Wizards use Vim

a JavaScript-mancy SideQuest

by Jaime González García

Wizards Use Vim

Jaime González García

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

This version was published on 2020-05-04

Contents

A Note About How I am Writing This Book

Hi dear reader!

I thought it would be interesting for you to learn a little bit about how I am writing this book. If you are indeed interested then don't hesitate to take a look below, otherwise continue directly into the next chapter.

The Process

I write each chapter individually and usually follow these steps:

1. outline
2. write content and code samples in full
3. review content
4. jump to next chapter while waiting for reviewer
5. reviewer reviews the chapter and gives feedback
6. second review
7. add story, exercises, code samples to GitHub
8. final review

I generally follow steps *1* to *3* and jump from one chapter to the next so I can get the most amount of content in your hands as soon as possible. I batch the reviewers feedback and I process it in chunks. The steps *7* and *8* I'll leave for the end when all the content that is going to be part of the book is complete.

Chapter Writing Status

In order to provide you with some guidance as to which chapters are safe to read and which are in progress I put a grade to each chapter in terms of its maturity. For this scale of maturity I am using the following levels that you can see below together with a brief description of what they entail:

Chapter in Stage 0 - Strawman

This chapter is still in progress. You are welcome to read it but know that it will be subject to many changes before the book is released.

Chapter in Stage 1 - Proposal

This chapter is still in progress. The content may look complete but I am still unsure if it contains the final content. In most cases this chapter may be based on a blog post.

You are welcome to read it but know that it may be subject to many changes before the book is released.

Chapter in Stage 2 - Draft

Completed writing the contents of the chapter. I still need to review it, complete code samples in GitHub, exercises, cross references and story.

You can read this chapter with the assurance that it looks 90% how it will look in the final version. But remember that your feedback is always welcomed nonetheless! :)

Chapter in Stage 3 - Candidate

Completed writing the contents of the chapter and I have incorporated improvements suggested by my great reviewers. I still need to complete code samples in GitHub, exercises, cross references and story.

You can read this chapter with the assurance that it looks 90% how it will look in the final version. But remember that your feedback is always welcomed nonetheless! :)

Chapter in Stage 4 - Finished

Completed writing the contents of the chapter for all intents and purposes. It has the content reviewed, code samples ready in GitHub, exercises, cross references and story.

You can read this chapter with the assurance that it looks 99% how it will look in the final version. But remember that your feedback is always welcomed nonetheless! :)

After this I'll remove the statuses to start working on the final formatting for the book!

Ok! Sounds Awesome Jaime! But When Should I Jump In And Read?

Great question! That depends on what you want to get out of this book:

- If you want to tag along and experience how a book gets made and is slowly improved over time, provide feedback that has a bigger impact then read the chapters from **Proposal** or **Draft** onward.
- If you want to read a version as closer to the final version as possible and still would like to provide feedback and impact the book then **Candidate** and **Finished** are the ones for you.

In any case, enjoy the book and don't hesitate to drop me a mail if you have any questions or feedback you'd love to share.

Jaime
Fortress of Solitude, January 2019

About The Author

Jaime González García

Jaime González García (@Vintharas[1]) *Software Developer and UX guy, speaker, author & nerd*

Jaime is a full stack web developer and UX designer who thinks it's weird to write about himself in the third person. During the past few years of his career he has been slowly but surely specializing in front-end development and user experience, and somewhere and some time along the way he fell in love with JavaScript. He still enjoys developing in the full stack though, bringing ideas to life, building things from nothingness, beautiful things that are a pleasure and a delight to use.

Jaime works as a Front-end Software Engineer at Google. He spends part of his time as a Developer Relations in the Nordics developer community. He speaks at conferences, writes articles, runs workshops and talks to developers and companies about how they can do cool things with Angular and JavaScript. He also arranges developer community events in Stockholm as a way to support and encourage the thriving local dev ecosystem.

[1]https://twitter.com/Vintharas

In his spare time he builds his own products, writes the JavaScript-mancy series and blogs at barbarianmeetscoding.com (long story that one). He loves spending time with his beloved wife Malin and son Teo, drawing, writing, reading fantasy and sci-fi, and lifting heavy weights.

Once Upon a Time...

Once upon a time, in a faraway land, there was a beautiful hidden island with captivating white sandy beaches, lush green hills and mighty white peaked mountains. The natives called it ***Asturi*** *and, if not for an incredible and unexpected event, it would have remained hidden and forgotten from the annals of history. A small, insignificant, unremarkable place.*

Some say it was during his early morning walk, some say that it happened in the shower. Be that as it may, ***Branden Iech****, at the time the local eccentric and today considered the greatest Philosopher of antiquity, stumbled upon something that would change the world forever.*

In talking to himself, as both his most beloved companions and his most bitter detractors would attest was a habit of his, he stumbled upon the magic words of JavaScript and the mysterious REPL.

In the years that followed he would teach the magic arts and fund The Order of JavaScriptmancers bringing a golden age to our civilization. Poor, naive philosopher. For such power wielded by mere humans was meant to be misused, to corrupt their fragile hearts and bring their and our downfall.

It has been ten thousand years, ten thousand years of wars, pain and struggle.

It is said that, in the 12th day of the 12th month of the 12th age a hero will rise and bring balance to the world. That happens to be today.

12th Age, Guardian of Chronicles

This is a **weird** programming book. On one hand it is a awesome book on the superb vim text editor. On the other a spell book of sorts set in a world of fantasy where some people can wield JavaScript to affect the world around them. To essentially program the world and bend it to their will.

Welcome to the world of JavaScript-mancy, may you enjoy your stay, learn a lot and have a ton of fun.

Preamble

Chapter in Stage 1 - Proposal

This chapter is still in progress. The content may look complete but I am still unsure if it contains the final content. In most cases this chapter may be based on a blog post.

You are welcome to read it but know that it may be subject to many changes before the book is released.

Order of Vim:

Also Secret Order of Vim, The Circle, The Emerald Guard as detailed in appendix 4b of Annals of the 5th Age
Tagged as: Myth, Legend, Of dubious historical value

Used to refer to minor order of wizard warrior monks said to have existed in the golden period of the Age of Light. The order disappeared without a trace after The Great Wizard War. There is no reference to The Order after the 6th Age. The Order importance during the Age of Light is unknown, the most significant reference appears in “The Crimson Chant”:

“...As the Red Fist raises
The Emerald shatters,
its place taken by The Ruby,
the crimson, the scarlett.
The balance is broken,
the future uncertain,
Who shall now guard the kingdoms of Terra?”

The Order of Vim was said to have attained communion with The Pattern. Using a mysterious and rigorous training, they developed a secret technique that allowed them to tap directly into the threads of existence and achieve near miraculous feats of magic. Aside from vague references in a handful of texts, there is no proof that the Order ever existed. The few references we can find in other works of the same period are hazy, more akin to myths, legends and the product of a colourful and imaginative mind.

From *Myth and Legends Of The Age of Light, A Humble Essay*
Harvastin, The Scholar
Guardian of Chronicles of The Library of Sallas
12th Age

A Note From a Friend

Ey Harv, I found this book in that backwater hole of Gigia in the island of Asturi. In a flea market of all things! It was dusty and full of dirt but I think it is in pretty good state considering the circumstances. I think it is missing some portions as well... The cover seems to be wider than the pages it contains. Well, I think you'll find it very interesting anyway.

I'm not an expert on the Age of Light as you are but after perusing the pages of this tome I think it could constitute the first solid proof that we have in Millennia of the elusive Order of Vim. Exciting, isn't it?

I suspect that this book, **The Book of Vim**, was a writing cornerstone to the very essence of The Order and part of the training material used to indoctrinate their members. A part of the very secrets that made them oh-so-powerful, famed and prominent in the Age of Light (at least in Myth). I get goosebumps just to think that this, my friend, are the same pages that would be read and practiced by the heroes of old, **The Emerald Guard** (You know I'm always daydreaming about these things, that's why I focus on Mythology and leave the boring History to you).

