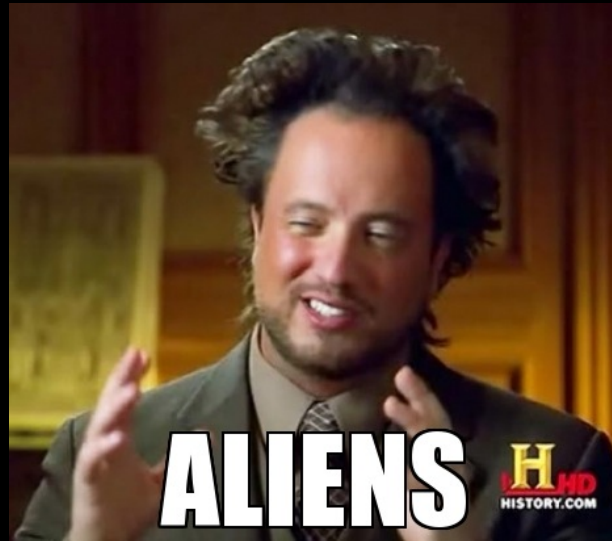


Shhhhhhhhhhhh

Matthew Swain
Spredfast

So apologies in advance to the Windows guys in the room. This talk is going to be pretty Unix focused, but a lot of the concepts can really be carried over if you're using a custom shell through Cygwin, powershell, etc. But for the most part, a vast majority of rails developers do their work on OSX and deploy their work on Linux. So tonight I'm going to talk a little about the history of Unix and why it's kinda fucked up, and introduce newcomers to some features they can use to make developing on a Unix system a more pleasant experience.

A Brief History of Unix



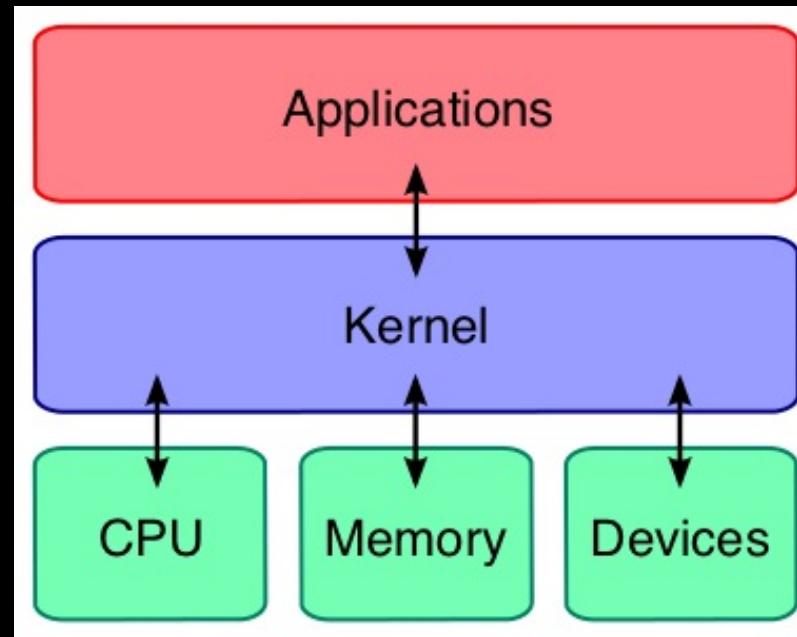
So that command prompt you look at every day has a very long history behind it, and that history is tied with the development of the c programming language and the Unix operating system itself. Linux and Mac OSX are actually flavors of Unix (though I'm sure some hardcore people would argue with me about both). A lot of the conventions we take for granted today come from decisions that were made in the late sixties and early seventies. And for better or for worse, we're kind of stuck with many of them.

A Brief History of Unix



Unix was originally developed by a really sharp guy named Ken Thompson in 1969. He worked Bell Labs, a division of the evil empire we all know and love called AT&T. It was developed in Assembler, but a short time later one of Thompson's colleagues, Dennis Ritchie, implemented the C programming language. Subsequently most of the Unix development occurred in C, and the language and the OS evolved around each other.

