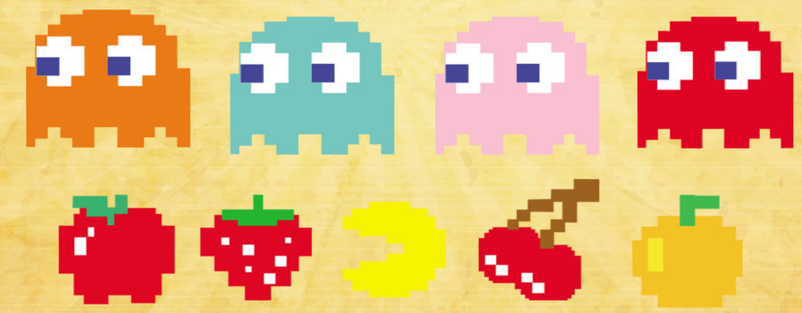
BUILD PACMAN

LEARN MODERN JAVASCRIPT, HTML5 CANVAS, AND A BIT OF EMBERJS



JEFFREY BILES

BUILD PACMAN

Learn Modern Javascript, HTML5 Canvas, and a bit of EmberJS

Jeffrey Biles

This book is for sale at http://leanpub.com/buildpacman

This version was published on 2021-01-28

Contents

In Media Res

[Content Warnings: Religion, Violence, Geopolitics, Creative Punctuation. Feel free to read just the instructional chapters if these things are gonna piss you off (or if you're at work)]

You're running through the jungle.

The why seems distant now, lost in the many dark mazes you've made. All you feel now is the whip of branches in your face, the crunch of twigs underfoot, and the chop chop chop of an approaching helicopter.

The helicopter suddenly comes into view. It's close. And there's a guy sitting there with a machine gun. "I've finally got you!" he yells. He starts firing, and trees splinter and fall under the spray of bullets.

One of those trees falls on you. Trapped. You're trapped. The helicopter is coming closer. He wants to see your face as you die.

But then a rocket bursts onto the helicopter, and it's in flames. Your pursuer leaps away from what is soon to be flaming rubble. He lands close to you, so close that you can hear the sickening crunch of his bones snapping. His power pellet had run out.

He still turns towards you, using his arms to drag his shattered body closer. His face strains with effort, pain, hate, but in it you see something familiar.

He collapses, out of breath, and it becomes clearer.

My god, it's him.

It's him.

He looks at you and sees the same thing. The rage drains from his face. Now… now it's just pain.

Just memory.

"I'll never forget," he says. "I'll never forget the caves."

And suddenly you're back there with him. Those dark caves, those blackened rooms, with only the pellets and the chase.

The constant fear.

They're abstractions now, but back then they were very real.

"I don't know why you turned," he says. "I guess I'll never know."

You start to explain, start to make excuses, but the words don't come. An untrained man would be crying, but you are not untrained. Instead, you only say: "You will be blessed in heaven for what you have done."

“Is it too late for you?” he asks. “Can I see you there?”

A bullet and he’s gone. The tree lifts off of you, and a hand extends, pulling you up.

You should feel relieved.

You’re safe.

Your fellow Ghosts have arrived.

1: Drawing A circle

This chapter is much like the previous one- weird, filled with characters you don't understand, and eventually coming back around to pac-man. The only difference is that in this chapter, we'll be doing lots of code.

It's a version of In Media Res- a storytelling technique used to get people into the action quickly. You open in the middle of the story, during a really exciting scene, and then you later go back and build up the characters and relationships that made that scene possible.

We're going to use a similar technique for this chapter- we're going to jump right into installing a bunch of tools and using some concepts you may not understand just yet. The upshot is that it's going to let us start drawing circles (little prototype pac-men) on our canvas really quickly. From there we can take it slow.

> A note about tools:
>
> We are going to be using lots of tools in this book- Javascript, Babel.js (a tool that lets us use the latest versions of Javascript before they're in all the browsers), HTML5 Canvas, Ember.js, ember-cli, and more. Originally the book was going to be a book for pros focusing on just ES2015 and HTML5 Canvas, but I discovered that using more of these tools actually makes our job easier.
>
> The goal of this book isn't to be a conclusive guide in any of these technologies- it's to give you enough knowledge in the tool to get started, and then use that tool to accomplish our goal (build a bad-ass game of pac-man). However, at the end of the book you'll be familiar with several really useful concepts- and if you decide to continue learning, I'll provide several resources for each.

