just keeps continuing to check the

same old condition over and over again. This is an important concept: whenever checking the condition on a `while` loop, you want to make sure something about it is changing in such a way as to ensure eventual completion. Without that `i++` before `continue`, `i` is always 0 and is therefore always less than 10.

Exercise 7-10: Basic For Loops

Show the `for` loop that is equivalent to the following code that uses `while`:

```
var i = 0;
while (i < 50) {
  console.log(i);
  i += 10;
}
```

Answer 7-10

This is a pretty straightforward transformation and is covered in the text:

:javascript:

```
for (var i = 0; i < 50; i += 10) {
  console.log(i);
}
```

Exercise 7-11: For Loops and Arrays

Write a loop that outputs (to the console, one at a time) the value of every item in an array named `ar`.

Answer 7-11

:javascript:

```
for (var i = 0; i < ar.length; i++) {
  console.log(ar[i]);
}
```

Working through this one step at a time, the loop says that we start with `i = 0`, we keep going so long as we have not hit `ar.length` yet, and every time through the loop we increment `i` by 1.

Inside the loop, we just get the value of `ar` at `i` and output that to the console.

Exercise 7-12: For and Continue

Without using the computer, show the value that will be alerted by the following code. Remember that when dealing with numbers, 0 is considered “falsy” and *everything else* is

considered “truthy”. Coupled with your understanding of the `%` operator, you can see when `continue` will be triggered.

After determining what this function does, give an alternative approach that just changes the increment and does *not* use `continue`.

```
for (var i = 0; i < 10; i++) {
  if (i % 2) {
    continue;
  }
  console.log(i);
}
```

Answer 7-12

The output of this code is

0, 2, 4, 6, 8

The reason is this: whenever `i` is odd, then dividing by 2 produces a remainder of 1. That value is truthy, so it triggers the `continue` statement.

Because this is a `for` loop and not a `while` loop, the `continue` statement causes `i++` to be run before the condition is checked again, so it would be incorrect to have a separate `i++` before the `continue` like we did in the `while` loop above.

If the code doesn’t `continue`, then it goes on to log the value, which must be even. Thus, we get even numbers from 0 through 8. The loop exits without doing anything once `i` reaches 10.

With that understanding, the second part is easy: simply add 2 each time through the loop and remove the `continue` part altogether:

:javascript:

```
for (var i = 0; i < 10; i += 2) {
  console.log(i);
}
```

Exercise 7-13: Nested Loops

Output all of the rows of a truth table with three variables `a`, `b`, and `c`. Use 0 for `false` and 1 for `true`. Use `console.log(a, b, c)` to output each row.

Your output should look like this:

:console:

```
0 0 0
0 0 1
0 1 0
```

```

0 1 1
1 0 0
1 0 1
1 1 0
1 1 1

```

Hint: Approach this by decomposing the problem into smaller ones. How would you output the table for a single variable using a loop?

:console:

```

0
1

```

Now, for each time through *that* loop, how would you output a 0 and a 1 for the next variable to produce this?

:console:

```

0 0
0 1
1 0
1 1

```

From there it should be a similar step to get the complete 3-variable answer.

For bonus points:

There is a way to output a table for *any* number of variables using only two loops, so if you want to try for bonus points, you can write a function that accepts a number of variables and outputs the entire table for that many variables. The foundation of this idea is the fact that every time a variable on the right “rolls over” (from 1 to 0), the variable next to it on the left should change.

Answer 7-13

The key, as noted in the question, is to think about decomposing this problem a bit. Counting in binary is a matter of finding all possible combinations of 0 and 1 in an orderly and predictable fashion. If we only had a single variable, we would just loop through 0 and 1. But with two variables we need to do that twice, once for each of 0 and 1 in the leftmost variable. You can see that pattern in the tables above.

Here is code that produces the table:

:javascript:

```

for (var a = 0; a < 2; a++) {
  for (var b = 0; b < 2; b++) {
    for (var c = 0; c < 2; c++) {
      console.log(a, b, c);
    }
  }
}

```



```

    }
}

```

As an interesting aside, when you start at 0 and increment by 1 as is common in these loops, then the number checked in the condition is *the number of times the loop will execute*. Thus, each variable is checked to be less than 2. We could as easily have said `a <= 1` and been correct, as well. It is just less common, and therefore less idiomatic.

That seems like rather a lot of code to generate an 8-line table, and it kind of is. But consider what it would take to change it to generate a 32-line table for 5 boolean variables:

:javascript:

```

for (var a = 0; a < 2; a++) {
  for (var b = 0; b < 2; b++) {
    for (var c = 0; c < 2; c++) {
      for (var d = 0; d < 2; d++) {
        for (var e = 0; e < 2; e++) {
          console.log(a, b, c, d, e);
        }
      }
    }
  }
}

```

The length of the code increases much more slowly than the length of the table. That's kind of neat.

There is an even neater trick for generating a table with *any* number of variables using only two loops. The idea is to set up an array of values, with as many entries as there are variables (so an array that is as large as one row of the table). Every entry starts out as zero, and then you start counting by flipping the rightmost value, checking whether it rolled over, and if it did, flipping the next one to the left, continuing all the way down. When leftmost variable rolls over, the process is finished, since it has already taken on both of its possible values. Here is a functioning program:

:javascript:

```

function truthTable(numVars) {
  var values = []; // Make an array of zeros.
  for (var i = 0; i < numVars; i++) {
    values.push(0);
  }

  // Now we output our row, then calculate a new one,
  // until the leftmost item wants to roll over again.
  while (true) {
    console.log(values.join(' ')); // output with spaces
    for (var i = values.length-1; i >= 0; i--) {
      values[i]++;
    }
  }
}

```

```

    if (values[i] === 1) {
        // All done creating new row, because we were
        // able to increment without rolling over.
        // Stopping the inner loop lets the outer loop
        // body execute, which prints this new one.
        break;
    }
    // We went too high, roll over.
    // Inner loop will try again, one to the left.
    values[i] = 0;
    // If we rolled over the leftmost, finish.
    if (i === 0) {
        return;
    }
}
}
}

truthTable(4); // Try calling with different values.

```

Exercise 7-14: Switches

Write a function that uses `switch` to return “yes” if given a string “a”, “b”, or “c”, and “no” otherwise. Use `default` as one of your cases.

Note that you can use a “fall-through” pattern to make your code smaller, and empty cases can go on the same line.

Can you do this without using `break` at all? Why?

Answer 7-14

There are a few ways to go about this, but the hints point us to this particular solution (the function and parameters can be named anything):

```

:javascript:

function(s) {
    switch(s) {
        case "a":
        case "b":
        case "c":
            return "yes";
        default:
            return "no";
    }
}

```

Note that `break` doesn’t appear above at all, even though it is usually needed in a correct `switch` statement. That is because of two things:

1. We are using fall-through for cases that are all alike in outcome, and
2. Every terminal outcome ends with `return`, which accomplishes the same thing and more.

That last point is the main reason we don't need `break`: we are executing `return`, which terminates the entire function, `switch` statement included.

Exercise 7-15: Random Numbers

To produce pseudo-random numbers, JavaScript gives us the `Math.random()` library function, which gives us a different number between 0 (inclusive) and 1 (exclusive) every time we call it. Using that function and basic arithmetic, write a function `dieRoll` that represents a single roll of a 6-sided die: it should produce a number in the set $\{1, 2, 3, 4, 5, 6\}$.

Hint: remember that `Math.floor` can be used to discard everything to the right of the decimal place (for non-negative numbers).

Answer 7-15

To do this, we first need to scale (stretch) our random result so that it produces values from 0 to (almost but not quite) 6, like this:

:javascript:

```
Math.random() * 6;
```

Then we add 1 and take the floor (or vice versa - order is not important here), giving us this:

:javascript:

```
function dieRoll() {
  return Math.floor(Math.random() * 6 + 1);
}

// Test it by rolling ten times:
for (var i = 0; i < 10; i++) {
  console.log(dieRoll());
}
```

Exercise 7-16: Lab: Objects, Randomness, and the Canvas

We are far enough into the course that it is time to start introducing some lab work. This assignment brings together many of the things that we have already learned, including randomness, canvas drawing, and storing data in arrays.

Basic Setup:

For this assignment we are going to plot a distribution over dice rolls. We will use our `dieRoll` function created in the recent homework assignment to roll *two* 6-sided dice over

and over again, keeping track of how many times we get each sum. For example, to get a single value, we can do this:

```
var val = dieRoll() + dieRoll();
```

And we might get the number 5 as an answer. That is a valid number for two dice, and we can get it as 1+4 or 2+3, with two ways to get each (because it doesn't matter which die has which value). As you might expect if you have ever played games with dice, other numbers are more or less likely. For example, there is only one way to get the number 2, and similarly there is only one way to get the number 12. This should show up in our lab results.

In this example, where we just rolled a 5, we need to keep track of the fact that we just got that one more time. So, we will keep a counter for each possible number, and we will increment it every time we get that number. There are many ways to do this, but for this assignment we will use an array with everything initialized to zero.

Once you have rolled the dice a number of times (say, 1000 or so), the next task is to display the results in a meaningful way, using a **histogram**. This is basically just a bar graph. You will need to figure out how long to make each bar based on the counts obtained from rolling dice. That might mean scaling your results, since it is possible to have more counts than you have space on the canvas. How you do that will be up to you. If you roll the dice 1000 times, then one pixel per roll is probably going to work out fine on our (heretofore) typical 300 x 300 canvas size. You might want to get more accuracy as you go, though, rolling 10,000 or even 100,000 times, and in that case the counts will increase far beyond the bounds of the canvas. Figure out some sort of scaling that keeps them within bounds for this lab so that you can easily change the number of rolls and have the picture automatically adapt.

Some advice: start small. First get your `dieRoll` function working and test it in the console (you should have had a homework assignment for this already, but if you didn't end up doing that, you will need to do it now). Then log what happens when you roll two dice over and over again. After you are satisfied that those small things work, then create an array that has space for numbers up to 12 and store zeros in it everywhere (use `push` in a loop). Output that to the console.