A Brief History of Unix



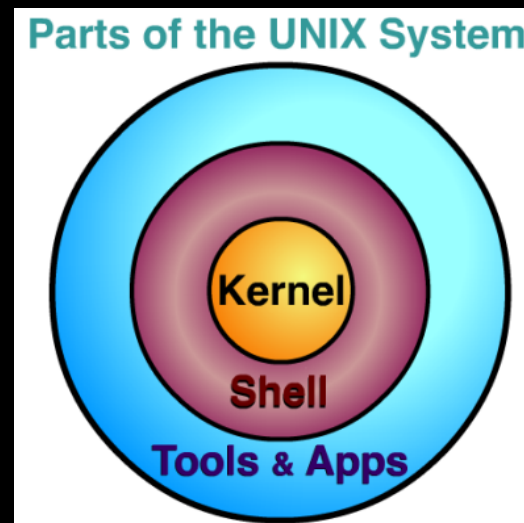
The core of the Unix operating system is called the kernel. It's the beast responsible for grabbing data off disks, loading programs into memory, switching between running programs, translating key presses on your keyboard into useable input, displaying things on your monitor, and more. It's an abstraction so you don't have to waste your time learning how to load the right instructions into the cpu to do these things. In a way, it makes using a computer a whole lot easier.

A Brief History of Unix



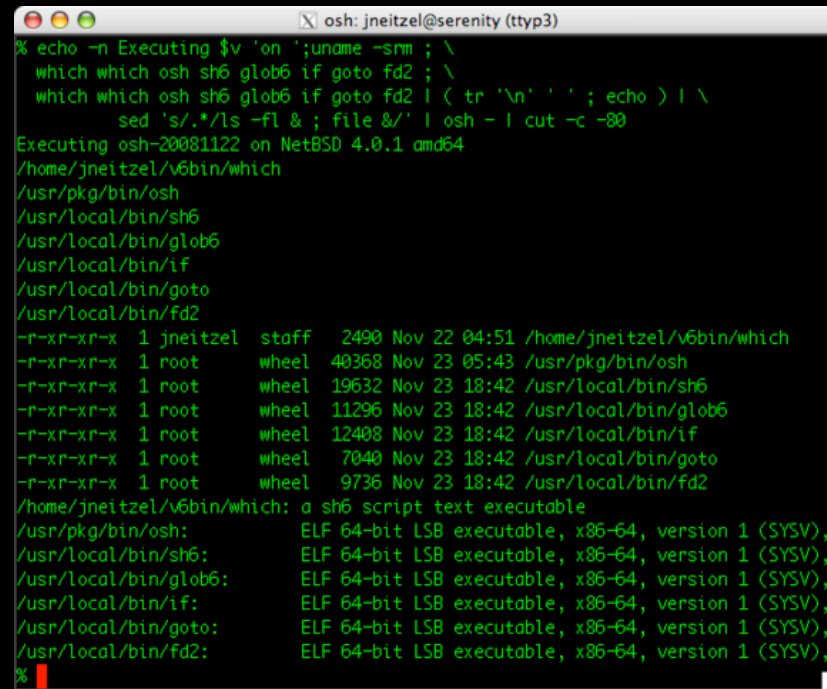
At least for hardcore computer scientists. However, for normal people like us system programming can be a tad impenetrable.

A Brief History of Unix



So in 1971, Thompson wrote a program called "sh". It was a very simple command interpreter that made working with Unix much easier. No longer did the stalwart graduate student have to write C code to copy a file from one place to another or print "hello world" to a terminal.