Installation

The first step is to install Node. Go to https://nodejs.org/ and hit the big green "install" button. Do whatever your operating system makes you do to install things, and then you have node and npm on your system. Go to your command line, and then type in `node -v`. Then type in `npm -v`. If the commands worked and they gave you numbers, hurray! If not, please google for help. This isn't a book about node and npm… they're just tools we need to get started.

Next, we're going to install ember-cli. Type `npm install -g ember-cli` into the command line. And… that's it. Type `ember -v` into your command line, and it should give you the ember-cli version (>= 2.3.0) and the node and npm versions.

Creating your app

```
ember new pac-man
```

That command will create for you an ember app called pac-man. It's going to spend a short amount of time creating a directory structure and some config files for you, and then a bit longer installing npm and bower packages.

```
cd pac-man
```

```
ember server
```

These commands will move you into your app's folder and then start the server. Go to localhost:4200 and you should see "Welcome to Ember.js". If you do, then congratulations! You've created your first ember app.

Displaying a box

```
ember generate component pac-man
```

Type that into the command line and you will get a new ember component called pac-man. ember-cli will generate the javascript file, template file, and test for a component.

In app/templates/application.hbs, delete everything, replace with the component `{{pac-man}}`.

This tells ember to display the component. From now on, everything will happen inside the component.

```
<!-- in app/templates/components/pac-man.hbs -->
<canvas id="myCanvas" width="800" height="600"></canvas>
```

This creates an html5 canvas that we can draw on.

```
/*in app/styles/app.css*/
#myCanvas {
  background-color: #EDC9AF;
}
```

This colors the canvas so we can see it. It's a desert-sand light brown.

If everything has gone correctly, at localhost:4200 you should see a rectangle the color of desert sand.

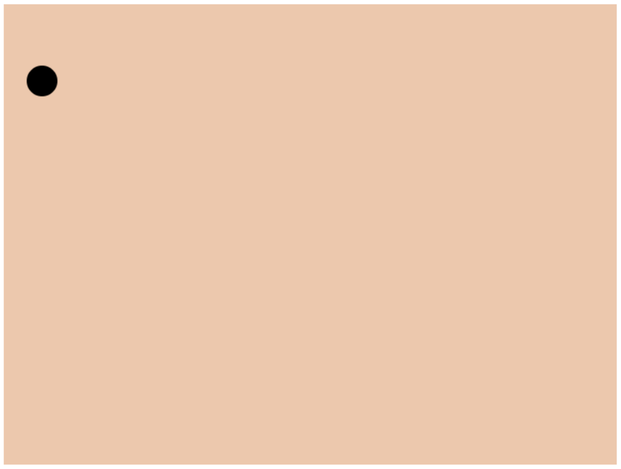
Drawing a circle

```
// in app/components/pac-man.js
import Ember from 'ember';

export default Ember.Component.extend({
  didInsertElement: function() {
    this.drawCircle();
  },

  drawCircle: function() {
    let canvas = document.getElementById("myCanvas");
    let ctx = canvas.getContext("2d");
    let x = 50;
    let y = 100;
    let radius = 20;

    ctx.fillStyle = '#000';
    ctx.beginPath();
    ctx.arc(x, y, radius, 0, Math.PI * 2, false);
    ctx.closePath();
    ctx.fill();
  },
});
```

`didInsertElement` runs whenever the component (the element) is loaded and put on the screen (more on this in the next chapter). In this case, we're just calling the drawCircle function, which is defined right below.

> Note for advanced Ember devs (everyone else ignore): using `didInsertElement` in this case will give a deprecation warning, but for now it's the best option. Using `init` will give an error because we need the html to have already been rendered. Using `didRender` will plunge you into an infinite loop.

In the drawCircle function, we're grabbing the canvas, telling it that we'll be drawing in 2d, and then doing the incantations that tell canvas to draw a circle. You won't have to worry about these specific incantations- this book will almost always hide shape-drawing code behind a function that you can just copy and paste without losing much.



Setup complete

Thus concludes our in-media-res introduction. It's expected that you won't understand a lot of what just happened- it's just meant to drop you into the world as quickly as possible. Some of the concepts introduced here will be given a lot more screen time later. Some of them were just necessary for setup and will never be brought up again.

Now let the training begin.