Oh, one last thing. The book is no easy reading. Its pages are filled with strange archaisms long forgotten that make the text hard to decipher. You'll find numerous references to *"the keyboard"* (a board full of keys?) or *"the text editor"* as if these were as commonly understood words as water or bread. So many millennia after, we can only wonder as to what these words may refer to.

Anyway. I must go now. Adventure awaits and all that...
Hope you enjoy it! (I know you will)
Your friend, H.

P.S. I’ve found a deep cave complex in the entrails of the Misty Mountains. There, miles and miles under stone and earth I found the remains of a humongous battle. Very weird and exciting! Who, What and Why would battle in this far away and meaningless island? I’ll keep you updated.

The Book of Vim

Choosing the right editor is the first
step towards effective editing.

If you don't know which editor to use or are
dissatisfied with what you are currently using,
give Vim a try; **you won't be disappointed**.

  - Bram Moolenaar,
    High Alchemist of the Dutchy of Lisse,
    Scroll of Seven Habits of Effective Text Editing

Chapter in Stage 1 - Proposal

This chapter is still in progress. The content may look complete but I am still unsure if it contains the final content. In most cases this chapter may be based on a blog post.

You are welcome to read it but know that it may be subject to many changes before the book is released.

Welcome to The Order. You have been selected, hand-picked, because you've shown promise in the arcane arts of javascript-mancy, demonstrated sound judgment and an affinity for defeating evil. Yet for all your prowess you are but a baby out of the cradle: ignorant, unknowing, with senses so dull and underdeveloped that you can't even begin to understand how much lies beyond your understanding and how far you are from achieving your true potential.

We're the keepers of balance. We keep the world from burning. The discovery of JavaScript-mancy was a magnificent moment that started an Age of Wonders. But it also means that we now have a significant part of the population of Terra with the power to level mountains and split the world apart. We live in the shadows, we counsel, we nudge, we protect and as a last resort we strike with deadly might. But that means that we have to be better, wiser, smarter, stronger and faster than other practitioners of the arcane. And that's why we cultivate **The Path**. And The Path starts with **The Book of Vim**.

This is **The Book of Vim**. A collection of teachings about **Vim** from our wisest masters of the arcane. **Vim** is our text editor because unlike ferals and wild mages, we carefully craft and improve our spells, an even adapt the REPL to our own personal inclinations and strengths. This book will become your closest companion for the next weeks, years and ages to come. Read it, learn it, practice it, until it becomes part of your very being. Then, and only then will you get a glimpse of the possibilities that lie beyond.

Let's get started and lay the first stone of the foundation of the new you. An awesomer version, capable of anything you can imagine.

Follow the Path. Protect from the Shadows.

– Randalf Saa'den
Keeper of the Red Flame,
Quartermaster of The Order, 5th Age

What is Vim?

Vi[2] is an ancient text editor, old even before the first age of the world[3]. It was designed to work on contraptions called terminals with the very uncommon yet inspired characteristic of functioning in a modal fashion. That is, it has a mode for inserting text, another for editing text, a different one for selecting text, and so on. Quaint isn't it? Don't dismiss it though, you'll soon learn how powerful this concept can be.

Vi's latest and most celebrated incarnation is Vim[4] (**Vi IM**proved and formerly **Vi IM**itation) which works both with text and graphical interfaces, comes with a plethora of improvements over vi[5] and is supported on every platform known to wizardkind. But the impact of Vim doesn't stop with Vim, Vim's ideas are so very remarkable that they've trascended the Vim editor itself and propagated into other editors. Today you can find Vim-like modes in almost any editor and IDE[6] that you can imagine[7].

Why Vim?

Why should you care about learning an ancient editor in this day and age? Aren't there newer and better alternatives out there? The truth is that Vim provides a different way of interacting with text from anything I've ever seen, a way that gives you **a completely different level of control and fluency when editing code**. At the hands of an experienced user, editing text with Vim **seems like magic**.

[2]https://www.vim.org

[3]https://en.wikipedia.org/wiki/Vi

[4]https://en.wikipedia.org/wiki/Vim_(text_editor)

[5]https://en.wikipedia.org/wiki/Vim_(text_editor)#Features_and_improvements_over_vi

[6]IDE stands for Integrated Development Environment. Like a text editor but loaded with lots of additional functionality and tools. As a result IDEs are typically heavier, slower and more powerful than text editors.

[7]And even in popular applications such as Gmail or Twitter but let's not break the fourth wall just yet.

With vim, the main way in which you interact with your code is not through inserting characters and using the mouse to click, select and move around. Instead **you'll be like a code surgeon** that makes expert incisions with surgical precision whenever and wherever it is required, navigating through your code and codebase with the ninja lightning speed and accuracy of a **entirely keyboard driven workflow**[8].

Indeed, with Vim the keyboard reigns supreme. Any aspect of the editor can be controlled through the keyboard. Moreover, Vim is designed to **provide the touch typist with the highest degree of productivity**, with the most common commands comfortably laid out in the home row and its neighbouring keys. Vim is a modal editor with its *normal* mode at the forefront. A mode devised to read, navigate and transform code at will. The fact that vim has modes allows keys near the home row to be reused in each separate mode, minimizing the need for slow and contorted key combinations, and heightening your speed and the longevity of your fingers and wrists[9].

Vim is **extremely customizable** and you can adapt it to your way of coding and the manner in which you do things. Part of the beauty, and the complexity of learning vim, is how you can slowly but surely make it work in the way that you work, in your project, alone or with your team.

Finally, even if you don't jump straight into vim, even if you don't intend to have vim be your main editor, you can reap the rewards of learning vim by bringing all the commands and motions that you learn to your favorite editor. That's it! **Vim is so good that most other editors today support some sort of vim mode which brings a vim-like experience right into the comfort of your well known editor**. Following this approach you'll be able to improve your text editing skills while keeping all the time and knowledge you have invested in your current editor. A great example of this

[8]So a code ninja surgeon…
[9]Great if you suffer or have suffered from carpal tunnel syndrome or similar joint maladies

is Visual Studio Code with its VsCodeVim[10] extension which not only supports motions and commands but even custom mappings, ex mode and a lot more.

So **Why would you want to learn Vim in this day and age?** Paraphrasing the mighty Drew Neil[11] author of Practical Vim[12] [13] and master of the most obscure arts of Vim:

> Vim is for **programmers who want to raise their game**. In the hands of an expert, **Vim shreds text at the speed of thought**.

And who wouldn't want that, right? **Are you ready to raise your game?** Are you ready to begin your journey along The Path and into text editing awesomeness?

What About Neovim?

Over the past couple of years you may have heard people talking about Neovim and wondered... what the heck is that!?

Neovim is a modern fork of Vim that aims to refactor Vim and make it more maintainable, extensible and easier to contribute to by a wider community. It's main innovation over traditional Vim was that it supported asynchronous processing, an integrated terminal and external plugins. Vim 8 later caught up with support for asynchronous processing and an integrated terminal but Neovim still remains very relevant.

[10]https://github.com/VSCodeVim/Vim

[11]https://twitter.com/nelstrom

[12]https://amzn.to/2CIzSpb

[13]Practical Vim is an insanely awesome book on Vim. It follows a tips format from beginner's tips to more advanced ones. In each one of these tips, it provides great background information and context that help you understand not only how to improve your vim skills with new commands and features, but also wider patterns and ideas to help you think like a vim master. All of it with great examples and exercises to help you practice your muscle memory. The original quote reads: Practical Vim is for programmers who want to raise their game. In the hands of an expert, Vim shreds text at the speed of thought. Reading this book is your next step towards that end.