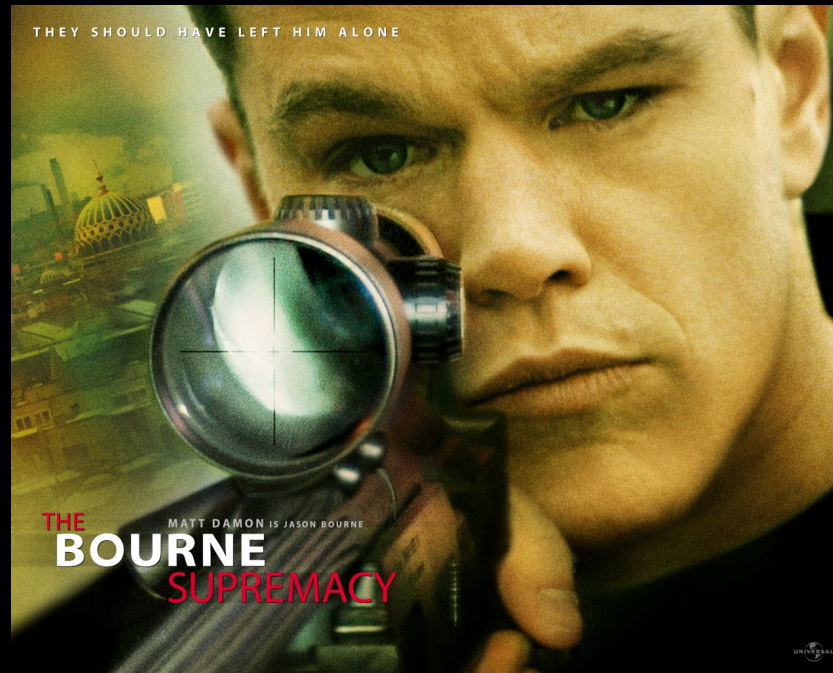
A Brief History of Unix



```
osh: jneitzel@serenity (tty3)
% echo -n Executing $v 'on ' ; uname -sm ; \
  which which osh sh6 glob6 if goto fd2 ; \
  which which osh sh6 glob6 if goto fd2 | ( tr '\n' ' ' ; echo ) | \
  sed 's/./ls -fl & ; file &/' | osh - | cut -c -80
Executing osh-20081122 on NetBSD 4.0.1 amd64
/home/jneitzel/v6bin/which
/usr/pkg/bin/osh
/usr/local/bin/sh6
/usr/local/bin/glob6
/usr/local/bin/if
/usr/local/bin/goto
/usr/local/bin/fd2
-r-xr-xr-x 1 jneitzel staff 2490 Nov 22 04:51 /home/jneitzel/v6bin/which
-r-xr-xr-x 1 root wheel 40368 Nov 23 05:43 /usr/pkg/bin/osh
-r-xr-xr-x 1 root wheel 19632 Nov 23 18:42 /usr/local/bin/sh6
-r-xr-xr-x 1 root wheel 11296 Nov 23 18:42 /usr/local/bin/glob6
-r-xr-xr-x 1 root wheel 12408 Nov 23 18:42 /usr/local/bin/if
-r-xr-xr-x 1 root wheel 7040 Nov 23 18:42 /usr/local/bin/goto
-r-xr-xr-x 1 root wheel 9736 Nov 23 18:42 /usr/local/bin/fd2
/home/jneitzel/v6bin/which: a sh6 script text executable
/usr/pkg/bin/osh: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
/usr/local/bin/sh6: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
/usr/local/bin/glob6: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
/usr/local/bin/if: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
/usr/local/bin/goto: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
/usr/local/bin/fd2: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
%
```

So Ken Thompson's shell was pretty nice, all things considered; but it didn't really allow people to write scripts or other fancy things. It was basically just "run this program," "pipe the output of this program to that program," "kill this program," and so on.

A Brief History of Unix



So glossing over a lot of history, in 1977 AT&T released the Bourne Shell with a ton of features for scripting and automation. It's named after Stephen Bourne who is shown here. This shell became a standard in a way, and a lot of its concepts are still in use today.

A Brief History of Unix



So then a bunch of stuff happened like New Coke, Crystal Pepsi, Ronald Reagan, and the cold war and stuff. And the legacy of the Bourne shell lived on as it waxed and waned over the years and was reborn again and again. And in fact, today the most popular shell in use (and the default shell on OSX), is called BASH. Which stands for the Bourne Again SHell.

A Brief History of Unix



So a bunch of stuff happened like New Coke, Crystal Pepsi, Ronald Reagan, and the cold war and stuff. And the legacy of the Bourne shell lived on as it waxed and waned over the years and was reborn again and again. And in fact, today the most popular shell in use (and the default shell on OSX), is called BASH. Which stands for the Bourne Again SHell.

A Brief History of Unix



So a bunch of stuff happened like New Coke, Crystal Pepsi, Ronald Reagan, and the cold war and stuff. And the legacy of the Bourne shell lived on as it waxed and waned over the years and was reborn again and again. And in fact, today the most popular shell in use (and the default shell on OSX), is called BASH. Which stands for the Bourne Again SHell.