> Having trouble? Send me a question via email at jeffrey@emberscreencasts.com. I'll send you a response, and create an FAQ with answers to the most common questions

> You can also check out the github repositories for each chapter's code[1] or the finished addon[2].

[1]https://github.com/jeffreybiles/chapter-by-chapter-game
[2]https://github.com/jeffreybiles/pac-man

Training

The desert stretches farther than you can see. Only the occasional dust eddy disrupts the monotony.

“So this is the world outside.”

Elder Matteo nods.

“Is it all like this?”

“Most of it. Still patches we haven’t gotten to yet.”

“So it was different before…”

“When I was a boy, we had lush jungles, quiet lakes… food… we didn’t need to mine pellets back then.”

“How long has it been since you’ve seen this… food?”

A light goes on in the elder’s eyes. “I saw a cherry last year. When visiting the neighboring bubble. It was beautiful…. but enough of an old man’s reminisces. If this bubble is to be fed, we need more PAC operators. Are you ready?”

You turn your imagination away from what a cherry might be, and look at your PAC. Your Pellet Acquisition Capsule. You’ve been processing the pellets for years, but seeing a PAC up close for the first time is still an awe-inspiring experience. “Ready as I’ll ever be.”

You get in to the yellow sphere and let the wires envelop you. Matteo closes the hatch.

“Let’s get you acquainted with this contraption, alright?”

2: Getting up to speed

The last chapter threw you headlong into a large number of new technologies. I'm glad you made it out alright!

This is the part of the book where we introduce a bunch of characters, but we don't know much about them yet. Don't worry, you will eventually! Remember what you can, and come back to this chapter if you need to.

In this chapter we're going to zero in on the pac-man component, going over different parts of the code until it's (mostly) no longer mysterious.

Components

In `app/compents/pac-man.js`, you'll see the following code:

```
Ember.Component.extend({
  ...
})
```

This means that PacMan is an Ember Component- it inherits all the properties of an Ember Component, while adding its own. An Ember Component is basically a container that is sealed from the outside world, except for if you poke a few holes into it. Most real-world programs will need to poke a couple holes, but we're not poking any holes right now, so all we need to care about is in `app/components/pac-man.js` and `app/templates/components/pac-man.hbs` (and `style/app.css`).

As someone writing an intro book, this is incredibly convenient. You may have heard about the Ember Router and how revolutionary and vital it is… well, we're going to write an entire game in Ember and never touch it. Same with Initializers, Controllers, Services, and innumerable other Ember features that are necessary in many apps but are also confusing to newcomers. We get to take advantage of the parts that are useful and leave the more confusing things for the second book.

> If you want to learn more about components, here is a series of videos about them[3].

Of course, components don't always consist of just a javascript file- they can have an associated handlebars (they can also consist of just a handlebars file, although that doesn't happen in this tutorial).

[3]https://www.emberscreencasts.com/tags/components

Handlebars

A handlebars file is kind of like an html file, but you can do some limited programming in it. You can input variables, use if statements, and more (but not much more). Each template is associated with an Ember class, and that is where it gets the variables it uses.

Later we'll use more of these features, but for right now all we're going to do is create the html5 Canvas (using just plain old html).

```
<!-- in app/templates/components/pac-man.hbs -->
<canvas id="myCanvas" width="800" height="600"></canvas>
```

Canvas

Canvas is well named- it's like a painter's canvas. You can paint on it... then you can paint on it again, partially painting over what you had previously drawn. And then, just like a real painter's canvas[4], you can create a bunch of robots that splatter paint over the canvas in precise intervals in order to create the illusion of motion.

This canvas defaults to pure white, but we've decided to tan it up a little using css.

Css

`app/styles/app.css` is where we're going to store the styling information. While css is a powerful tool, in this book we're basically just going to use it to turn things pretty colors, so there's no previous knowledge needed (and, likely, no knowledge to be gained). You can just copy and paste any css we present directly into `app/styles/app.css`.

Here's what you copy-pasted in the last chapter:

```
#myCanvas {
  background-color: #EDC9AF;
}
```

It turns the canvas a nice tan color.

> CSS means 'Cascading Style Sheets', and is way more important in a regular web app than it is in pac-man. If you're interested in learning more, click here for some great introductory courses[5] (subscription required).

[4] If you're fabulously wealthy, a brilliant machinist, and don't care for the laws of physics.
[5] https://www.codeschool.com/paths/html-css

Canvas context and drawing

Of course, we don't just want that tan color… we want a circle!