Do you see a pattern? The advice is to do things one step at a time. There are usually ways to tell if little pieces of your program are working before moving on to the bigger parts, and that is how you should proceed. We are stopping short of talking about unit tests, which are a great idea but are a bit beyond where we are right now, but this is part of the way you get testable code that you can be confident in: you get confidence in all of the little pieces before sticking them together into a larger program.

After getting counts, storing them in an array, and outputting to the console, only then should you start thinking about displaying things on the canvas. That will be its own chunk of work, and you want to be sure you have a good feel for what the data will look like before you tackle it.

Requirements:

- Keep track of how many times each value is obtained after rolling two dice multiple times.
- Use an array to store your counts. The index into the array will be the value obtained from rolling two dice. (HINT: this means that there will be a couple of entries that never get any value at all.)
- Use a variable to set how many times to roll, and make it easy to change it.
- Draw a bar graph on a canvas to show the final outcome of the dice rolls.
- Scale the bar graph so that no matter how many times you roll, the graph always fits on the canvas.

Bonus:

- Change your `dieRoll` function to accept a number of sides for the die instead of always assuming 6. Try using larger dice, like 10-sided dice. Assume that they are made to be perfectly fair (sometimes physical geometry doesn't allow this, by the way, but we will pretend that it does in this lab).
- Change your program to roll *three* dice and display the results.
- With that change, it should be relatively easy to see how you could roll *any number* of dice. Change your code to roll `N` dice. What happens if you roll 50 or 100 dice on each turn?

Answer 7-16

The first thing to do is to make sure we have a function that can roll a single die:

:javascript:

```
function dieRoll() {  
  return Math.floor(Math.random() * 6 + 1);  
}
```

With that, we can test it in the console quite easily. Running it multiple times will always produce a valid 6-sided die outcome, a number between 1 and 6 with even odds of getting any of them.

The next step is to make it possible to roll a number of times and output all of the answers, something perhaps like this:

:javascript:

```
var ROLLS = 100;  
  
for (var i = 0; i < ROLLS; i++) {  
  console.log(dieRoll() + dieRoll());  
}
```

That should output 100 different results to the console, each of them a number between 2 and

12 (inclusive). They should look like what you would expect from rolling two independent 6-sided dice.

After that, getting counts is pretty easy. We first create our counts array and initialize it to contain all zeros:

:javascript:

```
var counts = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0];
```

Question: when doing bonus work, what would we do instead? If we change the number of sides on each die, we won't have 13 entries anymore like we do above. Similarly, if we use 3 or more dice, we will have a different number of entries. 13 is a pretty magical number, so where did it come from?

Any time you see magic numbers like this, you need to ask that question. Let's try to answer it for the number 13: since we have two 6-sided dice, the maximum value we can get is $6+6$, which is obviously 12. The slot in an array with index 12 is actually the 13th slot, so our array has 13 zeros in it. The first two slots will always be zero, since the *smallest* value you can get from rolling two 6-sided dice is 2. We could shift things back and forth to save space, but that seems like overkill here, so we just allocate 13 slots.

So, now we know where our magic array length comes from. That is going to be very important when working on lab requirements like "add more dice" or "change the number of sides". The size of our counts array is always going to be one more than the sum of all possible maximum die values, and the first usable index in that array will always be $1 * \text{DICE}$, since that is the smallest number you can get (they all come up 1). If we have three 8-sided dice, then the length will be 25, and the first usable index will be 3. If we have 2 dice of different sizes, say one has 6 sides and the other has 10, then the length will be $6 + 10 + 1 = 17$.

Do you see how knowing where our magic number came from just gave us something really useful? If we know where it comes from, we can *generalize our solution*.

With an empty counts array, we can easily keep track of all of our rolls now, and we will crank it up to 1000:

:javascript:

```
var ROLLS = 1000;
var counts = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0];

function dieRoll() {
    return Math.floor(Math.random() * 6 + 1);
}

for (var i=0; i<ROLLS; i++) {
    var val = dieRoll() + dieRoll();
    counts[val]++;
}
```

```
console.log(counts);
```

That's not too bad! Now we can easily get a histogram of rolls, and we can easily change the number of rolls with a single variable. The next step is to figure out how to plot this in a meaningful way. Before worrying at all about scaling and making things pretty, just do the simplest possible thing: we will use the index into the array as the x coordinate and the number in the array as the maximum y coordinate, and we will draw a slender rectangle to show what we have done:

```
:html:
```

```
<canvas id="graph" width="300" height="300"></canvas>
<script>
var canvas = document.getElementById('graph'),
    ctx = canvas.getContext('2d');

var ROLLS = 1000;
var counts = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0];

function dieRoll() {
    return Math.floor(Math.random() * 6 + 1);
}

for (var i=0; i<ROLLS; i++) {
    var val = dieRoll() + dieRoll();
    counts[val]++;
}

for (var x=0; x<counts.length; x++) {
    var count = counts[x];
    ctx.fillRect(x, 0, 1, count);
}
</script>
```



That should basically work. You will see an upside-down triangle sort of thing, which is the histogram we wanted. It's pretty darn ugly, but it's there. It will look different every time you reload the page, because we are dealing with random events.

This next part is where we make things prettier and a bit more robust. There is nothing in our code that spreads things out horizontally, so everything is super squished together along the x axis. Let's tackle that issue first, by making wider rectangles that fill the whole canvas. We can do that by dividing our canvas length by the number of entries in our counts array and using that value (`rWidth` below) as the width. Note that we no longer iterate over `x`, but instead we calculate it from the width as well. We will also flip this right-side up by starting the rectangle at `canvas.height-count` instead of 0:

:html:

```
<canvas id="graph" width="300" height="300"></canvas>
<script>
var canvas = document.getElementById('graph'),
    ctx = canvas.getContext('2d');

var ROLLS = 1000;
    counts = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0];

function dieRoll() {
    return Math.floor(Math.random() * 6 + 1);
}

for (var i=0; i<ROLLS; i++) {
    var val = dieRoll() + dieRoll();
    counts[val]++;
}
```

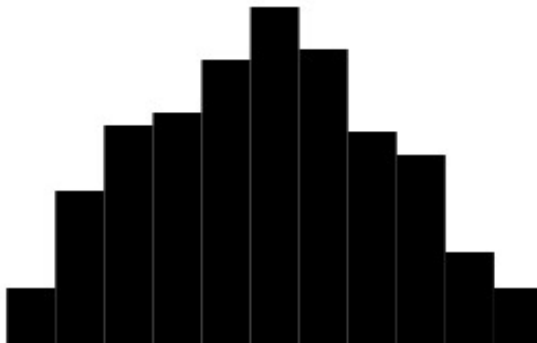


```

}

var rWidth = canvas.width / counts.length;
for (var i=0; i<counts.length; i++) {
    var x = i*rWidth;
    var count = counts[i];
    ctx.fillRect(x, canvas.height-count, rWidth, count);
}
</script>

```



That is really a lot better, but we are shifted to the right because we are showing values for impossible counts (0 and 1). Let's fix that, as well:

:html:

```

<canvas id="graph" width="300" height="300"></canvas>
<script>
var canvas = document.getElementById('graph'),
    ctx = canvas.getContext('2d');

var ROLLS = 1000;
counts = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0];

function dieRoll() {
    return Math.floor(Math.random() * 6 + 1);
}

for (var i=0; i<ROLLS; i++) {
    var val = dieRoll() + dieRoll();
    counts[val]++;
}

```

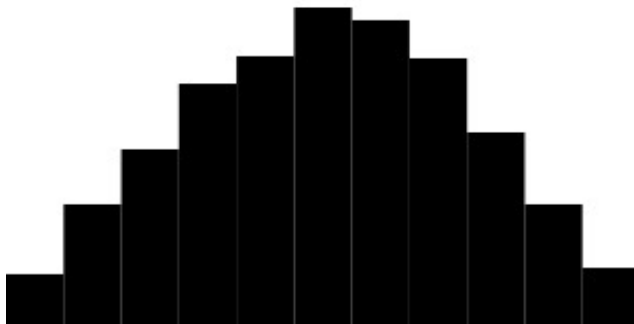
```

}

// Get rid of useless count entries.
counts = counts.slice(2);

var rWidth = canvas.width / counts.length;
for (var i=0; i<counts.length; i++) {
    var x = i*rWidth;
    var count = counts[i];
    ctx.fillRect(x, canvas.height-count, rWidth, count);
}
</script>

```



We would be done, except that we need to handle more than 1000 rolls of the dice. If we increase it to, say, 5000, then the graph will go off the top edge of the canvas, and that doesn't really help us. We need to scale our results. We just did that in the x direction, and now we need to do it in the y direction. There are a couple of ways to do this. We could find the maximum value and make that always take up all of the vertical space, or we can scale based on number of rolls.

To keep it simple, let's compute the maximum value and scale based on that. We have to search our `counts` array to get the biggest value, then associate that with the largest rectangle size. That means it becomes a standard, and every other value is represented as a fraction of that standard.

As an example, suppose we have 1000 for our count value at position 7, and that is our biggest one (it usually will be—can you see why?). Let's say that has a height of 1. If we have a count of 720 for position 6, then that rectangle will have a height of $720 / 1000$ which is

.72. If we take `canvas.height` to be the tallest value, which only makes sense, then we multiply those fractions by `canvas.height` to get the actual top of the rectangle. Here is a snippet of how that would be calculated:

:javascript:

```
var maxVal = 0;
for (var i=0; i<counts.length; i++) {
  if (maxVal < counts[i]) {
    maxVal = counts[i];
  }
}
```

Then, every time we get a count in our `fillRect` loop, we compute the height like this:

:javascript:

```
var height = canvas.height * count / maxVal;
```

Now we just use `height` instead of `count` in `fillRect`. That's not too bad! A complete program is below.

