Vim as an IDE

Writing Code Faster with Free Open-Source Software

Disclaimer for Windows Users

This presentation is primarily directed at working with *nix-flavored systems. The provided materials have not been tested on Windows systems. If you have a Windows system and still want to use items from this presentation, you can spin up a light-weight virtual machine with VirtualBox to emulate a *nix-flavored system (such as Ubuntu or Fedora/RedHat).

Alternatively, you could go through the many great pains of debugging all of the plugins featured in this presentation in a Windows environment by yourself (and waste many hours on StackOverflow in the process).

The choice is yours.

First Things First

- Please install Homebrew if you do not already have it by running the following command in your terminal:
    - `"/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"`
- Please run `brew install git` if you do not already have the `git` command line tool installed on your machine.

Humble Beginnings

Grokking Vim’s predecessor, vi

The Basics

First steps on the path to mastery

The Philosophy of vi

- Code is *read*, code is *written*, and code is *shared*
  - “Normal” mode is for moving through the text
  - “Insert” mode is for writing to the text
  - “Visual” mode is for selecting portions of text
- Normal mode is *and should be* the default mode
- Enter insert mode with the `i` key (and some others)
- Enter visual mode with the `v` key (or `V` for whole lines/blocks)
- `<Esc>` key always returns to Normal mode

Vi as a Language

- Vi contains a succinct ***language*** for editing text
  - Text Objects and Motions -> nouns
  - Commands -> verbs
  - Numbers -> adjectives
- The Vi command language is designed around ***flow***
  - Ideas should flow from brain to keyboard without strain
  - Do more with less keystrokes
  - Less keystrokes -> less time writing code
  - Less time writing -> more time thinking (and debugging)

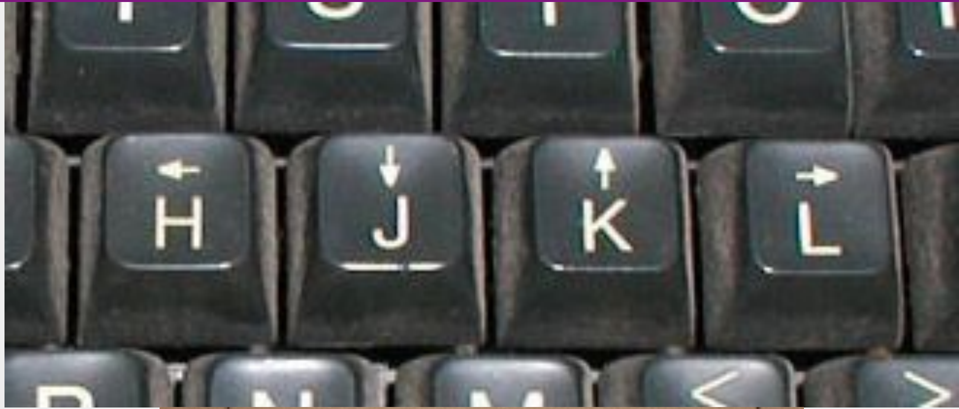
Vim Language: <Command><Number><Motion | Text Object>

Commands	Motions	Text Objects
Delete (`d`)	Chars Left (`h`)	Word (`w`)
Change (`c`)	Lines Down (`j`)	Sentence (`s`)
Yank (`y`)	Lines Up (`k`)	Paragraph (`p`)
Indent (`>`)	Chars Right (`l`)	In single-quotes (`i’`)
Unindent (`<`)	To end of line (`$`)	In double-quotes (`i”`)
Swap case (`~`)	To start of line (`^`)	In back-quotes (`` i` ``)
	To <char> (`t<char>`)	Around parens (`a)`)
	To start of word (`b`)	Around brackets (`a]`)
	To end of word (`e`)	Around curlies (`a}`)

Why Vim uses HJKL for navigation

This was the keyboard that vi was originally developed for, and the computer it was originally developed on. Since Vim is an improvement upon vi, Vim uses `hjkl` as well.

You *can* use the arrow keys, but most Vim users frown on it.



Buffers and Windows

Working with Multiple Files in Vim

What’s the Difference?

Buffer - The in-memory text of a loaded file
Window - Used for *viewing* the contents of a buffer

Common Buffer and Split commands

`:ls` - List all active buffers

`:bN` - Point the current window to the Nth buffer

`:bdN` - Delete the Nth buffer

`:bdM,N` - Delete all the buffers from M to N

`:sp <file>` - Split the current window horizontally, opening a new buffer for the provided file

`:vsp <file>` - Same as `:sp <file>`, but vertical

`:wincmd [h|j|k|l]` - Move the cursor between split windows

Common Miscellaneous Operations Translation Table

"I want to…"	"…, so in Vim normal mode I do…"
Paste a yanked snippet	`p`
Undo the last operation	`u`
Redo last undone operation	`<Ctrl+R>`
Save a buffer to disk	`:w`
Quit and close a window	`:q`
Quit window without saving	`:q!`
Quit out of all windows	`:qa`
Run a shell command	`:!<command>`

Sandbox Time

Use what you’ve learned so far!

Discussion - Initial thoughts

Awesome	Gross
●	●

Leaving vi Behind

Making Vim Your Own with Customization and Plugins

What are plugins (and why use them)?