A Brief History of Unix



So a bunch of stuff happened like New Coke, Crystal Pepsi, Ronald Reagan, and the cold war and stuff. And the legacy of the Bourne shell lived on as it waxed and waned over the years and was reborn again and again. And in fact, today the most popular shell in use (and the default shell on OSX), is called BASH. Which stands for the Bourne Again SHell.

A Brief History of Unix



So a bunch of stuff happened like New Coke, Crystal Pepsi, Ronald Reagan, and the cold war and stuff. And the legacy of the Bourne shell lived on as it waxed and waned over the years and was reborn again and again. And in fact, today the most popular shell in use (and the default shell on OSX), is called BASH. Which stands for the Bourne Again SHell.

A Brief History of Unix



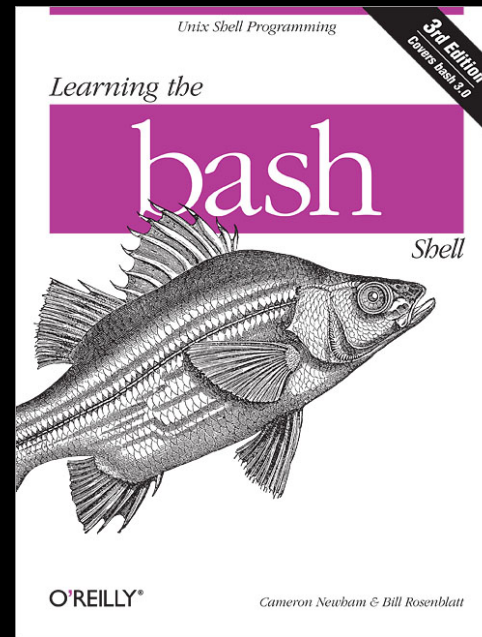
So a bunch of stuff happened like New Coke, Crystal Pepsi, Ronald Reagan, and the cold war and stuff. And the legacy of the Bourne shell lived on as it waxed and waned over the years and was reborn again and again. And in fact, today the most popular shell in use (and the default shell on OSX), is called BASH. Which stands for the Bourne Again SHell.

A Brief History of Unix



So a bunch of stuff happened like New Coke, Crystal Pepsi, Ronald Reagan, and the cold war and stuff. And the legacy of the Bourne shell lived on as it waxed and waned over the years and was reborn again and again. And in fact, today the most popular shell in use (and the default shell on OSX), is called BASH. Which stands for the Bourne Again SHell.

A Brief History of Unix



So a bunch of stuff happened like New Coke, Crystal Pepsi, Ronald Reagan, and the cold war and stuff. And the legacy of the Bourne shell lived on as it waxed and waned over the years and was reborn again and again. And in fact, today the most popular shell in use (and the default shell on OSX), is called BASH. Which stands for the Bourne Again SHell.