Neovim is indeed still very active, healthy and pushing the boundaries: Its support for external plugins written in JavaScript through node.js is a great boon to create an even more inclusive community of plugin writers. Its headless mode allows GUI applications (like Visual Studio Code[14] or Oni[15]) to consume Vim as a service. Its focus in a great user experience has brought cool new features to Vim like virtual lines with in-editor messages, floating windows and built-in LSP support just as what you have come to expect from modern text editors. Finally, its more sensible defaults make it a more approachable editor than the original Vim.

Moreover, the mere existence of Neovim as a competitor to Vim is something healthy for the community as a whole. For instance, features like the integrated terminal, async processing or floating windows were added to Vim after they were available in Neovim.

You can find out more information about Neovim at neovim.io[16].

A Brief Note About the Conventions Used in This Book

Since a lot of what happens in Vim depends on the location of your cursor, I've used a series of diagrams that show the position of the cursor changing over time as you type commands. And since it is quite unconventional from other programming books, I think you'll find it helpful to have it explained so you're prepared before you encounter it for the first time. Here's an example:

```
wwww ==> v   v v  v   v
         word. is two words
```

That means the following:

[14]https://code.visualstudio.com/
[15]https://www.onivim.io/
[16]https://neovim.io/

```
  commands you type    position of the
   /                    /   cursor changing
  /                    /    as you type
wwww ==> v   v v  v   v
         word. is two words
           /
          /
      text in your editor
```

So:

- The text `word. is two words` is the text that is inside your editor which is subject to change or navigation
- You type the command `w` successively (in this case 4 times)
- Every time you type the command, you move the cursor (represented by `v`) to a new location

At times, it will be helpful to compare how two commands perform when applied to the same bit of text. In those cases I've used the following diagrams:

```
wwww ==> v   v v  v   v
         word. is two words
         word. is one WORD
WWW  ==> ^     ^  ^   ^
```

Where the bottom part is similar in meaning to the top part we discussed earlier but for the fact that the cursor is represented by a caret `^` instead of `v`.

Other times it will be important to convey that some text has been selected (like in Visual mode). In order to that, we'll use an asterisk `*` to represent the breadth of the selected text:

```
  --red: #0F0;
        ******
```

When explaining commands we'll pay heed to the following conventions. For operations and motions:

```
f{character}

f            - f is a literal f, expected to
               be typed as-is.
{character}  - is a placeholder that needs to be
               substituted by something. The name
               between {} will be descriptive of
               what that something is expected to
               be. In this case a character.
```

When constructing and applying text-objects:

```
{operator}{a|i}{text-object}

{operator}    - placeholder
{a|i}         - either type the letter a or the letter i
{text-object} - another placeholder
```

And for ex-commands:

```
:[range]s/{pattern}/{substitute}/[flags]

:            - denotes the beginning of an ex-command.
[range]      - the [] denote that this part is optional.
               The name will be descriptive as in the
               case of placeholders.
s            - command to be typed as-is.
{pattern}    - again this is a placeholder.
{substitute} - another placeholder.
[flags]      - another optional part.
```

Give Feedback Freely!

If you find any diagram or explanation confusing, have any questions, want to provide feedback, or even if you enjoyed the book and want to say kudos! Don't hesitate to ask me on twitter at twitter.com/Vintharas[17]. My DMs are open and I'll always be more than happy to answer your questions.

[17]https://www.twitter.com/Vintharas

A Taste of Vim

// something something

- Someone,
  Some cool sounding title,
  Some even more cool sounding book or treatise

Chapter in Stage 1 - Proposal

This chapter is still in progress. The content may look complete but I am still unsure if it contains the final content. In most cases this chapter may be based on a blog post.

You are welcome to read it but know that it may be subject to many changes before the book is released.

The best way to get a feel for Vim and its capabilities is to jump straight to using it. The following chapter will give you a taste of what it is like to work with Vim. You don't need to start Vim, just follow along reading and don't worry if you don't understand everything right away or if some things seem weird. In fact, you can expect that things **will** get weird. But worry not! The remainder of the book will focus on explaining everything in detail so that by the end of it, you'll be a competent Vim user ready to eat the world. Achieving Vim mastery will then be up to you and whether you are ready, and wanting, to put in the practice it requires.

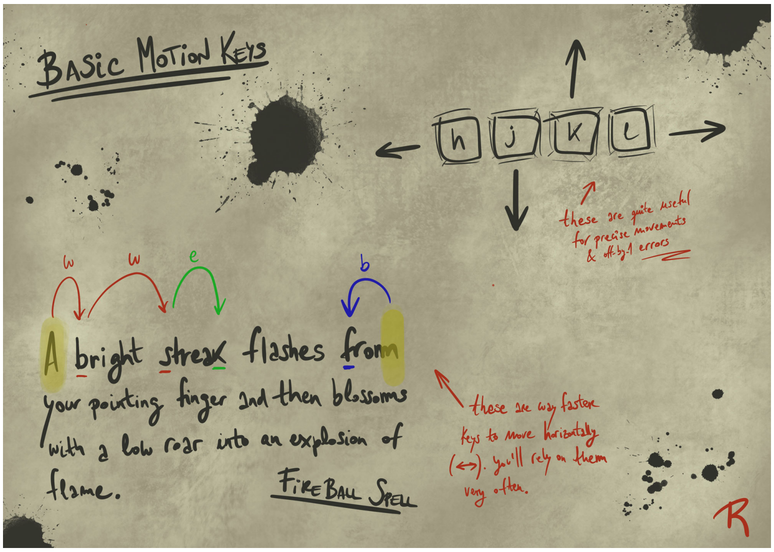
Excellent! Through the powers of arcanity this chapter will allow you to experience what it feels like to be an experienced Vim alchemist. Close your eyes and imagine… *(reality blurries and fades to dreamlike alternate world)*

You open Vim and you're embraced by Vim's core mode: normal mode. *"Hello, old friend"*, you think and smile. Unlike any other traditional editor, when using Vim you spend most of your time in this mode. Within it you don't explicitly write code. Instead, **normal mode is optimized for navigation and precise text changes** and makes you all powerful.

Your right hand rests firmly on[18] the core **motion** (as in movement) keys `hjkl`. These keys move the cursor around to left, down, up, and right respectively. You can also move left to right word by word using `w` (go to the beginning of next **w**ord) or `e` (go to the **e**nd of the next word), or use `b`/`ge` to do the same but from right to left.

This can feel intimidating for an initiate, so let's open a scroll with some fancy diagrams:

[18]The `h` key is beside your index finger readily available.


Basic Motion Keys
h
j
k
l
these are quite useful for precise movements & off-by-1 errors
w
w
e
b
A bright streak flashes from your pointing finger and then blossoms with a low roar into an explosion of flame.
Fireball Spell
these are way faster keys to move horizontally (↔). you'll rely on them very often.
R

Taking advantage of these keys you can navigate a file[19] with fine granularity and strike with great vengeance and spite: Type `daw` and bang! You **d**elete **a** **w**ord! `das` and you remove a **s**entence! `dap` and you obliterate a **p**aragraph! Fierce!

Or you can be less menacing and more nurturing and fix stuff instead. Using `caw`, `cas`, `cap` to **c**hange **a** **w**ord, **a** **s**entence or **a** **p**aragraph.

But it doesn't end there. You can `ctx` to **c**hange un**t**il the first `x` in the current line, or `c$` **c**hange everything until the end of line, or event better `ci(` to **c**hange the content **i**nside parentheses or `ci"` to do the same to the content **i**nside quotes.

On Notes, Melodies And Chords

Vim is quite special. If you've used other editors you're prob-

[19] A file is a synonym for scroll of magic. It's much shorter and convenient. We use it all the time.

ably accustomed to typing chords of keys. That is, typing a combination of keys at the same time. For instance, `CTRL-C` to copy and `CTRL-V` to paste. Vim uses chords as well but relies even more on **melodies of keys**.