That's where the following code comes into play

```
let canvas = document.getElementById("myCanvas");
let ctx = canvas.getContext("2d");

...

ctx.fillStyle = '#000';
ctx.beginPath();
ctx.arc(x, y, radius, 0, Math.PI * 2, false);
ctx.closePath();
ctx.fill();
```

The first line grabs the canvas, then the second line gets the 2d context, which we'll call `ctx` for short.

The last five lines are drawing the circle onto the 2d context of the canvas. You don't need to understand them. But you do need to know what `let` is.

Let

`let` is extremely well named. Take this example:

```
let chapterNumber = 2;
```

Read it out loud: 'Let chapterNumber equal two'. That's pretty much what it does. `chapterNumber` is now a variable equal to two. Another way to think of this is "assigning values to variables", where `2` is the value and `chapterNumber` is the variable.

If you've programmed in javascript before, you'll know about `var`. `let` is similar, but has slightly different scoping properties, which are generally less confusing. If you haven't programmed in javascript before, you can safely ignore that last sentence.

So right before we draw our circle we have a bunch of let statements, and then we use those variables in the drawing.

```
let x = 50;
let y = 100;
let radius = 20;
ctx.arc(x, y, radius, 0, Math.PI * 2, false);
```

is equivalent to

```
ctx.arc(50, 100, 20, 0, Math.PI * 2, false);
```

We name the variables because it's easier to read. We may also want to extract them later (say, next chapter).

> `let` is a feature of ES2015 which isn't supported in all browsers… why are we using it? Because we're using Babel, an ES2015+ transpiler. That means Babel takes your code that's written to the ES2015 (or ES2016, ES2017, etc.) and turns it into code that (almost) all browsers can read (sorry, IE7 users).
>
> "But wait," you ask, "how do I install Babel?"
>
> It's already installed. You started using it the moment you generated the project. This is (part of the) magic of ember-cli.

didInsertElement

`didInsertElement` is your way of telling the Ember component "hey, after you're done putting this component on the screen, I want you to do this."

"Putting this component on the screen" usually means "displaying whatever is in the handlebars file". In this case, the HTML5 Canvas.

Here's our full code:

```
didInsertElement: function() {
  this.drawCircle();
},
```

So what it's saying is "whenever you're done putting this component on the screen, draw a circle".

You may be wondering what's up with all the `function()` and the curly braces `{}` mean… well, those are parts of defining and using functions.

Functions

Functions are a great way to reuse code. We define it once, and then we never have to write that code again.

Here is us drawing a circle:

```
this.drawCircle();
```

Here is us drawing two circles:

```
this.drawCircle();
this.drawCircle();
```

It's easy, because we defined the `drawCircle` function. We'll get into the funny marks later.

Here's the basic format of defining a function:

```
functionName: function() {
  ...
}
```

> There's another slightly better way of defining functions that's new in ES2015. We're going to wait to introduce that technique, because introducing it now would make introducing hashes in the next chapter more confusing.

Here is us defining the drawCircle function:

```
drawCircle: function() {
  let canvas = document.getElementById("myCanvas");
  let ctx = canvas.getContext("2d");
  let x = 50;
  let y = 100;
  let radius = 20;

  ctx.fillStyle = '#000';
  ctx.beginPath();
  ctx.arc(x, y, radius, 0, Math.PI * 2, false);
  ctx.closePath();
  ctx.fill();
},
```

Everything in between `{` and `}` gets run when we call drawCircle.

We're defining the drawCircle function on the component. One way to think of it is to say that the component now knows how to draw a circle. Another more fancy way to think of it is 'the function drawCircle is assigned to the component scope'.

> Scope can be a scary word, but here's a basic way to think about it. If you're in your living room, and you say "I would like to sit on the couch", you don't have to specify which of the millions of couches you're sitting on. You're in the living room scope, so when you're trying to think of couches, the one in the living room comes to mind first. It's the same reason that if you talk to someone in the United States about "the civil war", they'll immediately think of the American Civil War, not the Spanish Civil War, the American Revolution, or Marvel Comic's Civil War (tm). That's because they're scoped to the United States[6].

Let's go back to where we were drawing circles and talk about those funny marks:

```
this.drawCircle();
```

`this`, in this case, means 'the component scope'. Hey, that's where we stored the drawCircle function. How lucky!