Show me the \$

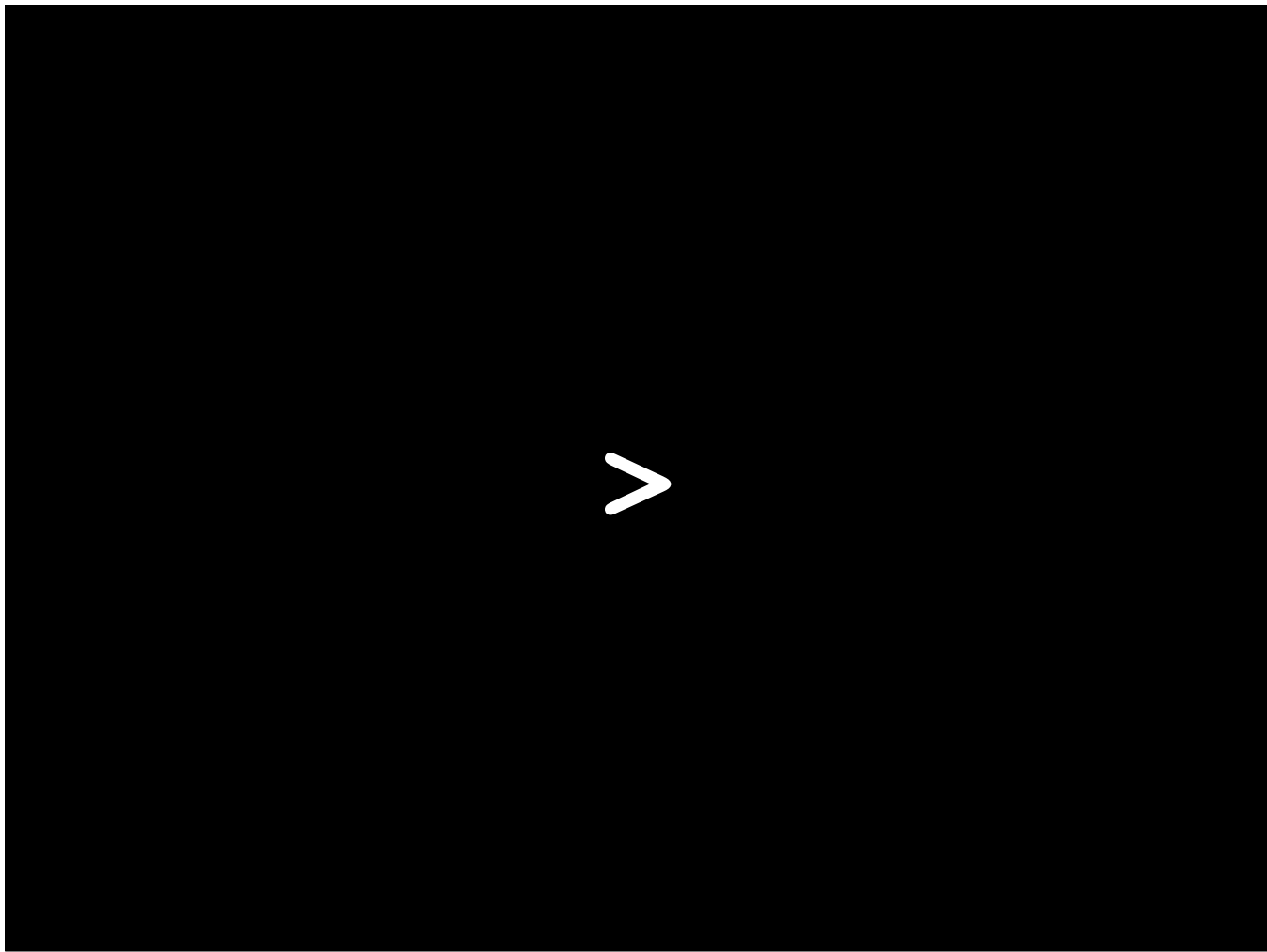
So lets start with input and output, without which we really wouldn't be able to do much of anything with our computers except generate heat and waste electricity.



So one of the core philosophies of Unix operating systems is everything can be thought of as a file. Your video card? that's a file. Hard drive? File. Network card? File. This is kind of an oversimplification, but essentially a lot of tasks reduce to simply moving data from one "file" to another.

```
$ cat dont-stop-believin.wav > /dev/dsp
```

For example, you can play sound files by simply dumping them into the soundcard file.



The real magic I want to show you from that example is the little ">" symbol.

I/O Redirection

>

This moving data from one file to another is called, in nerdy terms, "I/O Redirection".

I/O Redirection

```
$ cat dont-stop-believin.wav > /dev/dsp
```

“>” tells the shell to redirect the output of one command to a file. In this case, the sound card. In fact, if you just type "cat dont-stop-believin.wav" without that little > symbol, you're going to get a bunch of garbage on your screen.

I/O Redirection



So, what can you do with this and rails? Well, I'll start using some examples.

I/O Redirection

```
User.all.each { |u| puts u.login }
```

Familiar with rails runner? Let's say you have a script where you would like to print all the user login names on your system.

I/O Redirection

```
rails r /path/to/script.rb > logins.txt
```

Let's say you want to output that to a file? Simply redirect the output with >

I/O Redirection

```
rails r /path/to/script.rb >> logins.txt
```

If you'd like to append instead of overwriting the file, simply use >> in place of >

I/O Redirection

<

Similarly, we have a symbol "<" This one means "accept input from a file."

I/O Redirection

```
$ mysql blah_development < /rails/root/db/structure.sql
```

For example, you could do this to automate loading a mysql structure.sql file without loading the rails environment.

I/O Redirection

|

The coolest I/O redirection symbol is the pipe.

You can use this to connect the output of one program to the input of another.

I/O Redirection

```
rails r /path/to/script.rb | sort
```

For example, we can pipe the output of the script above through the unix command 'sort' to get a sorted list of user logins. The output of rails r is connected to the input of sort.