If you think of keys as musical notes, a melody is a series of notes one after the other. That's the most common way to interact with vim when you're in normal mode. So, when you read that you need to type `f{char}` to find a character in a line it means that first you type `f` and then you type the character `{char}` in rapid succession (f.i. `fa` to find the first `a`).

Using melodies of keys, although unfamiliar and kind of strange, is very convenient. **Controlling the editor will suddenly feel like you're just typing text**. And I'm sure you are very accustomed to typing text. As a bonus, it is also great for the health of your wrists and fingers.

Imagine a simple string:

```
v
const msg = 'Hither came Conan, the Cimmerian, black-haired, sullen-eyed, \
sword in hand, a thief, a reaver, a slayer, with gigantic melancholies and\
 gigantic mirth, to tread the jeweled thrones of the Earth under his sanda\
led feet.'
```

The `v` and `^` in this examples point to where the cursor is placed in your vim editor. Because that is a big deal in Vim as you'll soon experience.

You can change the string by typing: `f'ci'WAT<ESC>`

```
const msg = 'WAT'
             ^
```

Which means:

1. Find the next single quote `'` (**`f'`**)
2. Change everything inside `'` for `WAT` (**`ci'`** removes everything inside `'` and drops you into **insert** mode where you can type **`WAT`**)
3. Then `<ESC>` to leave **insert** mode back to the cozy **normal** mode

More Productivity Plz!

Did you know you can achieve the same results in the previous example by just typing `ci'WAT<ESC>`. There is no need for `f'` because the `ci'` command seeks forward within a line for a pair of single quoted text.

As you become a more proficient Vim wizard, you'll find that there are many ways to edit text in Vim. Oftentimes your goal will be to learn how to achieve a task with the least number of keystrokes. But this is something that you'll learn gradually so don't worry about it for now.

If you have an obsessive character and feel like **you must optimize**, then follow the path of the vimgolf sorcerers[a].

[a] https://www.vimgolf.com/

Ok, so we have this string now:

```
const msg = 'WAT'
               ^
```

You can **y**ank this line with `yy` (which is Vim's extremely evocative jargon for copy) and **p**ut it below with `p` (again Vim jargon for paste):

```
const msg = 'WAT'
const msg = 'WAT'
^
```

Yes, you got it! **yank** is another operator like **delete** and **change**. You can use `y` just like `d` or `c` to **y**ank a **w**ord `yaw` or `yas` **y**ank a **s**entence.

Moreover doubling a command like so `yy` makes the command operate on the entire line. Nifty! It follows that `cc` changes a line and `dd` deletes one. Gosh, look at how much you're learning!

Now try `ci'MAN<ESC>` which results in:

```
const msg = 'WAT'
const msg = 'MAN'
               ^
```

Then go crazy and join the lines with `kJ` (`k` to go up and `J` to join lines):

```
const msg = 'WAT' const msg = 'MAN'
                 ^
```

Rinse with `c3w+<ESC>` (as in **c**hange the next **3** **w**ords for a `+`) and we've got ourselves:

```
const msg = 'WAT' + 'MAN'
                    ^
```

Which is a completely nonsensical exercise of using Vim to reveal the secret identity of Conan but which stills manages to show you part of the magic of Vim.

For **longer motions** you can use **counts** in combination with motions (I actually sneaked on of those earlier). For instance, you can go down 5 lines by typing `5j` (as in `{count}{motion}`). Likewise you can use these together with the operators you saw above (`d`, `c`, etc) and `d2w` detele two words, or `c2s` change 2 sentences. There's longer motions too, you can `H` to move to the top of the visible area in the editor, or `gg` to go to the top of the file, `L` and `G` to achieve the same but downwards. Use `{` to move up a whole paragraph or `}` to do the same but down. While `%` helps you find matching parentheses.

You can **start a new line below** with `o` (drops into insert mode so you can start typing) or above with `O`. You can find patterns (of text) forward within a file using `/{pattern}` and navigate between patterns using `n` (next) and `N` (previous). You can do the same thing backwards using `?{pattern}`.

You can repeat previous changes using the `.` command. Just type `.` and Vim will repeat your last change. Likewise you can repeat motions. Type `;` and you'll repeat a motion started with `t,f,T,F` or type `n` to repeat a search. You can record a collection of commands using macros and replay them over and over at your will. And there's so. Much. More...

So much power at your fingertips and we have barely left **normal** mode or the confines of a single file. There's splits, there's tabs, there's regex, there's access to external tools, there's spell checking, word count, there's 6 basic modes more with 6 additional variant modes and infinite extensibility and customization possibilities!

Who's excited!?

Then proceed apprentice.

Apprentice

// TODO: write something here pertaining being an apprentice. Intro to this part of the book and such.

Creating, Opening and Saving Files

// something something

- Someone,
  Some cool sounding title,
  Some even more cool sounding book or treatise

Chapter in Stage 1 - Proposal

This chapter is still in progress. The content may look complete but I am still unsure if it contains the final content. In most cases this chapter may be based on a blog post.

You are welcome to read it but know that it may be subject to many changes before the book is released.

Ok! **Let's get started!** First things firsts… in order to use Vim, you need to open it! The way you do that is through the terminal.

Open your favorite terminal and type the following:

```
$ nvim
```

Boom! This will open Neovim in the current directory (you can verify that by typing the `:pwd` command which stands for *print working directory*.).

Using Vim?

If you're using Vim then type `vim` instead of `nvim`. Alternatively, you can use `gvim` or `mvim` for the GUI counterparts of Vim in Windows/Linux and OSX, respectively.

Aliasing For The Win

Find yourself typing the same things within your terminal over and over? Then consider creating an alias.

For instance, use *v* instead of *vim* or *nvim*. Creating an alias in zsh is as easy as writing the following bit in your `.zshrc`:

```
# super economic Vim alias
alias v="nvim"
# alias vim to nvim so you never open Vim by mistake
alias vim="nvim"
```

Create once, enjoy for ever.

Ok, so `nvim` opens Neovim into… *nothingness*. We can do better than that. We can create a file! Type the following:

```
$ nvim this-is-a-new-javascript-file.js
```

Which opens a blank new JavaScript file with that weird long name (Sorry for making you type that).

The same trick also works for existing files. Let's open a file from **the Wizards Use Vim code samples**.

Wizards Use Vim Code Samples?

Haven't downloaded the code samples for Wizards Use Vim yet? Then transport thyself to GitHub, magic vault where all the source code for all the spells in Terra lie, and get them[a].

[a] https://github.com/Vintharas/wizards-use-vim-code-samples

Ok! Now I expect you to be comfortably poised at the root folder of the code samples repository. If you aren't, please go there and type the following:

```
$ nvim 001-welcome-to-vim.js
```

Voilá! Vim will open the file of your choice and you should now see something like this:

```
export function welcomeToVim() {
  console.log('Welcome to Vim Oh Powerful Wizard!');
}
```

But Vim's capabilities for opening files are far superior. Vim can open multiple files in seemingly infinite configurations.

For instance, in **separate tabs** with the `-p` option:

```
# open multiple files in separate tabs
$ vim -p src/hello.js src/world.js
```

Or in **horizontally split windows** with `-o`:

```
# open multiple files in separate windows split horizontally
$ vim -o src/hello.js src/world.js
```

In **vertically split windows** using `-O`:

```
# open multiple files in separate windows split vertically
$ vim -O src/hello.js src/world.js
```

And all of these options also work when using a glob pattern:

```
# open multiple files using a glob pattern
# This particular glob means:
#   Open all js files in the src folder.
#   The -O option opens them in separate
#   windows split vertically
# you can also use -o and -p
$ vim -O src/*.js
```

Not Familiar with Glob?

glob, short for *global*, is a way to represent a set of filenames using wildcard characters in combination with normal characters. The most common wildcards are:

- `*` which is any character, any number of times
- `?` which is any character once
- `[abc]` matches any character within brackets once
- `[a-z]` matches any character within the specified range.