:html:

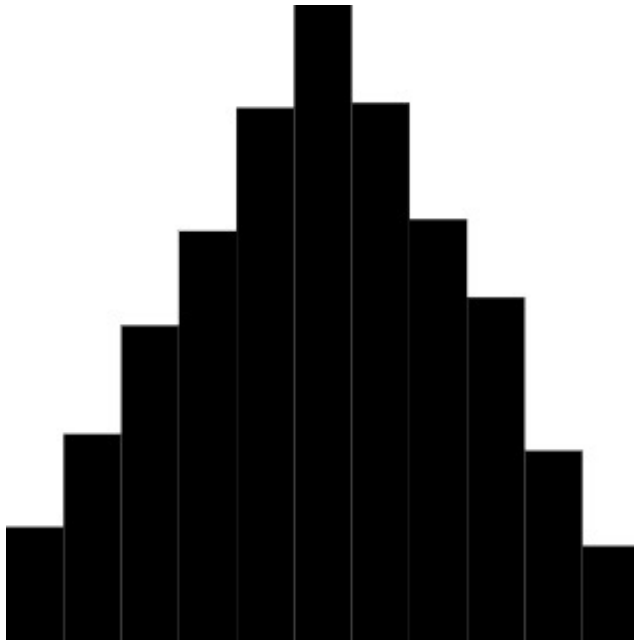
```
<canvas id="graph" width="300" height="300"></canvas>
<script>
var canvas = document.getElementById('graph'),
    ctx = canvas.getContext('2d'),
    ROLLS = 10000,
    counts = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0];
function dieRoll() {
  return Math.floor(Math.random() * 6 + 1);
}

for (var i=0; i<ROLLS; i++) {
  var val = dieRoll() + dieRoll();
  counts[val]++;
}

counts = counts.slice(2); // delete useless entries
var maxVal = 0;
for (var i = 0; i < counts.length; i++) {
  if (maxVal < counts[i]) {
    maxVal = counts[i];
  }
}

var rWidth = canvas.width / counts.length;
for (var i = 0; i < counts.length; i++) {
  var x = i*rWidth,
      count = counts[i],
      height = canvas.height * count / maxVal;
  ctx.fillRect(x, canvas.height-height, rWidth, height);
}
```

```
}
</script>
```



We have satisfied all of the minimum requirements. Let's dive into some of the bonus questions, too.

To change the number of dice we roll, we will want to trade our basic sum to get `val` for a loop, and we will need to be careful about how many useless small values we throw away. Then we can get graphs with lots more entries. Note that we will need to greatly increase the size of `counts`, as well, like we discussed earlier:

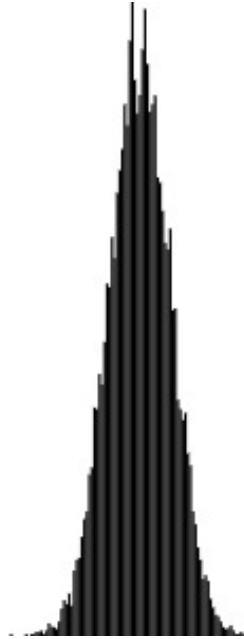
:javascript:

```
var DICE = 50;
var counts = [];

// Initialize counts:
for (var i=0; i<=6*DICE; i++) {
    counts.push(0);
}

// Get actual values.
for (var i=0; i<ROLLS; i++) {
    var val = 0;
    for (var j=0; j<DICE; j++) {
        val += dieRoll();
    }
    counts[val]++;
}
```

```
// Get rid of useless count entries.
counts = counts.slice(DICE);
```



That handles the case of multiple dice. Now we need to be able to change the number of sides. To do that, we note that we have the number 6 baked into our code in a few places. What if, instead of using the number everywhere, we made a constant for it, like `SIDES = 6` and used that instead? Would the code work if we changed it to something else? Let's take a look at a complete solution (note that some vertical space has been removed and some lines condensed to help it fit on a page, and we have omitted the canvas and script tags):

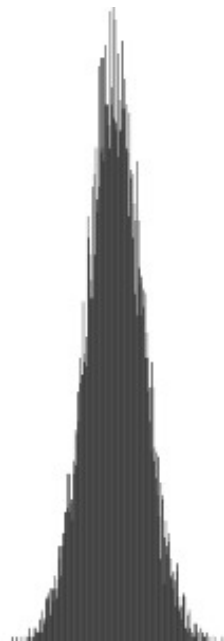
:javascript:

```
var canvas = document.getElementById('graph'),
    ctx = canvas.getContext('2d'),
    ROLLS = 10000, DICE = 50, SIDES = 15, counts = [];

function dieRoll() {
    return Math.floor(Math.random() * SIDES + 1);
}

for (var i=0; i<=SIDES*DICE; i++) {
    counts.push(0);
}
for (var i=0; i<ROLLS; i++) {
    var val = 0;
    for (var j=0; j<DICE; j++) {
        val += dieRoll();
    }
    counts[val]++;
}
counts = counts.slice(DICE); // delete useless entries
```

```
var maxVal = 0;
for (var i=0; i<counts.length; i++) {
  if (maxVal < counts[i]) {
    maxVal = counts[i];
  }
}
var rWidth = canvas.width / counts.length;
for (var i=0; i<counts.length; i++) {
  var x = i*rWidth,
      count = counts[i],
      height = canvas.height * count / maxVal;
  ctx.fillRect(x, canvas.height-height, rWidth, height);
}
```



Chapter 8 Solutions

Exercise 8-1: Practice with setTimeout

Write a function that draws a rectangle at a random location on a canvas every quarter second (250 milliseconds). The rectangle can be any color and the size of your choosing, so long as it is easy to tell that the program is working.

For bonus points, make the color random as well, so a different color can appear each time.

Answer 8-1

There is a lot of wiggle room here. A minimal solution might look like this:

:html:

```
<canvas id="randrect" _ width="300" _ height="300"></canvas>
<script>
var canvas = document.getElementById("randrect"),
    ctx = canvas.getContext("2d");

function randRect() {
    var x = Math.random() * canvas.width,
        y = Math.random() * canvas.height;
    ctx.fillRect(x, y, 10, 10);
    setTimeout(randRect, 250);
}

randRect();
</script>
```

You might choose to make the rectangle sizes random, as well, which can be fun. Colors can be made random by using a trick we have already seen, something like this program:

:javascript:

```
var colors = ["red", "blue", "yellow", "orange"];
function randRect() {
    var x = Math.random() * canvas.width,
        y = Math.random() * canvas.height,
        c = colors[Math.floor(Math.random() * colors.length)];

    ctx.fillStyle = c;
    ctx.fillRect(x, y, 10, 10);
    setTimeout(randRect, 250);
}
```

There are many interesting options.

Exercise 8-2: Scope and Closure

Use the following code to answer the questions below:

```
<script>
var a = 10;

function F() {
  // code here
}

var b = 15;

function G() {
  c = "hi";

  function H() {
    var d = [];
    // code here
  }

  var e = "there";

  function I() {
    var f = {};
    // code here
  }
}
</script>
```

- List every variable and function name below that is in the global scope. Assume that all functions have run at least once (and remember what happens when you leave off `var`).
- Can code inside of `H` access `f`? Why or why not?
- Can code inside of `I` call `H`? Why or why not?
- Can code inside of `H` call `F`? Why or why not?
- Can code inside of `H` access `e`? Why or why not?
- Can code inside of `F` call `H`? Why or why not?

Answer 8-2

Global scope Opening a `script` block puts you right into the global scope. Thus, all of these are in that scope: - `a` - `F` - `b` - `G` - `c` - no `var` keyword, so in the global scope

H -> f No, because `f` is not in the surrounding scope of `H`. It is hidden inside of `I`.

I -> H Yes, because `H` is in the surrounding scope of `I`.

H -> F Yes, because `F` is in the global scope.

H -> e Yes, because `e` is in its surrounding scope, even though it comes later.

F -> H No, because the only things in the global scope are `a`, `F`, `b`, and `G`. Nothing else is visible to `F` except its own local scope.

Exercise 8-3: Animation

Write a program that draws a rectangle on the left side of the canvas and moves it to the right until it reaches the edge. The rectangle should be of size 10 by 10, and should move 5 pixels every 1/20th of a second (50 milliseconds). It can be anywhere along the *y* axis (centered top to bottom, on the bottom, on the top, or anywhere in between).

Answer 8-3

There is some wiggle room in how the rectangle looks, etc., but there are some rigid requirements. The following code meets the minimum standard:

.html:

```
<canvas id="slider" width="300" height="300"></canvas>
<script>
var canvas = document.getElementById('slider'),
    ctx = canvas.getContext('2d');

var x = 0,
    SIZE = 10,
    y = (canvas.height - SIZE) / 2;

function tick() {
  if (x > canvas.width) {
    return;
  }
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  ctx.fillRect(x, y, SIZE, SIZE);
  x += 5;
  setTimeout(tick, 50);
}

tick();
</script>
```

The above function centers the rectangle from top to bottom, but that is not required. The essential elements are

- It starts at the left (`x = 0`),
- The rectangle side lengths are 10 (`SIZE = 10`),
- It stops when it reaches the edge (`if (x > canvas.width)`),
- It advances by 5 pixels every frame (`x += 5`), and
- It advances once every 50 milliseconds (`setTimeout(tick, 50)`).

Exercise 8-4: Shadowing

What is displayed by the following program? Why?

```
var a = "hello";

function message(a) {
  console.log("message is:", a);
}

message("hi");
```

Answer 8-4

The program outputs this:

```
                                :console:
message is: hi
```

The reason is that the `a` defined in the parameters for `message` *shadows* the `a` defined outside. Because there is a name conflict, the nearest scope wins, and `message` can no longer see the outer `a` from the code in its body.

