

Command Line



Might start off with <http://geektyper.com/?ref=producthunt> on the screen :)

Prerequisites

A Terminal installed and customized to your liking.

Today we're going to learn more about how that Terminal app on your Mac works. This should be a Unix-based terminal, not the Windows Command Prompt.

If you're using Windows PowerShell, you should be able to use the same commands, but the output may look different.

What you'll learn

How to really use that Terminal app we've been talking so much about.

Why Command Line?



App Store



Automator



Calculator



Calendar



Chess



Contacts



Dashboard



Dictionary

When you want to open an app, you double-click it.

Behind the scenes, your operating system translates that click into code. In particular, a single line of code is usually all it takes to launch an app.

Why Command Line?

Double-clicking here:



Runs this:

```
/Applications/Safari.app/Contents/MacOS/Safari
```

For example, on my laptop, when I click the Safari icon, it actually runs this line of code in the Terminal.

The icons and windows we see in the Mac OSX operating system (and any OS for that matter) are all GUIs (graphical user interfaces) for lines of code. They allow me to interact with my computer visually. But underneath those visuals are magic words: commands.

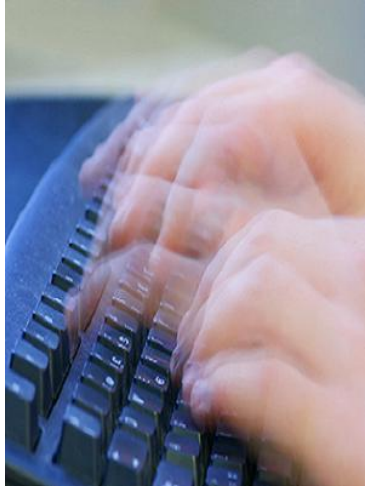
Why Command Line?

Name ▲	Date Modified	Size	Kind
▶ Desktop	Today, 8:11 PM	--	Folder
▶ Documents	Apr 3, 2014, 2:36 PM	--	Folder
▶ Downloads	Today, 11:29 AM	--	Folder
▶ Dropbox	Apr 1, 2014, 3:00 PM	--	Folder
▶ Google Drive	Apr 15, 2014, 11:34 AM	--	Folder
▶ Movies	Mar 29, 2014, 2:52 PM	--	Folder
▶ Music	Mar 29, 2014, 2:52 PM	--	Folder
▶ Pictures	Apr 14, 2014, 3:12 PM	--	Folder
▶ Public	Mar 29, 2014, 2:52 PM	--	Folder

Commands aren't just used to launch applications. They can also create, move, rename, copy, and delete files and folders.

So all the things you'd do in the Mac Finder or Windows File Explorer, you can also do from the command line.

Why Command Line?



So, if we have these beautiful and convenient GUIs, why bother learning the underlying command language? I'd rather click on pictures than type long lines of text.

Two reasons.

First, getting good with the command line helps you work faster. It takes time to move your hands from the keyboard to the mousepad and back. It's not a lot of time, but it adds up.

Over time, you'll save precious seconds doing some of the most common (and some uncommon) tasks. Instead of popping up windows and scrolling around looking for some little picture of a program, you'll just type its name.

The Terminal even allows you run hidden programs - ones that don't even have little icons to click on. The command line unlocks your computer's full potential.

Why Command Line?



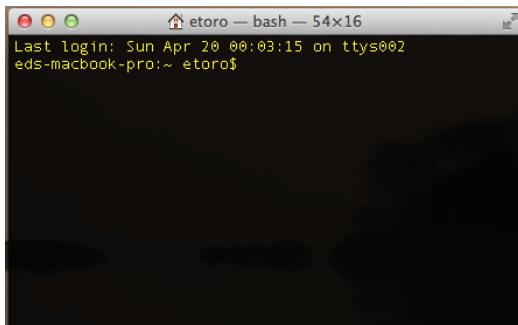
The second reason why the command line is faster is hinted at by this photo of Google's North Carolina datacenter. These stacks of computers run Google's various web properties. Does anyone notice what's missing?

There are no monitors. There are no mice. There are no keyboards. There might be a couple full workstations in this room. There might be some peripherals on little carts that you can roll around and plug into one of these boxes.