`this.drawCircle`, without the `()`, would return the code for the function. When we add the `()` it 'calls' the function, which means it runs the code in the function. And that code draws the circle.

> `didInsertElement` is a special type of function called a 'hook'. Don't worry about it for now, just noticed that we define it using the same patterns that we used to define `drawCircle`

import/export

One last thing is the import and export functionality. This is the ES2015 module system at work!

So what `import` says is "I'm gonna need some code. Give me the code."

```
import Ember from 'ember';
```

This says "Go to the place (library) known as 'ember', take the default thing there, and call it 'Ember'".

Then we have the following line:

[6](Okay, maybe some nerds here are scoped to Marvel Comics. I won't judge.)

```
export default Ember.Component.extend({
  ...
})
```

This says "Whenever someone asks for something from this file (and doesn't specify any particular thing), give them the Component we're defining."

If we wanted to get PacMan in another file, we'd need to say:

```
import PacMan from 'pac-man/components/pac-man';
```

So we go into the pac-man library (our project), go into the components folder, and get the component named pac-man.

There's other ways to use imports and exports, but we won't need them in this book.

> Click here for more about ES2015 modules[7]

Feeling good yet?

We've learned a lot of new concepts in this chapter- Components, Handlebars, Functions, Let, Canvas, CSS, and ES2015 Modules. The rest of the book will build on these concepts (while introducing new ones, albeit at a slower pace).

Now, back to our story.

[7]https://www.emberscreencasts.com/posts/62-es2015-modules-import-export

Control

The first day in the PAC was spent familiarizing yourself with the different screens. You could have done it just fine at a desk, but you suspect that part of the training is getting used to being in the PAC.

The PAC was not as cramped inside as you would have guessed from looking at it, but it was definitely different than the engineering desks you were used to. Your torso and limbs, even your head, were trapped in their own web of wires. You could move them slowly in any direction, within limits, but anything sudden caused them to tauten. Matteo had just said, “You’ll be glad of this once we start with movement. And collisions.”

For now it was just disorienting. Thankfully the screen stayed in front of your face, giving something of a familiarity to the proceedings. You’ve spent most of your working life in front of a screen.

So yesterday had been the screens, and today would be the buttons.

As you approach the training field, you see that Elder Matteo is talking with another person. Getting closer you start to recognize him- Terrance Mayhew. Terrance, the man whose discoveries raised PAC efficiency 14%. One of only five people with significant achievements in both engineering and the PAC corps (although he had started in PAC and shifted to engineering, where he was now much more famous). The most honored of the ‘Ahl Al-Kitab.

The talking ceases when you arrive.

“Salam,” says Matteo, “Are you ready for an experiment?”

You nod, buzzing faintly with nervousness. The soreness in your limbs is forgotten in the excitement.

“Terrance and I have been working together on an enhancement to the training process, one that could decrease the disorientation and death that sometimes accompanies this stage of the training. We thought that you, with your background, would be the perfect test subject.”

You nod again. Decreasing death sounds pretty good. And if it means you get to work on a project with Terrance…

You also begin to understand how big a deal Matteo is in the PAC world. Terrance could work with anyone, and it was Matteo. And you get to work with both of them. Maybe the summons wasn’t such a curse after all.

“Where do I go?” you ask.

Matteo points to the PAC. You step in, standing still for the few seconds it takes for the wires to take hold.

“We’re going to start with how you control the PAC.”

3: Input

In this chapter, we'll be learning how to take keyboard input. We won't be moving yet, but we'll be showing the result in the log.

Steal This Code

Input has historically been difficult in javascript… but luckily, other people have done all the hard work for us! We just need a way to get their hard work into our project. Even more luckily, other people have designed easy ways for us to take their code! It's truly a wonderful world.

> We're not really "taking" the code, since it is freely given. These are more accurately called "code-sharing" tools, but for the duration of this book we won't be sharing… just taking. But I encourage you to start sharing your code as soon as you feel comfortable doing so!

Since you're using ember-cli, you already have access to several of these wonderful code-taking tools- and another tool that ties them together.

To take the particular piece of code we'll need, type `ember install ember-keyboard-shortcuts` into the command line. Now you have the library "ember-keyboard-shortcuts[8]" available to you!

> This will give you the npm package 'ember-keyboard-shortcuts' and the bower package 'mousetrap'. Npm and bower are both vital tools in everyday programming life, but we won't see them again in this book.