Exercise 8-5: Lab: Animate a Die Roll

For this lab you will animate the roll of a die. It won't be a physically accurate simulation; it will just display random dots a few times until it settles on a value. The dots will be rectangles, and the code to draw a single dot based on where it is on a 3x3 grid is given below to make it a bit easier on you.

```
// drawDot draws a 'dot' at the given row and column,
// assuming that size is set to the square side length
// of the die. Both row and col are zero-based, so
// 0, 0 is the top left corner and 2, 2 is the bottom right.
function drawDot(ctx, size, row, col) {
  var margin = size / 6,
      x = margin + (2*margin*col),
      y = margin + (2*margin*row);
  ctx.fillRect(x-margin/4, y-margin/4, margin/2, margin/2);
}

// dieDots defines where the dots are located for every
// possible value on a 6-sided die, by row and column.
var dieDots = {
  1: [{r: 1, c: 1}],
  2: [{r: 0, c: 0}, {r: 2, c: 2}],
  3: [{r: 0, c: 0}, {r: 1, c: 1}, {r: 2, c: 2}],
  4: [{r: 0, c: 0}, {r: 0, c: 2}, {r: 2, c: 0},
      {r: 2, c: 2}],
  5: [{r: 0, c: 0}, {r: 0, c: 2}, {r: 2, c: 0},
      {r: 2, c: 2}, {r: 1, c: 1}],
  6: [{r: 0, c: 0}, {r: 1, c: 0}, {r: 2, c: 0},
```

```

    {r: 0, c: 2}, {r: 1, c: 2}, {r: 2, c: 2}],
  };

```

With these functions, you should be able to easily define a `drawDie` function that accepts a context, a size, and a value (the number of dots you want to show), like this:

```

function drawDie(ctx, size, value) {
  // Your code goes here, using drawDot and dieDots above.
}

```

And finally, with that, you can animate a die so that it picks a new random value every 200 milliseconds to display, and stops on the tenth one.

If you get this working quickly and just use random draws to create your values (that’s a good idea—start with that!), you might notice that sometimes you get the same number twice in a row, and it makes the animation look like it paused. Add code to ensure that you never repeat the same number twice in a row.

Bonus: make sure you never go directly to a number on the *opposite side* of the die, either. You can easily check this because numbers on opposite sides always sum to 7.

Answer 8-5

The tricky drawing stuff is pretty much done for you, but there is that `drawDie` function that has to be written. This is how it works:

:javascript:

```

function drawDie(ctx, size, value) {
  var locations = dieDots[value];
  for (var i=0; i<locations.length; i++) {
    var loc = locations[i];
    drawDot(ctx, size, loc.r, loc.c);
  }
}

```

Basically, it gets the list of locations for the appropriate value from `dieDots`. If, for example, `value` is 3, then it gets the list of three dot locations, each of which is an object containing `r` for “row” and `c` for “column”. It calls `drawDot` for each of those locations, ultimately creating three dots in the expected places.

To animate this, something like the following can be done:

:javascript:

```

var SIZE = canvas.height,
    valuesLeft = 10,
    lastVal = 0;

function animate() {
  // Are we done? No more animating.

```

```

    if (valuesLeft === 0) {
        return;
    }
    valuesLeft--;

    // Get the value.
    var value = Math.floor(Math.random() * 6) + 1;
    // If we just saw it, get another one until it's new.
    while (value === lastVal) {
        value = Math.floor(Math.random() * 6) + 1;
    }
    // Remember this value.
    lastVal = value;

    ctx.clearRect(0, 0, canvas.width, canvas.height);
    drawDie(ctx, SIZE, value);
    setTimeout(animate, 200);
}

animate();

```

This function is a little bit tricky because it has to try to find a new value if it collides with the most recent one. Other than that, it's pretty straightforward. Let's walk through it a step at a time.

First, we set the size to be the same as the canvas height. It can be anything, but this makes a nice large die animation.

We also note how many times we want to run this animation, and we set `lastVal` to 0, which is an impossible value for a die. That way the first one we try will never collide. That simplifies the logic a bit below.

In the `animate` function, we check to see if we are finished (no more `valuesLeft` to show). If we are, we simply return and leave up whatever is up. That is the final value of the roll. If we still have stuff left to do, we continue on and subtract one from `valuesLeft`. That way we will eventually get to zero.

The next little block is responsible for finding out what value to show. We get a random number in the range [1, 6]. If it's the same as the previous one (in `lastVal`), we loop, getting as many random numbers as it takes to stop colliding with the previous one. Because `lastVal` starts out at 0, it can never collide on the first roll.

As soon as we have a value, we remember it in `lastVal` for next time.

Finally, we clear the canvas and draw the die using our `drawDie` function. Then we call `setTimeout` to make sure that it happens again after 200 milliseconds.

For bonus points, we not only ensure that we don't get the same number twice in a row, but that we don't get a number on the opposite side, either. The updated random number code to

do this would simply add another check to the `while` loop; instead of merely checking that the number is the same as what we saw before, we *also* check whether the current and last number sum to 7:

:javascript:

```
var value = Math.floor(Math.random() * 6) + 1;
while (value === lastVal || value + lastVal === 7) {
  value = Math.floor(Math.random() * 6) + 1;
}
```

Pretty neat!

Finally, there is a way to do this a bit more concisely using a `do` loop:

:javascript:

```
var value;
do {
  value = Math.floor(Math.random() * 6) + 1;
} while (value === lastVal || value + lastVal === 7);
```

That is a perfectly acceptable solution, as well, and it only mentions the random value code once, which is why `do` loops exist in the first place.

Chapter 9 Solutions

Exercise 9-1: Trivia for `requestAnimationFrame`

- How often is the function given to `requestAnimationFrame` called if it sets up a new call every time?
- When is the function not called at all?
- What parameters are available when the function is called?
- Where in that function should the call to `requestAnimationFrame` reside (for animation), and why?

Answer 9-1

For the sake of clarity, we will call our function `tick`, and we will pass it thus: `requestAnimationFrame(tick)`. Thus, the function we register with the browser's drawing routines will be referred to as "the tick function".

- The tick function is called roughly 60 times per second, or roughly once every 16.7 milliseconds.
- The tick function is not called at all if the browser tab is not visible.
- The tick function can have a single formal parameter that receives the time in milliseconds since the page was loaded. We called this `t` in the text.
- The call to `requestAnimationFrame` should happen near the top of `tick`, before a lot of work has been done. This ensures that the work we do doesn't make us miss the deadline for asking to be called again, making us possibly miss a frame. It is not a guarantee, but it can help.

Exercise 9-2: Lab: Constant Acceleration

For this lab, change the ship's trajectory to act more like it was thrown upward in the presence of gravity. This means that it is *always* experiencing an acceleration downward, and it only has an *initial* velocity, not a *constant* velocity.

Requirements:

- Use an acceleration of 150.
- Use an initial velocity of -300.
- Stop the animation when the ship gets back to the bottom of the canvas.

Hints:

Acceleration is a *change in velocity*. Thus, if you have an acceleration of 5 pixels per second per second, that means that your velocity will change by 5 each second. If you have less than a second to work with (as is the case with our frame rate), then you can approximate reality by multiplying it by the amount of time that has passed.

Thus, we might calculate the current velocity like this: `yVel += ACCEL * dt`. With that, we can compute the new position just like we did before.

Answer 9-2

There are two parts to this: making the velocity change based on acceleration, and ensuring that the bottom of the ship doesn't go out of bounds. The changes are fairly simple. The change in position is calculated differently thus (assuming we have a new constant `ACCEL` and `yVel` becomes a variable):

:javascript:

```
// Somewhere above:
var ACCEL = 150,
    yVel = -300;

// Down in the tick function:
yVel += ACCEL * dt;
y += yVel * dt;
```

The pattern is pretty simple: we are approximating a constant acceleration by just adding it, scaled by time, to the current velocity. Then we do the same for position: add the velocity, scaled by time, to the current position.

To ensure that the ship doesn't go out of bounds when it falls back to earth, we simply add a check before calling `requestAnimationFrame` (or doing anything else, really):

:javascript:

```
if (y < 0 || y > canvas.height) {
    return;
}
requestAnimationFrame(tick);
```

We are basically making sure the bottom of the ship doesn't go out of bounds. If its position is at least 0 *and* (remember `||` means "OR" and `&&` means "AND") no more than `canvas.height`, we can keep animating.

If you recall De Morgan from earlier, you can see that these two expressions are equivalent:

- `(y < 0 || y > canvas.height)`
- `!(y >= 0 && y <= canvas.height)`

Intuitively, if you talk through it, it should make sense: "Testing whether `y` is out of bounds is the same as testing whether it is not in bounds."

The full listing follows:

:html:

```
<canvas id="drawing" width="300" height="300"></canvas>
<script>
var canvas = document.getElementById('drawing'),
    ctx = canvas.getContext('2d');
function drawShip(x, y) {
    ctx.fillStyle = 'blue';
    ctx.beginPath();
    ctx.moveTo(x, y-30);
    ctx.lineTo(x-5, y);
    ctx.lineTo(x+5, y);
    ctx.lineTo(x, y-30);
    ctx.fill();
}

var y = canvas.height, yVel = -300, lastTime = 0,
    ACCEL = 150;
function tick(t) {
    if (y < 0 || y > canvas.height) {
        return;
    }
    requestAnimationFrame(tick);

    var dt = (t - lastTime) / 1000;
    lastTime = t;

    ctx.clearRect(0, 0, canvas.width, canvas.height);
    drawShip(canvas.width/2, y);
    yVel += ACCEL * dt;
    y += yVel * dt;
}
requestAnimationFrame(tick);
</script>
```


Chapter 10 Solutions

Exercise 10-1: Key Events

Write a program that registers an event listener on the `document` object for the “keydown” event, and have it print the event’s `key` and `shiftKey` information to the console. Remember that every event listener receives an event object as its first parameter, and in this case we are interested in the `key` and `shiftKey` members of that object (capitalization matters, so be careful of that):

- What happens when you press ‘a’? What happens when you press ‘A’ instead?
- What happens when you press the shift key, with nothing else?
- How do arrow keys appear when pressed?
- Log the entire event instead of just two members of it. What happens when you press other modifiers like the control, alt, or command keys?

Answer 10-1

This is a pretty short program. You can even type it into the console because it’s easy to make it fit on one line if you want to get clever about it. The program looks like this:

:html:

```
<script>
document.addEventListener('keydown', function(e) {
  console.log(e.key, e.shiftKey);
});
</script>
```

If the student enters the JavaScript part into the console, after pressing , the main document has to be focused before the program will do anything. Click in the main document, start typing keys, and the log will show what’s happening.

For logging the entire event, you would simply do `console.log(e)` (or whatever the event variable is called).

Exercise 10-2: Click Events

Write a program that has a canvas. Each time you click the canvas, draw a line from the previous click location (start at 0, 0 by default) to the current click location. Note that you can use the event object’s `x` and `y` members to get the coordinates of the click within the canvas.