For instance, `./*.js` will match all JavaScript files in the current folder, `src/*.js` will match all JavaScript files in the `src` folder, `src/**/*.js` will match all JavaScript files in any folder inside the `src` folder.

Not surprisingly you can also **open files from within Vim**. You can achieve this feat using the `:edit {pathToFile}` command (shorthand `:e`) which will open a file in your current window.

This command, like many others, supports TAB completion. That is, if you type `:edit ` and then press `<TAB>`, you'll be offered a list of choices just like you would when you're in the terminal.

You can use the same `:e` command to create a new file. Type `:e {pathOfTheNewFile}` and a new file springs to life. Beware though! The new file will be kept only in memory until you save it with `:write` (shorthand `:w`).

On Buffers And Files

Whenever you open a file in Vim and start making changes you're not interacting with the file itself. Instead, you're working against an in-memory representation of that file. This in-memory representation is called a **buffer** in Vim terminology. You only make your changes persistent when you save that file using the `:write` command.

Would you like to learn more about buffers? The help is your friend. Type `:h buffers` and have fun.

I love saving things. I have this coding mannerism where I save my changes very often, as often as other people say "like" or "eeehmm". If you're like me, you may benefit from having a custom mapping for saving files. `:w` is great but it takes 3 keystrokes and a `:` which is somewhat hard to type with that `SHIFT` key in the way. So consider adding this to your `vimrc`

```
" As soon as you type <leader>w
" your file will be saved
noremap <leader>w :w<CR>
```

The `:e` command expects a full path in relation to your working directory (remember `:pwd`?). A useful command for creating new files in the same folder as the one you're currently editing is to use:

```
:e %:h\{newFileName}`
```

The `%` stands for current file, the `:h` is a modifier that gives you the directory in which the file is located, and then we just append the name of the new file we're interested in creating.

How Can % Be The Current File?

When you type ex commands (by writing `:{command}`) you enter command-line mode. In that mode, all commands that expect a filename can take advantage of a handful of special characters that work as shortcuts for useful filenames. `%` is one of those special characters which represents the path to the current file.

You can find more info about these special characters in `:h :_%` or `:h cmdline-special`

Ok! So let's say that you've been working on your beautiful application for a while and you're done editing things. Now you can either:

- **save a file and quit** with `:exit` (shorthand `:x`).
- quit without saving `:quit!` (shorthand `:q!`).

Or if you've been working on multiple files you can:

- use `:xall` (shorthand `:xa`) to save all of them and quit
- or, use `:qall!` (shorthand `:qa!`) to quit without saving

There's a lot of combinations of `:write` and `:quit` which will prompt you whether you want to save or quit under different conditions. If you are curious you can take a look at the help for all these commands but knowing the commands above should be enough 99% of the times. (Try `:h save-file` and `:h save-quit` for more info)

So now you can open **AND** more importantly **exit** vim. No more getting trapped inside vim. Ever. Again. (*Pheeeeww*)

Overwhelmed by All The Commands?

I know! There are loooooads of commands in Vim. But bear with me. A lot of the commands are easy to learn or use by just… guessing!

Think about what you want to do or the action that you want to perform. It is very likely that the word you're thinking is the name of the command you're looking for: *edit* a file, *close* or *quit* the current window, etc.

When in doubt, use Vim's help.

So! We're inside a code file. **What can we do next?**

Summary

Summarizing what you've learned at the end of a chapter is a great indicator of the quality of a book. But in this book we're going to do something different. **Something better**. **We're going to journal**, and **you**, my friend, are to going write the summary of what you've learned as you read each chapter. Using. Vim. *You're going to write down what you learn of Vim, with Vim*, this is gorgeously meta. Aside from the meta-awesomeness there's more advantages:

- You'll be able to see how much you learn on each chapter
- You'll consolidate the concepts you've learned
- You can reflect on how you can incorporate the things you learn in you current workflow. *Is this approach more effective that what I am currently doing?*
- You'll be able to use it as reference later on

Exercise Apprentice of Vim!

We're going to start The Journal of Vim!

1. Create a new markdown file using what you've learned in this chapter. Use a colourful name, this is a book of magic for Crom's sake!
2. Write down what you've learned in this chapter. If you want, you can use the template below. Also in order to insert text, you'll need to go into insert mode. Use the help to find out how to insert text in Vim. (Hint `:h insert` and `:h i_<Esc>`)
3. Save the file
4. Take a look and admire how much you've learned already

```
# Journal of Vim Awesomeness
## Date - Chapter

- I learned:
  - This and that
- Thoughts:
  - This is dope I'm going to be so productive!
- Can I use this to improve by current setup? How?
  - Yes!
```

Now practice opening the files in the code samples repository using different window and tab configurations:

- Open two JavaScript files in vertically split windows
- Close all of them
- Open all JavaScript files in different tabs
- Insert some arbitrary text in a file and then close all of them without saving anything

Help all the things! I'll keep adding section on this until `:help` feels super comfy to you:

- `:h :pwd` for more info about the `:pwd` command
- `:h :edit` for more info about the `:edit` command
- `:h save-file` and `:h quite-file` for more info about saving and quitting

Inserting Text

// something something

- Someone,
  Some cool sounding title,
  Some even more cool sounding book or treatise

Chapter in Stage 1 - Proposal

This chapter is still in progress. The content may look complete but I am still unsure if it contains the final content. In most cases this chapter may be based on a blog post.

You are welcome to read it but know that it may be subject to many changes before the book is released.

What can we do in a code file? We can write some code! With this and the past chapter we're aiming at making you comfortable in Vim and put you at a similar level of familiarity to what you would expect from other editors. So let's **insert some code**!

Inserting Code a.k.a. Writing Code

In Vim, you write code in **Insert mode**. There are two core commands that put you into Insert mode:

- `i` for **i**nsert and,
- `a` for **a**ppend

The `i` insert command puts you in Insert mode **before** the cursor. While the `a` append command puts you in Insert mode **after** the cursor (as if to append stuff wherever the cursor is placed). From then on you're in Insert mode and Vim pretty much behaves like any other editor.

Like with many other Vim commands `i` and `a` have uppercase counterparts that do **stronger** versions of inserting and appending:

- `I` puts you in **Insert mode at the beginning of the current line** whilst,
- `A` puts you in **Insert mode at the end of the current line**

Eventually though you'll want to exit *Insert mode* and do other stuff taking advantage of the powerful *Normal mode*. There are three ways to do this: `<ESC>`, `C-[` and `C-C`. Of all of these, I've found that the easier one to type by far is `C-C`. However, this is such an extremely common task that it pays of to have a simpler custom mapping that lets you come back to Normal mode seamlessly. So we'll add a new mapping to your **vimrc**.

Remember the vimrc?

We first learnt about it a couple of chapters back so you may have forgotten. `vimrc` is what we call your Vim configuration file.

You can easily open it from within Vim typing `:e $MYVIMRC`. We created a custom mapping that allows you to quickly open it in a vertical split by typing `<Space>ev`.

If you want to have your abbreviations ready as soon as you add them to your configuration then refresh your configuration by typing `:source $MYVIMRC`. Again, we created another convenient mapping for that with `<Space>sv`.

Ok! Enough repetition! I promise I won't clarify again what the `vimrc` is.

Add this to your `vimrc`:

```
" The 'i' before noremap means that this mapping
" will only be available in Insert mode
inoremap jk <ESC>
```

From now on, you can exit by typing **`jk`** in rapid succession. These keys are just below the index and middle fingers of your right hand, and they roll very naturally. Try it out! It's wonderful and fast!