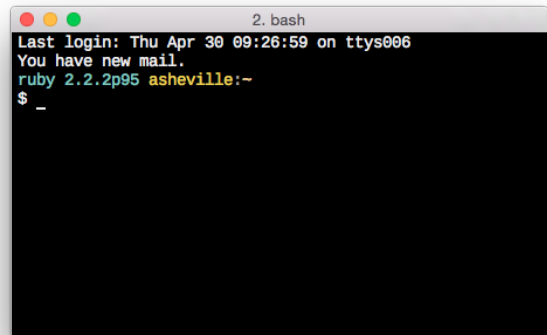
But, for the most part, each of these computers is managed **remotely**. Someone connects to them via a text-only terminal and runs commands. There is no GUI because there's no place to show a GUI. To interact with these computers, there's only the command line.

Text-only terminals are how most of the servers that run the web are managed. You need to know the Command Line to configure and control the infrastructure that runs your website.

Command Line



```
etoro — bash — 54x16
Last login: Sun Apr 20 00:03:15 on ttys002
eds-macbook-pro:~ etoro$
```



```
2. bash
Last login: Thu Apr 30 09:26:59 on ttys006
You have new mail.
ruby 2.2.2p95 asheville:~
$ _
```

Let's get started.

On the right is more or less what the Terminal app looks like on my laptop. On the right is what the iTerm 2 app looks like. They're pretty much the same.

You should see a blinking cursor preceded by some text. In Mac Terminal, it's the name of your computer and your current directory.

Hit enter a few times. Notice that the preceding text doesn't go away. Every command you type comes after the dollar sign, starting at the blinking cursor.

Hello World!

```
echo 'Hello world!'
```

Like with any new programming language, let's start at the beginning with our “Hello world” program. Instead of `print` or `puts`, the command line uses `echo`.

<tab> Autocomplete

ech<tab>

becomes

echo

One of the main reasons that the Terminal can be so much faster than point-and-clicking is that it will automatically complete your command. So if I type e-c-h and press the tab key, the Terminal application will automatically type the o and a space.

<tab><tab> Hints

e<tab><tab>

The autocomplete only works if the terminal can guess what you mean. If I just type e, there are many commands that start with the letter “e”. Pressing the tab key twice, the Terminal will list every command I could possibly mean. I can keep typing additional characters and auto-completing to get more specific results.

<tab><tab> Hints

```
etoro@eds-macbook-pro:~$
```

```
Display all 1521 possibilities? (y or n)
```

If I don't type anything at all and hit tab-tab, I haven't given the Terminal any hints. So it will recommend *every* possible command to me. Because the list is so long, it'll ask me up front if I'm sure I want to see *all* the suggestions. On my Mac, there are over 1500 commands available. This is like typing `methods` in Ruby, only it actually works because it doesn't hide anything from you.

As you can see, there are lots and lots of commands. We're not going to learn all of them. I don't even know what they all do. Over time, with experience, you'll learn more and more. But today we're only going to focus on some important ones.

Note: If you see someone constantly hitting the tab key as they type, chances are that person is using the command line. Auto-complete is very addictive. Your pinky finger is going to get some exercise.

man

```
man echo
```

```
man man
```

```
man top
```

```
man purge
```

As you explore the command line by smashing the tab key again and again, the first command you should learn is `man`, which stands for *manual*. If you type `man` before any command, the command line will show you the documentation for that command. Much learning will come from here.

Like `p` in Ruby, the `man` command is also the butt of jokes because of the funny looking lines of code it can produce - like these, for example. [slide]

Hit the 'q' key to exit the manual when you're done reading it.

[might also cover ``man -f``/'apropos' at this point?]

Command Syntax

SYNOPSIS

`echo [-n] [string ...]`

SYNOPSIS

`man [-acdfFhkKtwW] [--path] [-m system] [-p string] [-C config_file]
[-M pathlist] [-P pager] [-B browser] [-H htmlpager] [-S section_list]
[section] name ...`

If you read enough *manpages* (yes, that's actually what they're called), you'll notice that all of them contain a SYNOPSIS section near the top that quickly demonstrates how the command should be used.