I/O Redirection

<http://tille.garrels.be/training/tldp/ch05.html>

That's about all I'll say about I/O redirection, but you can check out a fantastic guide [here](http://tille.garrels.be/training/tldp/ch05.html).

Moving Around

So you're typing a really long, really terrible command your devops guy told you to type. You're 4 lines into it and you see you made a typo in the second word. What do you do? Well, you can certainly hold down that arrow key for 40 seconds and wait until it's in the right place, or you could use some shortcuts to make your life BETTER

Moving Around

Ctrl + a Go to the beginning of the line (Home)
Ctrl + e Go to the End of the line (End)
Ctrl + p Previous command (Up arrow)
Ctrl + n Next command (Down arrow)
Alt + b Back (left) one word
Alt + f Forward (right) one word
Ctrl + x, Ctrl + e Edit the current line in your editor!

<http://ss64.com/bash/syntax-keyboard.html>

These are really all about muscle memory, so I'll list a few of them here and leave the rest for you to look up. But getting these under your fingers will cut down on tons of typing and save you tons of time. I especially love ctrl + x, ctrl + e. It'll open up the current command line in your editor. Pretty rad sauce for working with huge commands with heredocs and such. ^^

Useful Builtins

Navigation is also a lot easier using some of the shell's builtins. These are shortcuts that can help you get around your filesystem without having to type too much.

Useful Builtins

```
$ cd ~ # moves you to your home directory
$ cd - # moves you to the last directory
$ pwd # shows you what directory you are in

$ pushd /directory # push /directory onto the dir stack
$ pwd
/directory
$ pushd /rails
$ pwd
/rails
$ dirs # prints the directory stack
/rails /directory ~
$ popd # change to the last dir on the dir stack
$ pwd
/directory
```

Here are a few examples ^^

Tab Completion

For getting around in Bash, the tab key really is your best friend. Learning how to use it effectively will also save you a ton of typing.

Tab Completion

```
~/rails$ bun<tab><tab>
bundle          bundle_image  bundler         bunzip2

~/rails$ cd t<tab><tab>
test/          tmp/
~/rails$ cd te<tab><enter>
~/rails/test $
```

In Bash you can type part of a command and hit tab to have it automatically complete it for you. This works on both command names, directories, files, etc. In fact, if there is any ambiguity about the command you're typing, you can hit tab twice and display a list of everything bash thinks it could be. ^^

Bash History

So have you ever been in a situation where you just can't remember a command you entered weeks ago? Just can't remember those arguments for the "git" command? Well bash and its ilk have you covered. There's a very extensive history management feature in bash that remembers every command you've typed, up to a certain configurable limit.

Bash History

```
$ history
```

```
23  bundle exec rake db:create  
24  bundle exec rake db:schema:load  
25  bundle exec rake db:migrate  
26  ./bin/rails c
```

```
$ !26
```

So how do you tap into this stored history? Well one of the easiest ways is to simply type the command "history" ^^

If you want to execute one of the commands in the list, simply type !<number of the command> ^^

Bash History

!!

!! will execute the last command entered.

Bash History

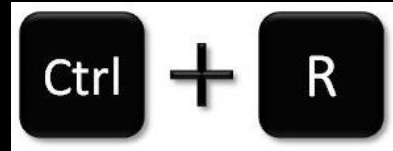
```
$ history |grep -i rails
```

```
26  ./bin/rails c
```

Naturally, “history,” like any other Unix command, can be piped to any other program as we discussed when talking about I/O redirection. ^^

Grep, in case you don't know, is a common unix program for filtering text, and “-i” tells it to be case insensitive.

Bash History



```
$ (reverse-i-search)`rai': bundle exec rails c
```

Bash even gives you tools to search through your history. All you have to do is type ctrl-r, then type a fragment of a command you previously entered. You can then hit ctrl-r multiple times until you find the command you're looking for. Once you find the right one, you can press enter to run it. If you want to edit the command you've found instead of running it, you can hit ctrl-j instead of enter.

Bash History

http://www.talug.org/events/20030709/cmdline_history.html

Check out this site for a more thorough tutorial.

Aliases

Let's say you have a command that's really long and shitty that you find yourself entering so often that even searching for it with "ctrl-r" drives you nuts. Enter aliases. Aliases are simply shortcuts that you can define to make your life better.