Now try typing just **`j`**. Did you notice how Vim stops for a heartbeat before moving the cursor after the **`j`**? This is what happens when you configure several mappings that start with the same key. In *Insert Mode* the **`j`** key has a function (to insert the letter j). By defining a new mapping **`jk`** Vim no longer knows what to do when it encounters a single `j`. It needs to wait and see what you'll type in next or it'll just determine you want to type a `j` if you don't type anything in a while. This isn't a big deal in this case, but later in the book we'll learn when this behavior can have a negative impact in your workflow and how to avoid that.

Let's see all these commands in action with an example. Open the next code sample from the *Wizards Use Vim* repository:

```
:e 002-on-the-art-of-naming-spells.md
```

This will show a markdown file that looks like this:

```
# On The Art of Naming Spells

Naming spells is extremely hard. You have to be intentional, you have to b\
e descriptive, but most importantly, you have to be true to yourself, your\
 voice, your essence as a Wizard. Your spells are a representation of your\
self, but elevated in the plane of the exoretic and arcane.

Through the ages I have developed a system that has treated me well and co\
ntributed in a high measure to receiving the illustrious Bonel Prize of Sp\
ellcrafting in no less than 7 ocassions.

The system goes as follows. You pick a word:

Fireball

And you append and prepend two particles:

Igneous Fireball of Incandescence

Boom! Brilliant! The more redundant and obnoxious the better. Can you crea\
te some awesome spells with the following words?

Imp
Agility
Sleep
Portal
Chorizo
Healing

-- Randalf Saa'den
Keeper of the Red Flame,
Quartermaster of The Order, 5th Age
```

After we open the file the cursor will be located in the first character. We're going to move it to the beginning of the word `Imp` and insert the name of a terrifying and powerful spell.

How to Move the Cursor Up and Down?

We're getting a little bit ahead of ourselves but knowledge in Vim has some cyclical dependencies and it forces me to teach you a little bit about motions before fully presenting motions.

Motions or motion commands is how you move in Vim when you're in Normal mode.

To move up and down type `j` (for down) and `k` (for up). These two keys are precisely below your right hand's index, and middle fingers.

In order to do that, we will use the `j` key to slowly but surely move the cursor down to the beginning of the word `Imp`.

```
Imp
^
```

Once there we type `i` to get into Insert mode just before the word `Imp` and type:

```
iSummon Fire <ESC>
```

Which results in:

```
Summon Fire Imp
           ^
```

Now we want to append something after Imp. *How can we achieve that?* Exactly! We can use the `A` command to append something at the end of a line:

```
A of Mayhem<ESC>
```

Which becomes:

```
Summon Fire Imp of Mayhem
                         ^
```

Excellent! Well done!

In addition to `i` and `a` there are another three **super useful** commands that I love to use to drop into Insert mode:

1. `o` **inserts a new line below the current one and drops you into Insert mode** (mnemonic **o**pen a line below)
2. `O` inserts a **new line above** the current one and also drops you into Insert mode
3. `gi` puts you into Insert mode **at the last place you left Insert mode**. This is great if you drop out of Insert mode by mistake and want to go back where you were and continue typing.

Correcting Mistakes

Ok, so let's say that now you are in Insert mode, typing away and you make a mistake, like a typo. Do you go back to normal mode, fix the typo and go back into Insert mode? **No!** Much too slow and it'll kill your flow. Instead, there's a couple of bindings that can help you fix an error right from within Insert mode:

- `C-H` lets you delete the last character you typed
- `C-W` lets you delete the last word you typed
- `C-U` lets you delete the last line you typed

Using any of these you can type, type, type, ooooops! delete char, word or line and type, type, type.

On Correctness

To be more correct, the key bindings above don't remove the last thing you typed. They remove the character, word or line **before the cursor**. The last thing you typed is the special case when you happen to be typing on a new line.

Since this is an introductory text to Vim, it will happen on occasion that I'll deliberately lie to you in order to make things more approachable and useful.

Abbreviations. They Sound Boring But They Are Awesome

Abbreviations are the snippets of Vim. They are a simplified version of the snippets that you can find in other editors and IDEs (there are Vim plugins for that) but they're very handy nonetheless.

A cool thing about abbreviations is that they are expanded automatically as you finish typing them and pressing `<Space>` which fits in perfectly with the natural flow of typing text.

Say I'm typing the following:

```
You konw nothing Jon Snow
```

Having an abbreviation to correct the misspelling of `konw` would automatically substitute that word as I type it for the right spelling:

```
You know nothing Jon Snow
```

Neat right?

You define abbreviations using the `:iabbrev` command (shorthand `:iab`). The `i` in `iab` stands for *Insert mode* as these are abbreviations that are only applied to Insert mode. For instance:

```
:iab konw know
```

Now every time that I misspell the word 'know' as I often do, Vim will come to the rescue and fix it for me. I don't even need to do anything myself. I just type `konw` and Vim will just fix it. **Sweet!**

But abbreviations aren't just for misspellings. You can use them as, well, abbreviations. Is there some moderately long or complex bit of text that you keep writing over and over again? That's a perfect candidate for an abbreviation:

```
:iab wuv http://www.wizards-use-vim.com/
```

Finally, you can also use abbreviations as code snippets:

```
" The Left command puts the cursor
" inside the function argument section
:iab f function(){}<Left><Left><Left>
```

Snippets work quite well in tandem with autocommands or filetype plugins which ensure that they are only loaded in file types where they make sense. That is, you want your JavaScript snippets to be available only when in JavaScript. For now however we shan't go deeper into these waters. We'll keep autocommands and filetype plugins for later chapters within the book.

Abbreviation! Expand Yourself!

In addition to being expanded automatically, you can expand abbreviations explicitly by typing `C-]`. This allows you to expand an abbreviation without inserting any additional characters.

Up until now we've just defined abbreviations ad hoc using Ex commands. Since it is very likely that you want to reuse your abbreviations, you'll want to add them to your **vimrc** file:

```
iab konw know
iab bmc http://www.barbarianmeetscoding.com/
iab f function(){}<Left><Left><Left>
```

Superb! Now we're ready to dive in one of the things that make Vim special: **The Mystical Normal mode**, **Motions and Operators**!

Exercise Apprentice of Vim!

- Experiment with `i`,`a`,`I`,`A` and the rest of the commands you've learned in this chapter by naming more awesome spells. Create some new spells using `o` and `O`.
- Continue updating your Journal of Vim and include what you've learned in this chapter
- Create an abbreviation for `teh` to `the` and test that it works
- Add that abbreviation to your **vimrc**
- Take a look at the help for `:h Insert`
- Find out more about abbreviations by using the help `:h abbreviations`. Is it possible to define abbreviations in modes other than *Insert mode*?

Moving Around With Motions

// something something

- Someone,
  Some cool sounding title,
  Some even more cool sounding book or treatise

Chapter in Stage 1 - Proposal

This chapter is still in progress. The content may look complete but I am still unsure if it contains the final content. In most cases this chapter may be based on a blog post.

You are welcome to read it but know that it may be subject to many changes before the book is released.

Modes are one of defining features of Vim. And amongst them, were we to compare modes to Gods, **Normal mode** reigns supreme as the Allmother, God amongst Gods, to whom all must bow and obey (that or risk being chastised with the sole of a massive, godly slipper).

As a Vim acolyte **Normal mode** is where you'll spend most of your time. This innovative notion springs from the realization that we spend far more time reading, navigating and changing code, than we do writing it from nothingness. *Normal mode* is thus designed, to make you extremely proficient at navigating and changing code, like a true wizard. And with enough practice even like a demi-god.

But How can Vim achieve this? That's what the next chapters revolve around. Sounds interesting? Then jump right in!

Moving Swiftly Inside A File With Vim Motions

Are you a Touch Typist?