Be sure to restart your Ember server to get the keyboard shortcuts to load- stop the server in the command line (Ctl + C on OSX), then restart (`ember s`) after installing the addon.

Bringing it in

You can now access the 'ember-keyboard-shortcuts' codebase from within your project, but there's an intermediate step before you can use any of their code. You need to import it.

At the top of our pac-man.js component, under the Ember import, we'll import the KeyboardShortcuts mixin.

[8]https://github.com/Skalar/ember-keyboard-shortcuts

```
import Ember from 'ember';
import KeyboardShortcuts from 'ember-keyboard-shortcuts/mixins/component';
```

In this code, we're importing just one mixin- specifically, the mixin that is meant to be used in an Ember component.

> If you remember our discussion of modules from chapter 2, you may be able to guess which file in the addon we're importing this from. If you can't, no worries- understanding this isn't core to this book.

What is a mixin? Glad you asked.

Mixins

Remember when we said that `PacMan` would "inherit" a bunch of properties from `Ember.Component`? That basically said "Hi PacMan, I'm Ember.Component. You're going to be just like me, but don't worry- you can change things". Mixins say "Hey PacMan, here's a bundle of new related code you can use if you want."

How do you mix in a Mixin?

```
import KeyboardShortcuts from 'ember-keyboard-shortcuts/mixins/component';
export default Ember.Component.extend(KeyboardShortcuts, {
  ...
})
```

So after you call `extend`, but before the `{`, you put your Mixins (separated by commas). You can mix in as many Mixins as you want.

> ### Inheritance Chain
> Now PacMan has a mix of stuff on it from Ember.Component, KeyboardShortcuts, and the PacMan object itself. What happens if something is defined twice? In the general case, a definition on the object itself takes precedence over a definition on the Mixin, which takes precedence over a definition on the parent class. Applied to this specific case, PacMan beats KeyboardShortcuts, which beats Ember.Component. We've seen this already with `didInsertElement`, which was previously defined on Ember.Component but then overwritten on PacMan.

Defining keyboard shortcuts

After you've mixed in the KeyboardShortcuts mixin, you can put the following code into your PacMan class:

```
keyboardShortcuts: {
  up: function() { console.log('up');},
},
```

The effect of this is that when you hit the ‘up’ key, your browser says ‘up’ (in the console, which I’ll show you how to access soon).

Let’s go over each element of this, starting with hashes.

Hashes

Here’s an example of a hash:

```
{
  firstName: "Jeffrey",
  lastName: "Biles",
  sideBusiness: "EmberScreencasts.com",
  firstTechnicalBook: "BUILD PACMAN"
}
```

The way to read this is: “first name is Jeffrey, last name is Biles, side business is emberscreencasts.com, and first technical book is BUILD PACMAN”.

In fancy technical terms, you could talk about ‘keys’ and ‘values’, like “The firstName key has a value of Jeffrey”, or “‘firstName: “Jeffrey”’ is a key-value pair”. I bring up those terms partially to be fancy, but also so I can give important messages like “a hash is a set of key-value pairs” and “key-value pairs in a hash are separated by commas”.

```
{
  key: value,
  anotherKey: 'anotherValue'
}
```

> Beginners: Read the last paragraph and code sample a couple times if you have to. Advanced folks: email me any easier explanations you’ve found.

The value in a key-value pair could be a function. That’s what we’ll use in our keyboardShortcuts hash.

```
keyboardShortcuts: {
  up: function() { console.log('up');},
  down: function() { console.log('down');},
  left: function() { console.log('left');},
  right: function() { console.log('right');},
},
```

Now that you know what a hash looks like, you may notice one other hash we've been using:

```
export default Ember.Component.extend(KeyboardShortcuts, {
  didInsertElement: function(){...},
  drawCircle: function(){...},
  keyboardShortcuts: {...},
})
```

So we can see that the value in each key-value pair can be a function or another hash. It can also be a simpler property, such as a number or a string.

Now that we understand hashes, let's look at the results of our work.