Answer 10-2

This program has a few parts:

- Create a canvas
- Register a click listener
- Draw lines

This is not an animation, so we don't need to request an animation frame or anything like that. We can just draw on the context directly every time a click happens.

An example program is given below. Note that the previous click position is kept track of in the variables `prevX` and `prevY`. Drawing moves to the previous location then draws a line to the current location, strokes the line, and finally remembers the current location for next time.

:html:

```
<canvas id="lines" width="300" height="300"
      style="border: 1px solid black"></canvas>
<script>
var canvas = document.getElementById("lines"),
    ctx = canvas.getContext("2d"),
    prevX = 0, prevY = 0;
canvas.addEventListener('click', function(e) {
    ctx.moveTo(prevX, prevY);
    ctx.lineTo(e.x, e.y);
    ctx.stroke();
    prevX = e.x;
    prevY = e.y;
});
</script>
```

Hopefully most of these concepts are becoming natural at this point. The above code exercises canvas drawing routines and creates an anonymous closure (with access to `prevX`, `prevY`, and `ctx`) to pass directly into `addEventListener`.

Exercise 10-3: More Mouse Events

Taking inspiration from the previous program, alter it so that it draws lines as before, but every time the mouse *moves* while the *button is down*. The events you will need to make this work are called “mousedown”, “mouseup”, and “mousemove”.

Hint: set a boolean to true when “mousedown” happens, and only draw in “mousemove” if it is true. Set it to false when “mouseup” happens.

Bonus: fix the program so that pressing the mouse button resets the line's starting point.

Answer 10-3

The drawing routine is largely the same as in the previous program, but the event in which it happens is “mousemove” instead of “click”. It also needs to check whether the mouse button is currently pressed before drawing, and should bail out early if not. A working listing is below:

:html:

```

<canvas id="lines" width="300" height="300"
        style="border: 1px solid black"></canvas>
<script>
var canvas = document.getElementById("lines"),
    ctx = canvas.getContext("2d"),
    prevX = 0, prevY = 0, buttonDown = false;
canvas.addEventListener('mousedown', function(e) {
    buttonDown = true;
});
canvas.addEventListener('mouseup', function(e) {
    buttonDown = false;
});
canvas.addEventListener('mousemove', function(e) {
    if (!buttonDown) {
        return;
    }
    ctx.moveTo(prevX, prevY);
    ctx.lineTo(e.x, e.y);
    ctx.stroke();
    prevX = e.x;
    prevY = e.y;
});
</script>

```

Note that there are now three events, but two of them are trivial and the third is familiar. The trivial events are “mousedown” and “mouseup”, which manage the current button state in `buttonDown`. Then we simply change “click” to “mousemove” and only proceed if the button state is what we want it to be. And with that, we have a simple drawing program!

To get the bonus points, you just set `prevX` and `prevY` inside of the “mousedown” handler, like this:

:javascript:

```

canvas.addEventListener("mousedown", function(e) {
    buttonDown = true;
    prevX = e.x;
    prevY = e.y;
});

```

This allows gaps to happen in line drawing, because whenever the button is pressed, it resets the previous point to wherever that button press happened instead of where drawing left off last time.

Exercise 10-4: Output a Simple List of Numbers

Use recursion to output the numbers 1 to 100 in the console, in order. You may use any recursive strategy you like (for example, it is fine to either output then recur, or recur then output, it just depends on how you’re thinking of it).

Note: if you get an error about “maximum stack length” or something similar, just output fewer numbers, like 1 to 50. My browser let me do well over 10,000, however, so it is unlikely that you will run into this.

Answer 10-4

There are two common approaches to recursion like this: output on “descent” into recursion, and output on “ascent” out of recursion. Each is shown here.

On descent:

:javascript:

```
function count(curr) {
  // Terminal case.
  if (curr > 100) {
    return;
  }
  console.log(curr);
  count(curr + 1);
}
count(1);
```

On ascent:

:javascript:

```
function count(max) {
  // Terminal case.
  if (max < 1) {
    return;
  }
  count(max - 1);
  console.log(max);
}
count(100);
```

The “descent” version outputs the current number, then makes the call again with the next number in the series.

The “ascent” version remembers the current number, but waits to output it until all of the rest of the numbers have been output. Thus, it starts at the maximum instead of the minimum.

Exercise 10-5: Create an Array of Numbers

Use recursion to create an array consisting of all numbers from 1 to 100. Remember that if `a` is an array, you can create a new array consisting of `a` together with `b` by calling `a.concat(b)`. That doesn’t change `a`, but creates a new array with the contents of `a`, followed by the contents of `b`. Use `concat` in your recursive code to create the array of numbers.

Any working recursive strategy is allowed, but no loops, and do not edit any array in place: always just return a new one.

Output the final array to the console when complete.

Note: it is fine to not make this efficient. Tail-call-optimizable is not a necessary condition for a correct answer, in case you were worried about that.

Answer 10-5

To build an array using recursion is similar to the console output functions demonstrated above, but now something is returned instead of output to the console. One possible solution is found below.

:javascript:

```
function buildArray(max) {  
  if (max < 1) {  
    return [];  
  }  
  return buildArray(max - 1).concat([max])  
}  
console.log(buildArray(100));
```

That's one approach. There are others. This one is nice and straightforward, though.

As is always the case for recursion, a correct solution will have a base case (in this case the `if` statement that immediately returns) and a recursive case that chips away at the problem one step at a time (here passing `max - 1` into `buildArray`).

It is *possible* to do this while changing the contents of an array, e.g., using `push`, but that is explicitly disallowed for this problem. If the given solution does this, then half credit is appropriate.

Here is an example of a half-credit solution that changes things in place:

:javascript:

```
function buildArray(max) {  
  if (max < 1) {  
    return [];  
  }  
  var arr = buildArray(max - 1);  
  arr.push(max);  
  return arr;  
}
```

Again, this works, but is only good for half credit, since mutating things in place is explicitly not part of this problem.

Exercise 10-6: Output Integers

Use a `while` loop to output the integers from 0 through 99 to `console.log`.

Answer 10-6

Here is one solution:

:javascript:

```
var n = 0;
while (n < 100) {
  console.log(n);
  n++;
}
```

This sets up a variable to hold the current number to be logged, `n`. It starts out holding the value 0, since that is the first value we will be logging.

Then it enters a loop. While the number is less than 100, it executes the body, which logs the current number, then increments it by 1. The following alternative solution shows a couple of things that might be different in your answer, but that are still just fine:

:javascript:

```
var n = 0;
while (n <= 99) {
  console.log(n);
  n = n + 1;
}
```

There are several variations on the theme of “test and increment”, and that’s basically what this `while` loop does: it tests the value to see if it is in range, then outputs and increments it. The loop runs again, testing the newly incremented value to see if it should quit.

There is another variant that can work, that uses `break`. Note also the alternate increment variation:

:javascript:

```
var n = 0;
while (true) {
  console.log(n);
  if (n > 99) {
    break;
  }
  n += 1;
}
```

Any of these, or any combination of these, is fine as an answer.

Exercise 10-7: Do It Again

This time, use a `for` loop to output the numbers from 1 through 99.

Answer 10-7

The `for` loop moves the test, initialization, and increment into one place. It's a lot shorter, but is basically equivalent to the `while` loop above. Here is a canonical approach to this problem using a `for` loop:

:javascript:

```
for (var i=0; i<100; i++) {  
    console.log(i);  
}
```

The increment variable is usually called `i` in these loops, though that is not required. It is extremely common, however, and generally considered to be standard practice. The use of the postincrement operator `++` is also standard in these cases, though `i = i + 1` and `i += 1` are also valid increments in that position. Still, the postincrement is standard. Finally, testing for `< 100` is more common than the equivalent `<= 99` in this case, because the value `100` tells you *how many times the loop body runs*. That's why it's standard practice.

Any of the variations mentioned are fine as answers, however.

Exercise 10-8: Randomness, Key and Click Events, and Anonymous Functions

This exercise is more like a lab. Here we are going to ask you to write a program that creates two event listeners, one for key presses, and one for page clicks. Each of these is going to log a random number to the console, with text indicating which kind of event it was.

The requirements:

- Use `document.body` as the target for your events.
- In the click event, output "click:" and a random number (between 0 and 1).
- When a key is pressed down, output
 - the word "key"
 - the key that was pressed
 - if the key pressed is "Enter", output your name, otherwise
 - output a random number (between 0 and 1).
- Use the `switch` statement to determine whether the key was "Enter".
- Use anonymous functions as the event listeners.

Answer 10-8

There are obviously a few moving parts in this exercise, but it isn't too bad when you take things one at a time. A full solution is below.

:javascript:

```
document.body.addEventListener('click', function(event) {
  console.log('click ' + Math.random());
});

document.body.addEventListener('keydown', function(event) {
  switch (event.key) {
    case 'Enter':
      console.log('my name');
      break;
    default:
      console.log('key ' + Math.random());
      break;
  }
});
```

That's it. There are two event listeners that are registered here, one for "keydown" events, and another for "click" events. Both of them log something to the console. In the case of a click event, it simply logs the word "click" and a random number. In the case of a key event, it logs "my name" if the enter key was pressed, otherwise it logs the word "key" and a random number.

Important parts of this answer include - The use of a `switch` statement in the "keydown" event, - The use of the `break` keyword in each `case`, including the `default` case. - The use of anonymous functions as event listeners.

Partial credit can be given where some of these elements might be missing (for example, if named functions are defined and then passed in, that's worth partial credit).

Exercise 10-9: Animate With Frames

Using a canvas and `requestAnimationFrame`, make an animation. You can use any shape you like. Requirements: - Move at roughly 300 pixels per second, - Start in the upper left corner, - Move in a straight line, and - Stop when it hits an edge.

Note that "a straight line" can be diagonal. Since your shape is starting in the upper left corner, make sure that your straight line motion is to the right, down, or a little bit of both. Though potentially clever, drawing a shape in the upper left and saying "it is traveling left, and it stopped already" in order to avoid writing an animation loop is not really an acceptable answer. Your shape should be seen to be moving.