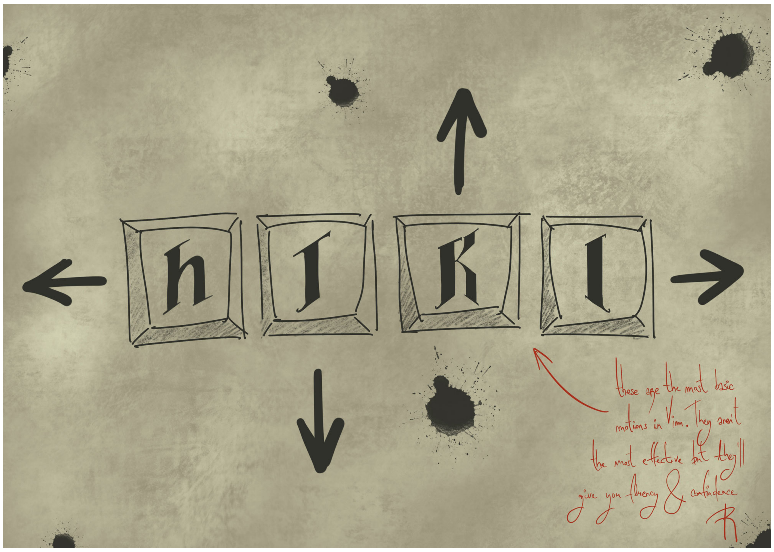
Can you touch type? Touch typing is extremely helpful when learning and using Vim. In order to be the most effective with Vim, your hands must rest in the home row and your fingers should reach up and down like those of a touch typist.

In *Here Be Dragons* I shared a bunch of resources that can help you become a better touch typist. Go back and take a look if you haven't read it.

Motions (as in movements) are how you move around in Vim. They are commands that when typed move the cursor around with high speed and precision. There are many of them, and each one is best suited for different types and lengths of movement.

`hjkl` are the core and most basic motions in Vim. They allow you to move the cursor:

- `h`: to the **left**
- `j`: **downwards**
- `k`: **upwards**
- `l`: to the **right**


h
j
k
l
these are the most basic motions in Vim. They aren't the most effective but they'll give you fluency & confidence
R

They are not the most effective, nor the most efficient way to move around in Vim. But one thing that they do, which is very important as a first-time student of the arts of Vim, is to **give you agility and confidence** to move around a file in *Normal mode*.

Learning to use `hjkl` effectively is the equivalent of learning to walk, or learning to ride a bipedal-cart-contraption (in some regions also referred to as bi-cycle) for the first time. After you get comfortable with more Vim motions you won't use `hjkl` as much, but they'll come **very handy for short-distance movements and small corrections**.

Moving Horizontally

A better way to **move horizontally** (that is left-right or right-left), is to jump around word by word.

- `w` lets you jump to the **beginning** of **the next word**
- `e` moves the cursor to the **end** of a word and from there to the **end** of the **next word**
- `b` helps you jump backwards to the **beginning** of a word and from there to the **beginning** of the **previous word**
- `ge` sends the cursor backwards to the **end** of **the previous word**

// TODO: Add image with word motions explained // And some examples inside image as well

Remember The Melodies

In a *Taste Of Vim* we discussed how Vim relies heavily in the use of melodies, a series of keys typed swiftly after the other. Whenever you see something like `ge` it means that you should first type `g` and then rapidly `e` to get the desired effects.

In Vim not all words are created equal and that distinction is very helpful to us developers as you'll soon see. Within Vim, there's words, and then there's WORDS:

- A **word** is a sequence of letters, digits and underscores separated by whitespace,
- Whilst a **WORD** is a sequence of **any** non-blank characters separated by whitespace

That is, **the difference between words and WORDS is that the former only include letters, digits and numbers**. That and the fact that any time you explain what a WORD is to a fellow Vim apprentice you must scream it to the top of your lungs (WOOOO-ORD!).

So, in addition to be able to move word by word, Vim also provides a series of motions that let you jump from WORD to WORD. These are the capitalized versions of the previous motions: Where we used `w` we will now use `W`, where we used `b` now we'll use `B`, `E` instead of `e` and `gE` in place for `ge`. Again this reinforces the notion that commands with capital letters are oftentimes stronger versions of the ones with lowercase letters. In this case, **capitalized WORD based motions allow you to move to and from larger distances**.

Let's look at an example to coalesce the difference between words and WORDs. You are welcome to follow along by opening this file from the *Wizards Use Vim* code samples:

```
$ nvim 003-fireball-spell.js
```

Upon opening that file you'll find the treasured Fireball spell:

```
export const fireballSpell = {
  manaCost: 10,
  name: "Fireball",
  type: "Fire",
  description:
    "A bright streak flashes from your pointing finger to a point you choo\
se within range and then blossoms with a low roar into an explosion of fla\
me.",
  damage: "d20",
  modifier: 8
  range: 20,
  area: 5
};
```

Let's focus on one of the properties of this JavaScript object and see how the use of `w` and `W` differ. Let's say that we want to change the mana cost of Fireball. `10` sounds like way too little mana for one of the most classic spells ever devised. In order to do that, we need to move as fast as we can over to the digit `10`. Let's see how `w` and `W` fare in this task:

```
using www =>  v       v v
              manaCost: 10,
using WW  =>  ^         ^
```

As we expected, using `W` brings us faster to the desired position with two (`WW`) versus there (`www`) keystrokes. This doesn't mean there are no use cases where `w` is equally effective. It just depends on the use case.

High Precision Motions With Find Character

To **move horizontally even faster** you can use `f` and `t` (which have the `F` and `T` variants to go backwards).

`f` lets you **find the next occurrence of a character** in the line you're in. `f{char}` (as in `f"` f.i.) brings you to the next occurrence of that character (`"` in our example).

If we review our previous example with the mana cost, we could achieve a similar result by typing `f1`:

```
using WW  =>  v         v
              manaCost: 10,
using f1  =>            ^
```

`f1` takes us exactly where we want to go in just one motion. Whether to use `W` or `f` will depend on how far to the right the desired position is. For further positions `f` will always be a superior choice.

Repeating Searches

After using `f{char}` you can type `;` to go to the next occurrence or `,` to go to the previous one. **You can see the `;` and `,` as commands for repeating the last search**.

The previous example won't be super helpful to illustrate the use of `;` so we'll pick another one. Let's say that we now want to change the type of this spell from `"Fire"` to `"Fira"` following the tradition of the popular masterpiece Final Fantasy[20].

```
  type: "Fire",
```

In order to do that, one solution would be to position our cursor of the `e` of `Fire` and change that letter for an `a`. We could do that as follows:

```
Type WWe =>   v     v   v
              type: "Fire",
Type fe; =>      ^      ^
```

The `t` motion is very much like `f`. The only difference is that `f` places the cursor on top of the character you want to find whereas `t` (think of until) places it just before:

```
type f{ =>                             v
          export const fireballSpell = {
type t{ =>                            ^
```

Knowing the difference between `f` and `t` will be helpful when combining these motions with operators such as *delete* and *change* as you'll soon discover.

Moving To The End Or The Beginning

To **move extremely horizontally**, that is, to the beginning or the end of a line you can use:

- **`0`**: Moves to the **first character of a line**

[20]Where Fira denotes a more powerful version of a Fire Spell. A Level 2 Fire spell to be specific.

- `^`: Moves to the **first non-blank character of a line** (so it would exclude the indentation of a line of code for instance)
- `$`: Moves to the **end of a line**
- `g_`: Moves to the **non-blank character at the end of a line**

All of these are quite hard to type so I prefer to use a couple of custom mappings to move to the beginning, and to the end of the current line. Add this to your **vimrc**:

```
" move to the first character in a line
noremap H ^
" move to the last character in a line
noremap L g_
```

Now, you can use use `H` (stronger version of `h`) to move to the beginning of a line and `L` (stronger version of `l`) to move to then end of a line. Makes sense, doesn't it?

Also now, with this mapping, I'm being a hypocrite and breaking my rule for sensible mappings (remember? When I recommended you to use `<leader>` when creating custom mappings?).