The first word is the command name itself. The command line is all commands. Unlike Ruby, there's no object. The object receiving your command is the computer itself.

Next, after the command name, comes a list of *flags* preceded by one (-) or two dashes (--). Typically a single-dash precedes a single-letter flag (-n) and a double-dash precedes a word flag (--path). These flags change the way the command works (kind of like a Ruby method options hash).

All the words at the end are the *arguments*.

`echo`, for example, takes one or more string arguments, as indicated by the word *string* followed by three dots.

`man` takes two arguments, a *section* and one or more command *names*.

Why are some arguments in brackets and some not? Arguments in brackets are optional. So, for example, you can just type `echo` and it'll work. Your computer will just print a blank line. Everything else is listed in square-brackets, so it's optional.

The `man` command, on the other hand, *requires* at least one *name* argument (the

name of a command). The *name* argument is not listed in square brackets, so it is **required**. Everything else is listed in square brackets, so they're optional.

Command Syntax

```
echo
echo 'Hello world!'
echo 'Hello' 'World!'
echo -n 'Hello world!'
echo -n 'Hello' 'world!'
```

According to the man page, here are 5 ways to use the echo command:

- by itself
- with a string
- with a list of strings (separated by spaces)
- with the n flag, which **doesn't** put a newline character at the end of the output
- and altogether, with both the n flag and multiple arguments

pwd

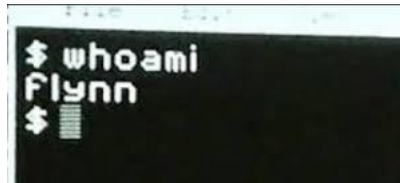
```
$ pwd  
/Users/etoro
```

Now let's get to some real commands. `pwd` stands for “print working directory”. It's like asking “where am I?”.

[DD: It is **not** a shortcut for ``passwd``.]

By default the Mac Terminal drops you into your home directory: `/Users/your-username`.

whoami



The aptly named `whoami` command tells you who you're currently logged in as. On Mac, it's your username.

This command, among others, makes a cameo in the movie "Tron Legacy". It's how Sam finds out his dad was the last one logged-in to an old dusty terminal he finds in a back room of Flynn's Arcade.

mkdir

```
mkdir
```

```
mkdir -p code/1/2/3
```

The `mkdir` command “makes a directory”. The `-p` flag (for pathname) allows you to make multiple directories at once (e.g. make the entire path).

cd

```
etoro@eds-macbook-pro:~$ cd code/
```

```
etoro@eds-macbook-pro:~/code$ pwd  
/Users/etoro/code
```

The `cd` command doesn't play the cd in your cd player. It actually changes your current directory. Let's change your current directory to the code directory you just created in the last slide. Now when you use the `pwd` command, you should see that your Terminal is now located in a directory named "code". The prompt (the stuff before the `$`) should have updated as well.

rmkdir

```
rmkdir 1
```

```
rmkdir -p 1/2/3
```



`rmkdir` removes a directory. In the Terminal, remove means delete. But when you delete from the Terminal, you delete for good. There's no recycle bin. It's just gone. Thus the saying that `rm`, like a diamond, is forever.

`rmkdir 1` doesn't work because the `1` directory (the directory named `1`) isn't empty. You'll learn to love that warning. It'll save you from making big mistakes (Why? Because "rm is forever"). If you'd like to delete multiple (empty) directories at once, you can use the `-p` flag (just like `mkdir`).

touch and ls

```
touch some_code  
touch some_more_code
```

```
ls  
ls -a -l  
ls -al
```

Let's quickly create some files. The quickest way to create a file is the touch command. The touch command is like a test. It tests whether or not you're allowed to create files. It does so by creating an empty file. Think of it like you're touching the directory to see if it works. (And, btw, yes, "man touch" is a valid command.)