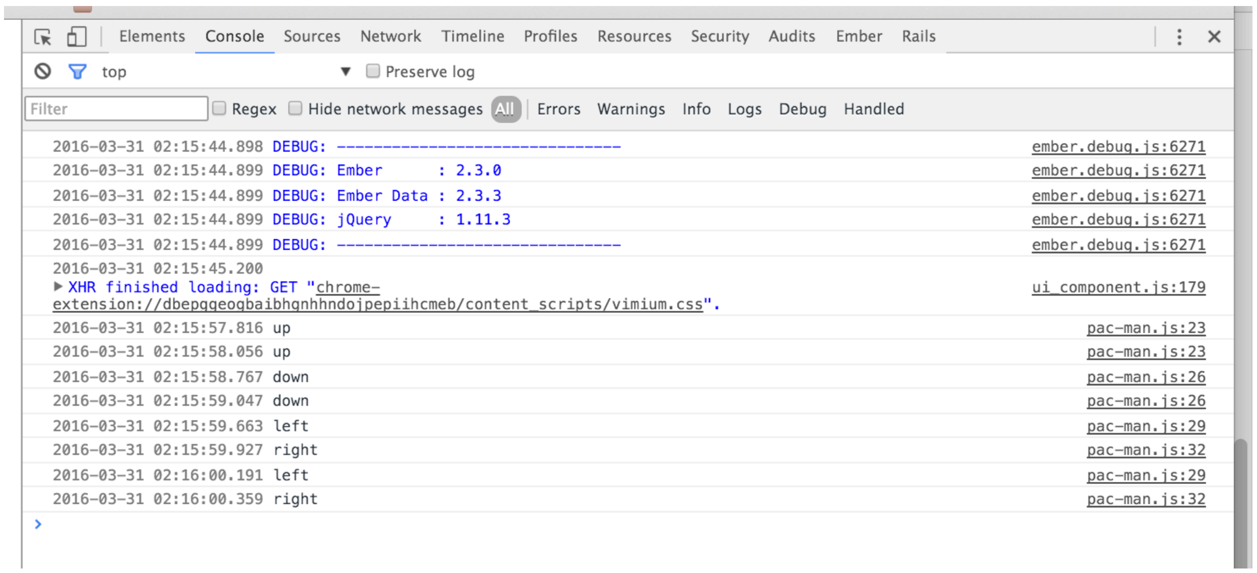
Logging

You'll notice that after each of the directions in keyboardShortcuts we've used a `console.log`. This says, "Hey browser, put this stuff in the console so the programmer can look at it."

But where in the browser does it put it?

1. Open up Chrome.
2. Right click anywhere on the screen.
3. Choose "Inspect Element" from the drop-down menu
4. Either click the 'Console' tab or press the escape key.

Now you're looking at the console. Hurrah! Click back into the game, then hit some direction keys. If you've set up your code like we've described, you'll be seeing the directions printed out.



```
2016-03-31 02:15:44.898 DEBUG: -------------------------------    ember.debug.js:6271
2016-03-31 02:15:44.899 DEBUG: Ember      : 2.3.0                 ember.debug.js:6271
2016-03-31 02:15:44.899 DEBUG: Ember Data : 2.3.3                 ember.debug.js:6271
2016-03-31 02:15:44.899 DEBUG: jQuery     : 1.11.3                ember.debug.js:6271
2016-03-31 02:15:44.899 DEBUG: -------------------------------    ember.debug.js:6271
2016-03-31 02:15:45.200
▶ XHR finished loading: GET "chrome-                              ui_component.js:179
extension://dbepggeogbaibhgnhhndojpepiihcmeb/content_scripts/vimium.css".
2016-03-31 02:15:57.816 up                                             pac-man.js:23
2016-03-31 02:15:58.056 up                                             pac-man.js:23
2016-03-31 02:15:58.767 down                                           pac-man.js:26
2016-03-31 02:15:59.047 down                                           pac-man.js:26
2016-03-31 02:15:59.663 left                                           pac-man.js:29
2016-03-31 02:15:59.927 right                                          pac-man.js:32
2016-03-31 02:16:00.191 left                                           pac-man.js:29
2016-03-31 02:16:00.359 right                                          pac-man.js:32
>
```

You can put almost anything into `console.log`. You can put classes, functions, numbers, strings (what we currently have in there), hashes, and more. You can also put more than one thing in there- just separate them by commas.

Summary

In this chapter, we've given our code some… direction. We learned about how to import external libraries, how to use mixins, how to use hashes, and how to log stuff to the console.

The next chapter will use our button pressing to actually change stuff in the game world.

… And Much Much More

For more, buy the full book[9].

[9] https://leanpub.com/buildpacman