Note also that you should be writing HTML with JavaScript within it, this time, since you need a canvas element to operate on.

Bonus points: wrap all of your code in an immediate function.

Answer 10-9

Here is one solution. There are places that can be changed (e.g., the shape might be a different color, size, or kind), but the general structure should look something like this (including the bonus immediate function):

:html:

```
<canvas id="drawing" width="400" height="300"></canvas>
<script>(function() {

var canvas = document.getElementById('drawing');
var x = 0,
    y = 0,
    W = 10,
    H = 10;

function draw() {
    var ctx = canvas.getContext('2d');
    ctx.clearRect(0, 0, canvas.width, canvas.height);
    ctx.fillRect(x, y, W, H);
}

var lastTime = 0;
function tick(t) {
    // Quit if out of bounds.
    if (y + H > canvas.height) {
        return;
    }
    if (x + W > canvas.width) {
        return;
    }

    // Set up for the next one
    requestAnimationFrame(tick);

    // Figure out how much time has passed in seconds.
    // Note that there are lots of ways to go about this,
    // and if the student assumes that they're getting roughly
    // 60 frames per second, that's probably fine.
    var dt = (t - lastTime) / 1000;
    if (dt > 2/60) {
        dt = 1/60; // each frame in 1/60th of a second.
    }
    lastTime = t;

    draw();
    x += 300 * dt;
}

// Kick things off.
requestAnimationFrame(tick);
})();</script>
```

The above code will draw a black rectangle in the upper left corner, 10 pixels square. It then moves that rectangle to the right at a rate of about 300 pixels per second, stopping when it hits the right boundary of the canvas.

This is less complex than what we have done in the chapter just before this midterm, so it should hopefully not be too surprising if there was some study before the test.

The basic elements are - Using a `tick` function, - Clearing the canvas before drawing the scene, - Calculating the number of pixels to move based on the desired speed, - Remembering the previous time given in the call to `tick` to compute `dt`.

Chapter 11 Solutions

Exercise 11-1: Library Functions

Add a `clearContext (ctx)` function to “animate.js” that clears the canvas. Note that you can get the underlying canvas object from the context thus: `ctx.canvas`.

Answer 11-1

The function will probably look something like this:

:javascript:

```
function clearContext(ctx) {  
  ctx.clearRect(0, 0, ctx.canvas.width, ctx.canvas.height);  
}
```

It's a simple thing, but it makes clearing a canvas nice because it hides the stuff that doesn't change. That's a theme of abstraction: hide the things that don't change and make the rest into parameters.

Exercise 11-2: Toggle Pause Behavior

Our `animate` library function returns several useful behaviors as functions, stored in an object. Since toggling paused state is something we'll want to do with many animations, move the `togglePaused` function into `animate` and return it with the rest.

Answer 11-2

To do this, we start with the original main program `togglePaused`:

:javascript:

```
function togglePaused() {  
  if (animation.running()) {  
    animation.pause();  
  } else {  
    animation.start();  
  }  
}
```

That should be relatively easy to move into `animate`, but we have a problem: if we just put the function into the returned object, it can't see the other functions inside that object. That's not a big deal, and there are two ways to solve it: move the other function definitions outside of the object, and reimplement the functionality with more primitive operations.

Here's what it looks like if you move the other functions out:

:javascript:

```

function isRunning() {
  return !!frame;
}

function pause() {
  if (isRunning()) {
    cancelAnimationFrame(frame);
    frame = null;
  }
}

function start() {
  if (!isRunning()) {
    frame = requestAnimationFrame(tick);
  }
}

function togglePaused() {
  if (isRunning()) {
    pause();
  } else {
    start();
  }
}

return {
  'running': isRunning,
  'pause': pause,
  'start': start,
  'togglePaused': togglePaused,
};

```

That works fine; the `togglePaused` function can see the other functions now, because they're in scope. We can actually inline `togglePaused` when returning, if we want, as well. That would eliminate a function and replace it with an anonymous one, thus:

:javascript:

```

return {
  'running': isRunning,
  'pause': pause,
  'start': start,
  'togglePaused': function() {
    if (isRunning()) {
      pause();
    } else {
      start();
    }
  },
};

```

Chapter 12 Solutions

Exercise 12-1: Short Circuit Logic Operators

For all expressions below, indicate whether anything is written to the console.

- `var a = false || console.log("hi")`
- `var b = true || console.log("hi")`
- `var c = false && console.log("hi")`
- `var d = true && console.log("hi")`

Note that we are only interested in whether "hi" gets written, not in the actual value of the expressions.

Answer 12-1

- `var a = false || console.log("hi")`: yes - false on the left is not enough to get the value, so the right side is evaluated.
- `var b = true || console.log("hi")`: no - true on the left makes the outcome inevitable, so the right side is skipped.
- `var c = false && console.log("hi")`: no - false on the left side makes the outcome inevitable.
- `var d = true && console.log("hi")`: yes - true on the left side is not enough to get the value, so the right side is evaluated.

Exercise 12-2: Logic Operator Evaluation

For each expression below, predict what it evaluates to:

- `1 || undefined`
- `1 && undefined`
- `0 || 1`
- `undefined || false`
- `true && 6`

Answer 12-2

The rules are these:

- `||`: evaluate to the leftmost truthy value, or the rightmost value if none are found
- `&&`: evaluate to the leftmost falsy value, or the rightmost value if none are found

That's the super general definition that works for Lisp-like languages that can accept any number of operands for OR and AND, but since these JavaScript operators only take two operands, this is a valid, simpler explanation:

- `||`: left if truthy, else right
- `&&`: left if falsy, else right

For the above, then, we have these answers:

- `1 || undefined === 1`
- `1 && undefined === undefined`
- `0 || 1 === 1`
- `undefined || false === false`
- `true && 6 === 6`

Exercise 12-3: Lab Part I: Adding a Score and Increasing Difficulty

For this lab you will add some improvements to the paddle game that we hinted at while creating the first version. These are

- Add a score (at upper right) that increments for each successful hit,
- Make the paddle slightly smaller (multiply its width by `0.99`) with each successful hit,

In other words, we will make this more like an actual game, one that keeps score and gets a little harder and more interesting to play as time goes on.

Note that every context has a `textAlign` property that you can set to “left”, “right”, or “center” and it affects how the `fillText` method decides where to draw your text. For displaying time, we used the default value (because we didn’t specify it at all), which is “left”. Now that we will be changing it, we will need to specify it for time (“left”, since it’s on the left side), and then specify a different one for the score (“right”, since it will be on the right side).

Answer 12-3

Adding The Score

To add a score to our game, we need to do three things:

1. Keep track of the score in a variable,
2. Update the score when we get a hit, and
3. Display the score on the screen.

Let’s tackle those one at a time. First, we need to add some state to our program:

```

:javascript:
var score = 0;

```

That was easy. Next, we need to update it when we get a hit. Remember the code that tests for a paddle hit? We will just increment the score there:

```

:javascript:

```

```

if (ballX >= paddleX && ballX <= paddleX + paddleWidth) {
    score++; // NEW
    ballVelY = -ballVelY;
}

```

Finally, we need to actually display the score. Since we want it on the upper right, we will alter our draw function to also display the score, with a slightly different setup than what we used for displaying time:

:javascript:

```

draw: function() {
    ctx.clearRect(0, 0, canvas.width, canvas.height);
    ctx.font = '20px sans-serif';
    ctx.textAlign = 'left'; // NEW
    ctx.textBaseline = 'hanging';
    ctx.fillStyle = 'black';
    ctx.fillText(Math.floor(animation.elapsed()), 2, 2);
    ctx.textAlign = 'right'; // NEW
    ctx.fillText(score, canvas.width - 2, 2); // NEW

    ctx.fillStyle = "blue";
    ctx.fillRect(paddleX, PADDLE_Y,
        paddleWidth, PADDLE_HEIGHT);

    fillCircle(ctx, ballX, ballY, BALL_RADIUS);
},

```

The changes have “NEW” comments on them. Basically, two `textAlign` settings are added, and a new `fillText` call is added for the score for an overall change of three lines.

Adding Paddle Changes

This change is simpler than the text change because it only affects the `move` function. The width of the paddle is stored in a variable called `paddleWidth`, so all we have to do is make it smaller every time a hit is scored. We just finished changing that code to add `score++` to it; now we need to add `paddleWidth *= 0.99`:

:javascript:

```

if (ballX >= paddleX && ballX <= paddleX + paddleWidth) {
    paddleWidth *= 0.99; // NEW
    ballVelY = -ballVelY;
}

```

This reduces the paddle size by 1% every time a hit is scored. That guarantees that it will get smaller over time, but that it won't completely vanish.

Exercise 12-4: Lab Part II: Adding Randomness and Speed Changes

To continue the lab, we will make the game more difficult by adding some randomness to how

the ball bounces off of the paddle, and by making it go slightly faster every time it does. This requires a bit more vector math, so let's start with some hints about how that should work.

- Make the ball a bit faster with each successful hit, and
- Add randomness to the movement of the ball so it isn't so predictable,

Previously, we just reversed x and y velocity every time a bounce occurred. Among other things, this guaranteed that the overall speed of the ball never changes. Simply reversing one of the components of the velocity doesn't change its speed, only its direction. That is nice, but we're about to fiddle with things on a deeper level now, so we have to be extra careful about maintaining speed.

How do we know that reversing direction doesn't change speed (aside from intuition, I mean)? We know that because speed is the magnitude of the velocity vector, and that's calculated like this:

$$s = \sqrt{v_x^2 + v_y^2}$$

Or, in code, like this:

```
speed = Math.sqrt(ballVelX*ballVelX + ballVelY*ballVelY);
```

Do you see why the speed doesn't change when we change the sign of `ballVelX` or `ballVelY`? It's because they're squared, and that always produces a positive value.

Our goal is to change the direction of the ball in more ways than just "bouncing" it, and we want to slowly increase speed with every bounce. That formula we just used is going to come in handy. Here are some hints for how to go about making this change:

Use `Math.random` to *slightly* alter the `ballVelX` value when bouncing off of the paddle:

We already change `ballVelY` to be its negative when that happens, but now we will *also* change `ballVelX`, just a bit, just to keep things interesting. It should increase *or* decrease the value by a small amount, say up to 10%. How would you accomplish this using `Math.random`?