Aliases

```
$ alias rct="bundle exec rails console testing"  
$ rct  
Loading testing environment (Rails 3.2.18)  
[1] pry(main)>
```

For example, you might enter "bundle exec rails console testing" a few thousand times per day. Wouldn't it be nice if you could just type something like "rct"? ^^

Aliases

```
alias g="git"  
alias l="ls"  
alias ll="ls -alhtr"  
alias gb="git branch"  
alias gc="git commit"  
alias gca="git commit -a"
```

Here a few examples of aliases I use every day, and my fingers are thankful for it.

Aliases

<https://www.digitalocean.com/community/tutorials/an-introduction-to-useful-bash-aliases-and-functions>

Config Files

Now that I've mentioned aliases, I should probably bring up bash's configuration files. Using these files, you can store all your aliases, change your bash history length (I set mine to 500,000), change your font colors, and so on.

Config Files

OSX: `/Users/<your user name>/.bash_profile`

Linux: `/home/<your user name>/.bashrc`

http://www.joshstaiger.org/archives/2005/07/bash_profile_vs.html

If you use Terminal.app on OSX, you're going to want to use the hidden file in your home directory `"/Users/<your user name>/.bash_profile"`. On ubuntu you're probably going to want to use `.bashrc` instead. There's a logic to this that is steeped in Unix history, so I'll direct you to some further reading to check out why.

Config Files

```
# ~/.bashrc: executed by bash(1) for non-login shells
# see /usr/share/doc/bash/examples/startup-files
# for examples

export HISTFILESIZE=100000
export HISTSIZE=100000
alias g="git"
alias l="ls"
alias ll="ls -alhtr"
alias gb="git branch"
alias gc="git commit"
alias gca="git commit -a"
```

Every time you launch a new shell or login to your computer, the commands in your config files will run. Notice the first couple non commented lines. I mentioned before how the history length is configurable, and these are the variables you'll need to set to change them. ^^



ZSH

Zsh is like bash on crack, it comes installed on OSX, and it has an amazing community supported set of tools to make your shell experience totally rock.

ZSH



<https://github.com/robbyrussell/oh-my-zsh>

Those community plugins are collectively known as “oh my zsh”

Installation instructions are on the Github page here.

ZSH

```
$ git --<tab><tab>
--bare                -- use $PWD as repository
--exec-path           -- path containing core git-programs
--git-dir             -- path to repository
--help               -- display help message
--html-path           -- display path to HTML documentation and exit
--no-pager            -- do not pipe git output into a pager
--no-replace-objects -- do not use replacement refs to replace git
objects
--paginate           -- pipe output into $PAGER
--version            -- display version information
--work-tree          -- path to working tree
```

ZSH is a superset of bash, and comes with even more goodies to rock your world. Coupled with the oh-my-zsh plugins, it'll show you what git branch you're in, make your command prompt compact and pretty, spell check your commands, and even tab complete options. Here's a great example. You type git —<tab><tab> and it gives you inline documentation for all the options!

ZSH

```
$ → ~ exti  
zsh: correct 'exti' to 'exit' [nyae]?
```

Here is an example of the spell correction at work. If you type exti instead of exit, zsh will notice and ask you if you screwed up.

ZSH

```
→ chef git:(master) cd -<tab>  
1 -- /Users/mswain/code/tweetriver  
2 -- /Users/mswain  
→ chef git:(master) cd -1  
~/code/tweetriver  
→ tweetriver git:(forking-analyzers) X
```

Even `cd "-"` is enhanced. type `cd -<tab>` and it will give you a list of directories you've been in before that you can select by number. ^^



Questions?

So that concludes my brief introduction to the Unix shell. This is really the tip of the iceberg, so if you have any questions ask away, or come find me @ the meetup!

References

1. Kerrisk, Michael. The Linux Programming Interface. San Francisco: No Starch Press, 2010.
2. Störimer, Jesse. Working with Unix Processes.
3. Stephenson, Neal. In the Beginning There Was the Command Line. New York: William Morrow Paperback. 1999.
4. Levy, Stephen. Hackers - Heroes of the Computer Revolution. Sebastopol: O'Reilly Media, 2010.
5. The Linux Documentation Project. <http://www.tldp.org>