- Pre-existing programs to add new features to Vim (so you don't have to!)
- Can be in vimscript or other languages (like Python)
- Use them to augment Vim with additional functionality to improve your flow

Managing Your Plugins

There are a lot of plugin managers out there, but my favorite (and the one I think is easiest to use) is called Pathogen. Setting up Pathogen is easy, but setting up all the plugins is a tedious process. So I scripted it. Please run `git clone` on this URL: https://github.com/AlexisGoodfellow/vim_wit_workshop.git

Then run this:

```
cd vim_wit_workshop && ./install.sh &
```

What the install script is doing

1. Ensuring /usr/local/bin is at front of `$PATH`
    a. Looks for user-installed executables in /usr/local/bin first
    b. Falls back to the operating system’s pre-loaded system executables in /usr/bin if not found in /usr/local/bin
2. Downloading vim and ctags with homebrew
3. Setting up and downloading pathogen
4. Changing directory to `~/.vim/bundle`
5. Downloading vim plugins by `git clone`-ing their repos

The `.vimrc` file

Whenever you open Vim, the `~/.vimrc` file is scanned first and all the settings in it are applied to your current session of Vim.

Thus, all customizations settings you want to apply can be put here and executed on entering a Vim session.

The `.vimrc` file provided in the git repo is a sample of my current working settings and options. Take anything you wish from it and put it in your own `~/.vimrc`!

Some Plugins and their Purposes

- NERDTree: For easy directory traversal and visualization
- NERDCommenter: Comment out many lines at a time
- fzf: Fuzzy file finder (works blazingly fast!)
- vim-dadbod: A SQL database client from within Vim
- YouCompleteMe: A multi-language autocompletion engine
- syntastic: Does syntax checking on every write to a buffer
- vim-gutentags: Manages tag files in the background
- simpleterm.vim: Manage terminals from inside Vim easily
- vim-test: Customizable multi-language unit test runner
- vim-fugitive: Run git commands without leaving Vim

Vim's Command Line Mode

- Entered via "`:`" in normal mode
- Both more powerful and more verbose than normal mode
- Can call Vimscript functions from the command-line mode
- Lots of plugins expose command-line mode only functions
- Use `<Leader>` key mappings
  - Avoid typing long function names
  - Improve flow

The <Leader> Key

`<Leader>` is a special key that Vim listens for in normal mode. Once found, Vim takes the characters you type after it and tries to match them to a keymapping in your `~/.vimrc`.

`<Leader>` defaults to “`/`”, but is often overridden to “`,`” instead.

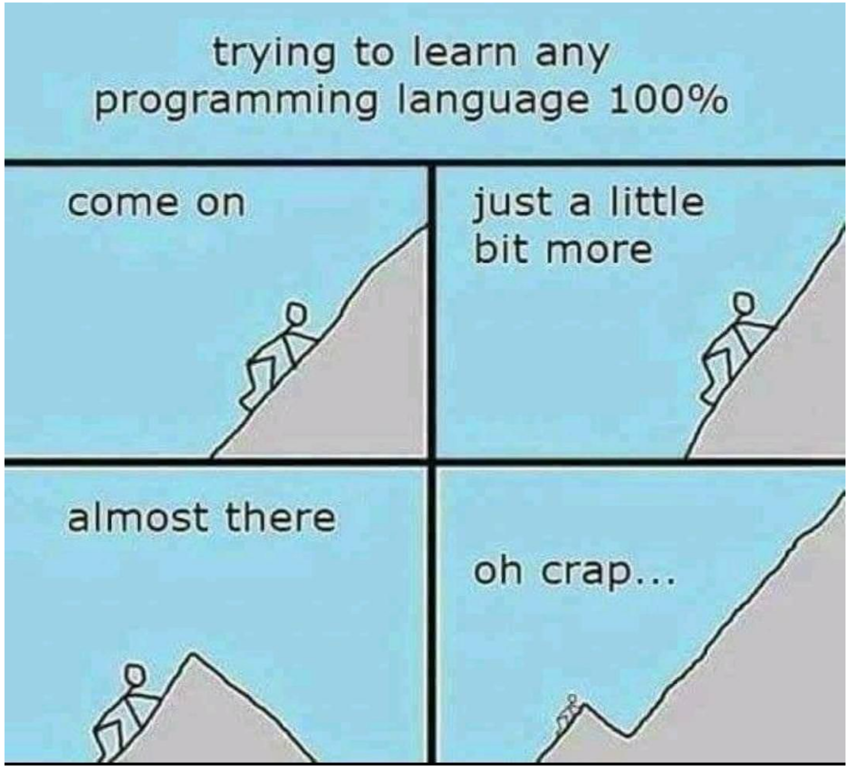
Example:

Mapping: `nmap <Leader>v :GitGutterNextHunk<CR>`

Usage: `,v`

Final Thoughts

- RTFM with `:help`
- I am still learning new things every day
- You will best learn Vim by using Vim for *everything*
- The learning curve is steep, but the rewards are plentiful
- Vimscript itself is ***terrible***



What questions do you have?

(I know you have some - I’ve been using Vim for 5 years and still learn new things on a near-daily basis)

Thank you very much!

(Now hack away on a project with what you’ve learned!)

wayfair®