The first thing to do is to break this idea down into small pieces. How much will we add to (or subtract from) `ballVelX`? It's up to 10% of its current value, let's start with getting a random percentage from 0 to 10: `Math.random() * 0.1`. That will do it.

Now that we can compute a random percentage up to 10%, how do we apply that to `ballVelX` to get a velocity delta (change)? We multiply. The amount we will want to change `ballVelX` by is thus `ballVelX * 0.1 * Math.random()`.

Finally, we want to either add or subtract that value. There are a couple of ways to approach this. You can flip a coin (use `Math.random() > 0.5` for this) to decide whether to add or

subtract, or you can generate a number that is randomly positive or negative and always add. Your choice (there are others, but let's keep it simple).

There are some problems with the idea presented here, and you may notice them as you play for a while. Because the randomness is multiplicative, it's possible (likely, even) to get into a state where the x velocity just keeps getting smaller over time: it's harder to increase its magnitude (because we do so by a percentage) than it is to decrease it: 10% added to a small number doesn't make up for the 10% previously subtracted from a larger one. There are better ways, but this was simple to explain and implement. You may want to try other things.

Keep track of speed, and make sure it only changes *exactly* how you want it to:

Speed can be a tricky thing. If you don't keep a careful eye on it, you could easily end up in a situation where random changes to direction result in a very large increase in overall speed, and that's not intended: we want to *control* speed in this game, not let it go haywire. That means we will need to make absolutely sure that it doesn't get messed up by our introduction of random motion.

Here's the basic idea. Right when a paddle hit is detected, do this:

- Calculate the speed of the ball and remember it (we will call this `originalSpeed`),
- Change direction and add randomness as above,
- Calculate the new speed (we will call this `newSpeed`),
- Scale `ballVelX` and `ballVelY` to make the new speed the same as the original speed.

Most of this is pretty straightforward using the speed formula. If you can calculate speed using the formula given above, then you can do most of the steps in this process. The one that might be a little new is figuring out how to scale things once you have `originalSpeed` and `newSpeed`.

The answer can be found with a little algebra, which you are encouraged to do. This, however, is not an algebra course, so here's the answer: scale the velocity components by `originalSpeed / newSpeed` after changing them. Intuitively, if the new speed is smaller than the original, this makes it bigger by the right amount, and if it is bigger, it makes it smaller. What you end up with is the same speed you started with, but a different direction. You are "renormalizing" the speed.

You don't want it to just stay the same, though. You want it to get slightly faster in a controlled way. Well, that should be pretty simple since we have the original speed. Instead of scaling everything by `originalSpeed / newSpeed`, we will scale it by that times some factor that makes it a bit quicker. For example, you could imagine multiplying that whole thing by `1.01` to add 1% to the speed every time, making the new scale factor `1.01 * originalSpeed / newSpeed`.

Answer 12-4

To make all of this happen, the explanation is a great deal longer than the actual code. First, let's get the random direction handled. We do that by adding or subtracting up to 10% of the current x velocity. That's the same as calculating a random value up to 20% and shifting it so it's plus or minus 10%, like this:

:javascript:

```
var p = 0.2 * (Math.random() - 0.5);
```

See how we compute a random value, then shift it by 0.5? That produces a value between -0.5 and +0.5. Then we multiply that by 0.2, which gives us a number between -0.1 and +0.1, which is exactly what we want: something between -10% and +10%. You can probably see how a little algebra would give us this equivalent expression:

:javascript:

```
var p = 0.2 * Math.random() - 0.1;
```

With this percentage change, it's pretty easy to calculate what to do next. The actual change will just be this multiplied by the current value:

:javascript:

```
var dvx = ballVelX * p;
```

Finally, we add that to the current x velocity to get the new velocity:

:javascript:

```
ballVelX = ballVelX + dvx;
```

Putting it all together and removing temporary variables, this gives us the following collapsed expression:

:javascript:

```
ballVelX = ballVelX + ballVelX * 0.2 * (Math.random() - 0.5);
```

There is one more simplification that can be made here, and it only really becomes obvious when not using += (which is why I didn't, though it's an obvious thing to do): we can factor out ballVelX on the right side:

:javascript:

```
ballVelX = ballVelX * (1 + 0.2 * (Math.random() - 0.5));
```

And now instead of using +=, we can use *=:

:javascript:

```
ballVelX *= 1 + 0.2 * (Math.random() - 0.5);
```

That's kind of fun and clever, but that's not the reason for all of this exposition. The real reason is that any of the above will work: pulling things into multiple variables, various

algebraic representations of percentages, using `+=`, `*=` or neither, etc. The only thing that matters is the right basic calculation.

Now to update the speed: we both want to keep it normalized and control how it gets faster. There was a pretty big hint given in the lab description, and we will make use of that here. The success-detection code for a paddle hit will look like this:

:javascript:

```
if (ballX >= paddleX && ballX <= paddleX + paddleWidth) {
  score++;
  paddleWidth *= 0.99;
  var originalSpeed = Math.sqrt(ballVelX*ballVelX +
                                ballVelY*ballVelY); // NEW

  ballVelY = -ballVelY;
  ballVelX *= 1 + 0.2 * (Math.random() - 0.5); // NEW
  var newSpeed = Math.sqrt(ballVelX*ballVelX +
                           ballVelY*ballVelY); // NEW
  var scale = 1.01 * originalSpeed / newSpeed; // NEW
  ballVelX *= scale; // NEW
  ballVelY *= scale; // NEW
}
```

We added a few more lines here, but they should all be pretty easy to grasp at this point. Let's walk through the changes:

- Calculate `originalSpeed` using the appropriate euclidean distance formula (or the Pythagorean Theorem, if you prefer),
- Add a random x velocity change as described previously,
- Calculate `newSpeed` using the same formula again (this could be pulled into a function),
- Calculate a scale factor for our velocity components that restores the original speed, then increases it by 1%, and finally
- Scale each of the x and y velocities by that factor.

This does what we want. It randomly changes ball direction while maintaining speed as a constant, and it slightly increases that speed in a controlled way.

The game should be a lot more interesting to play, now, with only a handful of lines changed.

We can adjust how quickly things get difficult by changing one of the three parameters:

- The `paddleWidth` multiplier (currently 0.99),
- The randomness in `ballVelX` (currently at 0.2), and
- The speed increase in the `scale` calculation (currently 1.01).

Chapter 13 Solutions

Exercise 13-1: Lab: Improving the Snake Game

For this lab, several improvements will be made. Some of them will come from things we did in the ball and paddle game, and some of them will be new. Here they are:

- Add a score to the game (the length of the snake minus 1).
- Increase the speed of the snake every time it eats.
- Fix it so the snake can never go backward, not even if you go sideways and backward really fast.

The first two should be pretty easy to make work, given that we've done things like them in the previous chapter. The third one might require some more thought. Some possibilities for fixing it might include

- Keep track of all direction changes and process them in order, or
- Keep track of the current direction separate from the requested direction, and only change the current direction when moving, if it's allowed.
- Simply don't let the snake head hit the very next segment. It would not be possible in normal play anyway.

The last of these is easiest. Feel free to toy around with the others, but a valid solution can also be an easy solution.

Answer 13-1

Adding a Score

To add a score, you can put it anywhere that makes sense. You will just draw `segments.length - 1` on the board wherever you choose, maybe something like this (in the `draw` function):

:javascript:

```
ctx.font = '1px sans-serif';
ctx.textAlign = 'left';
ctx.textBaseline = 'hanging';
ctx.fillStyle = 'black';
ctx.fillText(segments.length - 1, 0, 0);
```

That is pretty standard stuff, and we have seen it all before in the previous game. You might find it odd that we are using a 1-pixel font size and shoving things all the way into the upper-left corner instead of having a slight margin around the text. It turns out that after a scaling transform, the canvas treats “1 pixel” as “1 scaled unit”. Thus, in our new regime, 1 pixel is interpreted as “1 grid cell”, and that is a reasonable size for the text. You can use non-integer numbers, too, like “1.5px”, which might be a bit easier to read.

Increasing Speed

To increase speed, you need to decrease `MOVE_EVERY`. That means you should probably rename it to be less like a constant, perhaps something like `moveEvery`. Then, when the snake eats, you can change it to make it smaller. A fairly fun and challenging way to do that is to take 10% away every time, like this:

:javascript:

```
moveEvery *= 0.9;
```

That works pretty well. One of the reasons it works well is that speed has an inverse relationship to delay, so multiplication on the delay gives a nice linear feel to the speed.

Fixing the Bug

The bug we refer to is this: if the snake is moving right, and you quickly tap up then left, the snake will crash backwards into itself. But backwards should simply be disallowed, not game-ending, so we need to somehow ignore that move.

There are lots of interesting ways to fix this in general, but the easiest solution is absolutely fine. It involves changing the key handler to ignore any requests to change direction toward the segment right behind the head.

This is actually more accurate anyway, because if all we have is the head, we should be able to move it any which way, including backwards. This takes care of that as well! It's amazing what we can accomplish when we express what we actually want and not merely an approximation of it. Let's look at the keyboard handler again and see what we can do with it.

The existing version is this:

:javascript:

```
document.addEventListener('keydown', function(event) {
  var keyDirs = {
    'ArrowLeft': direction === 'R' ? 'R' : 'L',
    'ArrowRight': direction === 'L' ? 'L' : 'R',
    'ArrowUp': direction === 'D' ? 'D' : 'U',
    'ArrowDown': direction === 'U' ? 'U' : 'D',
  };

  var dir = keyDirs[event.key];
  if (dir) {
    direction = dir;
  }
});
```

Our existing code is actually kind of tortured. There is a lot of repetition in there and it feels like it is just a little too cute. That is a sign that we might want to rethink things. Let's make this more straightforward by assuming we have a `nextPosition` function that accepts a direction and gives us the next position (we will talk about how to write that function next):

:javascript:

```
document.addEventListener('keydown', function(event) {
  var keyDirs = {
    'ArrowLeft': 'L',
    'ArrowRight': 'R',
    'ArrowUp': 'U',
    'ArrowDown': 'D',
  };

  var dir = keyDirs[event.key];
  // If we got an unexpected key, don't do anything.
  if (!dir) {
    return;
  }
  // Get the next position, ignore if
  // it is segment 1's position:
  var pos = nextPosition(segments[0], dir);
  if (segments.length > 1 && sameCell(pos, segments[1])) {
    return;
  }
  // Finally, we have a good direction,
  // so set the main value:
  direction = dir;
});
```

There's that `sameCell` function again—it just keeps on giving!