You may have guessed it already. The default `H` and `L` mappings are overwritten by the custom mappings above. But this is one of those instances, where the default mappings aren't really that useful. `H` normally takes you to the top of the current window while `L` takes you to the bottom (see `:h H` and `:h L` for more information). In all my years using Vim, I've never used these mappings for their default purpose.

Moving Vertically

Starting from `k` and `j` which move the cursor one line up and down respectively, we move on to a **faster way of maneuvering vertically** which are `{` and `}`:

- `{` lets you **jump entire paragraphs forward** (downwards)

- `}` allows you to jump entire paragraphs **backwards** (upwards)

These two have a couple of disadvantages, in my opinion:

1. The `{` and `}` are hard to type (they require you to flex both of those pinky fingers far and wide)
2. Moving down a paragraph may take you too far. For instance, if you have a long-ish method with no whitespace you may be brought to the end of the method, or even worse the end of the class after many other methods. That's probably not what you want.

A slightly better way to **move vertically** is to scroll up and down by half a page:

1. `C-D` let's you **move down half a page**
2. `C-U` let's you **move up half a page**

A Must Have Before You Move On!

One thing that will make your Vim experience orders of magnitude better is to remap your useless `Caps Lock` key to `CTRL`. Doing that moves a super helpful modifier key to the home row and makes it easily reachable using the pinky finger.

If you're using a Mac you can do this directly from within your keyboard preferences. If using other operative system just Google it. **Remap this key**. **You won't regret it**.

Changing How Much You Scroll

Is half a page too much? Do you feel lost after each jump? You can change how many lines you move vertically by using the `scroll` option. For instance, try `:set scroll=5` and try `C-D` and `C-U` again.

Remember that you can make this setting persistent by updating your **vimrc** adding the following:

```
set scroll=5
```

High Precision Motions With Search Pattern

To **move vertically faster** when you actually have a target in mind, your best option is to **search** with the `/{pattern}` and `?{pattern}` commands:

- Use `/{pattern}` to **search forward** inside a file (downwards)
- Use `?{pattern}` to **search backwards** inside a file (upwards)

Now let's try using `/{pattern}` to find something in our previous *fireball* example. For instance, the word `fire`. Go back to the code sample within Vim and type `/fire`.

```
// I hid all lines ut those containing the word "fire"
// for your convenience
export const fireballSpell = {
  ...
  name: "Fireball",
  type: "Fire",
  ...
}
```

You'll see that as you type, the patterns matched by whatever you write appear highlighted within Vim. When you find what you want, you can type `<Enter>` and your cursor will jump to the first match in the search. There you can perform some editing if you want and later use `n` to jump to the next match (or `N` for the previous one). **You can think of `n` as repeating a search**. Try it with the fireball example. What happens when you are in the last match of the file and you press `n`?

// TODO: Add image that shows the highlighted search. Be fun

Not Seeing Any Highlights?

Depending on how your Vim is setup you may not see the pattern being highlighted as you type. If that is the case, try adding the following to your vimrc:

```
" highlight matches as you type
set incsearch
" highlight matches once a search is complete
set hlsearch
```

Neovim users get this enabled by default. Good Olde Neovim!

Vim loves saving you time: At any time, you can type `n`, `N` to jump to the next or previous matches of your last search. Alternative, you can type `/<Enter>` or `?<Enter>` to run the last search forwards or backwards. Also if you happen to have the cursor on top a word, and you want to see where other instances of that word appear within the document you can type `*`. **The asterisk `*` will trigger a search for that word equivalent to `/{wordUnderTheCursor}` and send you flying to the next occurrence of that word**.

n, N vs /<Enter>, ?<Enter>

If all `n`, `N`, `/<Enter>` and `?<Enter>` repeat searches you may be wondering which one you should use. In most ocassions you'll want to use `n` and `N` because they're just one keystroke. The only advantage of using `/<Enter>` and `?<Enter>` is that they allow you to change the direction of a search.

Before we move forward here's a great tip: When you do a search, the pattern that you're looking for will be highlighted **FOR EVER** until you say otherwise. The reason for this is that it's helpful when you're repeating searches with `n` and `N`. But eventually you'll want those highlights gone so you can continue working peacefully on something else.

There's a command called `:nohlsearch` or `:noh` for short that can help you clear all the highlighted text from the previous search (see `:h :noh` for more info). This is such a frequent task that I suggest you to add a custom mapping for it. Add the following line to your **vimrc**:

```
" type <space> twice to clean the highlighted search
noremap <leader><space> :nohl<CR>
```

Go back to the fireball example and move the cursor on top of the first `A` inside the description field. Type `*` and you'll see how all occurrences of `a` get highlighted. Try jumping from one to the other with `n` and `N`, when you get tired type the `:noh` command and see the highlights vanish.

Noticed Something Interesting?

Did you notice how if you search for `fire` you also matched the word `Fire` that is capitalized? That's because we have configured Neovim to work that way via the `ignorecase` and `smartcase` commands. To find out more about these commands take a look at the help.

Moving Faster With Counts

We've seen a lot of motions in this chapter. Some of them used for short movements like `hjkl`, others for large ones like `{`, yet other ones for high precision assaults like `f{char}` or `/{pattern}`. You would think that that is enough but Vim has yet one additional trick that can make moving even faster: **counts**.

Counts are numbers which can be prefixed to a command, like any of the motions we've seen thus far, to multiply the effect of that command. For instance, `2w` allows us to move the cursor `2` words forward, likewise `2fe` sends us to the second occurrence of the character `e` within a line. And this also works with search repeaters like `;` or `n`.

Let's review the now-super-familiar-to-the-brink-of-nearly-hating-Fireballs Fireball example and illustrate the effect of counts:

```
using 3w =>            v
             manaCost: 10,
using 2W  =>           ^
```

A great way to move vertically is to take advantage of counts in combination with `j` and `k`. One of my favorite ways to browse a code file is by using `5j` and `5k`. This is much nicer than `C-d` and `C-u` because it moves the cursor instead of scrolling the whole buffer. Why is that? Because a moving cursor is much easier to follow with your eyes (and brain) than a scrolling buffer. But don't just take my word for it. Try it yourself and you'll see what I mean.

Typing `5j` and `5k` can be tedious and that's the type of thing you want to avoid by defining your own custom mappings. Let's add the following mapping to your **vimrc**:

```
" Move down file lines
noremap J 5j
" Move up file lines
noremap K 5k
```

Awesome! Now `J` and `K` have default behaviors that are actually useful. `J` lets you join lines together and `K` lets you search for help for the term under your cursor. A good approach in this case, is to keep the default mappings available but downgrade how easy they are to type. We can do that with the `<leader>` key by adding this to your **vimrc**:

```
" join lines
noremap <leader>j J
" keyword search
noremap <leader>k K
```

Excellent! Now we have some useful and frequently used mappings closer to our fingers. And some also useful but more rarerly used mappings still available but a little harder to type. As you hone your Vim skills and learn to know your Vim workflow you'll find yourself making these tradeoffs.

Exercise Apprentice of Vim!

- Apply the motions that you've learned in this chapter and complete **The Labyrinth** for great honor and infinite treasure:

```
$ nvim 003-exercise-the-labyrinth.txt
```

- Continue practicing these motions when you have an opportunity at work or home. Bored at breakfast? **Practice the motions**. Passé conversation in the coffee machine at work? **Practice the motions**. Your boss asks you for an important presentation or design document? **Practice the motions**. Crunching deadline to release a product? **Practice the motions**.
- Continue updating your Journal of Vim and include what you've learned in this chapter.
- Find out more about words and WORDS with `:h word` and `:h WORD`.
- See if you can find the third chapter of the Vim user manual titled *Moving Around*. Skim it, reinforce what you've learned and see if you can learn something new.