Now, what did we just do? We first assumed that there is such a thing as a `nextPosition` function, which there currently is not. But let's assume we can make one (we definitely can).

We made our key object simpler: it just maps keyboard keys to game directions, which is much nicer to think about and work with. As before, we get the direction from the key map, and if we don't recognize the key pressed, we bail out immediately without doing anything.

Then we get the new position that would result from moving in this direction, and if that lands us on segment 1 (the one right behind head), then we're done. We have to test that segment 1 even exists first, which we do by making sure that the `segments` length is greater than 1.

Finally, if none of those exceptional things happens, we allow the direction to change. Yay!

How do we create `nextPosition`, though? It turns out that we already wrote the main code for it, we just need to move it into its own function and call it from the right places. Remember this code inside of the `move` function?

:javascript:

```
// Create a new head, same as the old head.
var newHead = {
  x: segments[0].x,
  y: segments[0].y,
```

```
};

// Then change it based on current direction.
switch (direction) {
case 'U': --newHead.y; break;
case 'D': ++newHead.y; break;
case 'L': --newHead.x; break;
case 'R': ++newHead.x; break;
}
```

We can just lift that all out and put it into a function called `nextPosition` that accepts a `direction` parameter, and then we can replace it with this:

:javascript:

```
var newHead = nextPosition(segments[0], direction);
```

Here is the new `nextPosition` function:

:javascript:

```
function nextPosition(pos, dir) {
  // Create a new head, same as the old head.
  var next = {
    x: pos.x,
    y: pos.y,
  };

  // Then change it based on current direction.
  switch (dir) {
case 'U': --next.y; break;
case 'D': ++next.y; break;
case 'L': --next.x; break;
case 'R': ++next.x; break;
}

  return next;
}
```

Note that we changed some variable names to make it extra clear that these are local variables only valid inside of this new function.

And that's it. It might look like a lot, but really we just did the following:

- Moved the position calculation code into a function, leaving it almost exactly as it was, then
- Using that new function in the key handler to make sure we don't step on segment 1, and removing the ugly parts of that function.

If you repeated logic for this, that is also fine. Extracting things into a function is not strictly necessary, it's just nice to not have to write the same code (and different bugs, usually) twice.

Chapter 14 Solutions

Exercise 14-1: Lab: Create a Segment Class For the Snake Game

We just finished making our snake game work really well, and now it's time to change the implementation without hurting game play. For this lab, you will create a new `Segment` class that accepts x and y coordinates and keeps track of them. Each instance of this class will have three methods:

- `move(dir)`: moves the segment in the direction specified,
- `copy()`: returns a copy of the segment, and
- `collides(other)`: returns `true` if this instance is at the same position as `other`.

Once done, make all segments into instances of this class and change your code to use these new methods where possible.

Answer 14-1

To do this, the first step is to try to create the class. That means we write a function called `Segment`, and then we set prototype functions on it.

Constructor

The `Segment` function is the constructor, and it is supposed to accept x and y coordinates, so it should probably look something like this:

:javascript:

```
function Segment(x, y) {  
  this.x = x;  
  this.y = y;  
}
```

That's pretty simple. All segments have coordinates and don't really need to keep track of much else.

Move

To implement the `move` function, we can pull the move code out of the `move` function (or `nextPosition` if you did the previous chapter's exercises). Remembering that methods are assigned to keys on the constructor's prototype, we get something like this:

:javascript:

```
Segment.prototype.move = function(dir) {  
  switch (dir) {  
    case 'U': this.y--; break;  
    case 'D': this.y++; break;
```



```

    case 'L': this.x--; break;
    case 'R': this.x++; break;
  }
};

```

Since we are not creating a copy in this function, we can just change things directly instead, so that part of the code is somewhat simplified.

Copy

To copy the instance, we just create a new one at the same coordinates:

:javascript:

```

Segment.prototype.copy = function() {
  return new Segment(this.x, this.y);
};

```

Collides

Testing for collision is also relatively straightforward:

:javascript:

```

Segments.prototype.collides = function(other) {
  return other.x === this.x && other.y === this.y;
};

```

Using The Methods

There is at least one place where these methods will be used. First, instead of creating the segment objects directly using {}, we will use `new Segment(x, y)` instead. That transforms this code

:javascript:

```

var segments = [
  {x: Math.floor(CELLS_PER_DIM / 2),
   y: Math.floor(CELLS_PER_DIM / 2)},
];

```

into this code

:javascript:

```

var segments = [
  new Segment(Math.floor(CELLS_PER_DIM / 2),
               Math.floor(CELLS_PER_DIM / 2)),
];

```

It is not really any clearer, but we will see benefits as we use it elsewhere. For example, in `move`, we can get the new head like this:

:javascript:

```
var newHead = segments[0].copy().move(direction);
```

That is nice - we create a copy and then move it, and that is the new head.

Finally, to test for collision with body segments, we did this before:

:javascript:

```
var head = segments[0];

for (var i = 1; i < segments.length; i++) {
  var pos = segments[i];
  if (head.x === pos.x && head.y === pos.y) {
    alert("Game over: self crash");
    return true;
  }
}
```

Now we do this:

:javascript:

```
for (var i = 1; i < segments.length; i++) {
  if (segments[0].collides(segments[i])) {
    alert("Game Over: self crash");
    return true;
  }
}
```

That is a little nicer.

Finally, if you did the exercises earlier, you will have changed the way that backward motion is calculated to use a `nextPosition` function. That has been replaced with the copy-plus-move code shown above, and `nextPosition` is no longer needed.

Chapter 15 Solutions

Exercise 15-1: Lab: More Interesting Counters

For this final lab, you will add 2 or 3 more counters to the page and allow them to be named using input fields. All of them should work simultaneously.

For this, a few little details might trip you up.

First of all, if you are in a loop and create a closure based on variables in that loop, you are going to have a bad time. Check this out:

```
for (var i = 0; i < 3; i++) {  
  setInterval(function() {  
    console.log("I'm the " + i + " value");  
  }, 1000);  
}
```

What is this attempting to do? It's trying to set up three intervals, each of which outputs something different to the console. If you run it, you will notice that every single one of them says "I'm the 3 value". Oops, that's not what we wanted!

The problem is this: our little function closes over the `i` variable, but that variable just changes value each time through the loop. It is not a new variable. So, by the time our functions are run (one second later), that value is 3 for all of them. They all close over the same variable.

Ugh. How do we fix that? Well, you can use `let` instead of `var` if you are in a browser (or other environment) that supports it. That has much saner behavior in these circumstances and creates new, scoped variables in a more natural way. If you don't have that option, you can create a function that does this registration for you, effectively giving your closure new variables to close over:

```
function register(i) {  
  setInterval(function() {  
    console.log("I'm the " + i + " value");  
  }, 1000);  
}  
  
for (var i = 0; i < 3; i++) {  
  register(i);  
}
```

That works. The reason is that the `register` function is called immediately each time through the loop, and each function call gets its own variables to play with. Thus, our completely unchanged closure now works properly because it is closing over new variables each time.

Why do you need this? You might not. There are plenty of ways to do this lab without knowing

this rather subtle detail, but at least one of the ways, the one that I'll show in the answer key, needs to handle this case properly.

Answer 15-1

To make this change, let's just add 2 new counters, and some names. We started out with these tags in the body:

:html:

```
<input type="text" size="5" maxlength="5" id="date"> :
<span id="counter"></span> :
```

Now we will add more, give them each editable names, number them, and add line breaks between them:

:html:

```
<input type="text" id="name0">:
<input type="text" id="date0" size="5" maxlength="5"> :
<span id="counter0"></span><br>

<input type="text" id="name1">:
<input type="text" id="date1" size="5" maxlength="5"> :
<span id="counter1"></span><br>

<input type="text" id="name2">:
<input type="text" id="date2" size="5" maxlength="5"> :
<span id="counter2"></span><br>
```

The spacing is not terribly important; most spaces are collapsed into a single space in HTML anyway (even newlines, which is why the `
` tag is needed). The important thing is that now there are three of these, and they are all given different but predictable names. Now we can start changing the way we work with them. Previously we had this:

:javascript:

```
function main() {
    var inputTarget = document.getElementById('date'),
        elCounter = document.getElementById('counter');

    // Set a default value for it (next year):
    inputTarget.value = "01-01";

    // Every second, show the counter.
    setInterval(function() {
        showCounter(inputTarget, elCounter);
    }, 1000);
}
```

But now we will want to do these things for all of them (and give each of them a default name). Since we will be looping over them and using a closure in that loop, we will need to employ the

function trick shown in the requirements as a hint. Or we can use `let`. Let's use the function closure trick to create a new closure with the desired variables on the fly:

:javascript:

```
function makeCounter(elTarget, elCounter) {
  return function() {
    showCounter(elTarget, elCounter);
  };
}

function main() {
  // Every second, show the counters.
  for (var i = 0; i < 3; i++) {
    var elTarget = document.getElementById('date' + i),
        elCounter = document.getElementById('counter' + i),
        elName = document.getElementById('name' + i);
    // Set a default value for it (next year):
    elTarget.value = "01-01";
    elName.value = "event " + i;

    setInterval(makeCounter(elTarget, elCounter), 1000);
  }
}
```

We basically put the entirety of `main` into a loop, taking care to ensure that closures that depend on loop variables (or any variable inside the loop, for that matter) are wrapped inside of a function that creates them. We could also have registered them with `setInterval` directly in that function if we wanted. There are many approaches. The key is to isolate the closure in a place where it has its own unique variables.

With these changes, you can have three different fun dates that you are counting down to, and you can give them names. They will keep counting until you reload the page.