If you'd like to see the files you just created, use the ls command. ls stands for list, as in "list the files in the current directory". If you check the manpage for ls, you'll see it has a ton of flag options. It looks like almost every letter of the alphabet does something. But the most common flags are a (for all) and l (for long). Note that you can combine single-letter flags.

ls -al

```
total 0
drwxr-xr-x   4 etoro  staff  136 Apr 22 01:59 .
drwxr-xr-x+ 23 etoro  staff  782 Apr 22 01:41 ..
-rw-r--r--   1 etoro  staff    0 Apr 22 01:59 some_code
-rw-r--r--   1 etoro  staff    0 Apr 22 01:59
some_more_code
```

You'll notice that the output of `ls -al` shows a bunch of information about the files we just created.

The first letter is the file type. "d" means directory and "-" means file.

The next 9 letters represent the file permissions. The permissions are three sets of three letters. The first three represent the permissions of the person who "owns" the file. The next three represent the permissions of the family or "group" of the person who owns the file. And the last three represent everyone else.

In each triplet, r stands for read, w for write, and x for execute.

So, for both of our new files, `some_code` and `some_more_code`, both the owner, the group, and everyone else can read the file, but only the owner can write the file. And no one can execute the file.

Who's the owner of the file? That's the "etoro" part of the line. The group? That's the staff. "Staff" is the default group on my Mac.

The next number, the 0, is the file size. Like I said, `touch` creates empty files.

The date and time that follow are the times that the file was last modified.

The last item in the row is the filename.

. and ..

```
drwxr-xr-x  4 etoro  staff  136 Apr 22 01:59 .  
drwxr-xr-x+ 23 etoro  staff  782 Apr 22 01:41 ..
```

```
cd .  
cd ..  
cd -  
cd ~ # same as just typing cd  
cd -
```

You may have noticed that `ls -al` returns two extra lines, one for dot and one for double-dot. The `-a` flag caused those mysterious directories to appear. Those are shorthand symbols for two very important directories.

dot is the current directory. That's telling you information about where you are at the moment. If you "cd" to dot, you're just changing to the current directory, which does nothing.

dot-dot is the parent directory. The `ls` output is telling you information about the directory above the one you're currently at. If you "cd ..", you'll be moving up one directory.

There are other directory shortcuts. If you cd to dash, that'll take you back to wherever you came from.

If you cd to the tilde (which is the squiggly line at the top-left corner of your keyboard) that sends you to your "home" directory. Just typing `cd` will do the same thing. Your "home" is where you start when you first open the Terminal app.

Let's cd to - to get back to where we were. Use `pwd` to confirm that you're back at the code directory before we move on.

cp

```
cp some_code some_code_backup
```

```
mkdir a
```

```
cp -r a b
```

The `cp` command is used to copy files or directories. The contents of the first file are copied into the second. If the second file doesn't exist, it's created. If it does exist, it's overwritten. Like `rm`, overwriting a file is forever.

There's no `cpdir` command, so `cp` works for both files and directories. For directories, use the `-r` flag, which stands for "recursive", which is just a fancy word for saying that you want the copy to work on all subdirectories.

cp

```
mkdir -p 1/2/3
```

```
mkdir without_slash  
cp -r 1 without_slash
```

```
mkdir with_slash  
cp -r 1/ with_slash
```

For directories, cp works differently depending on whether or not you include a slash at the end. Without it, it copies the entire directory. With it, it just copies the contents of the directory.

ls -R vs. find .

```
ls -R
```

```
find .
```

In that last slide, it was hard to confirm that the `cp` was doing what I told you it was doing. You have to `ls` a bunch of directories.

As a shortcut, you can use the `-R` flag, which will run `ls` on the current directory and all subdirectories.

Even better than `ls -R`, however, is the “`find .`” command, which just lists the file and directory names of everything in and below your current directory.

`find` has a bunch of useful options you can use to help you find a file, so check the manual.

mv

```
mv some_more_code some_other_code
```

```
mv 1 a
```

mv stands for “move”. Move is just like copy, except the source file is deleted. Think of it like renaming.

mv works for directories as well, but doesn’t require the -r flag.

However, the end-slash rule does apply. Without it, you’re moving a file into a directory. With it, you’re renaming a directory.

rm

```
rm some_code_backup
```

```
rm a
```

```
rm -r a
```



rm stands for “remove” and it removes (or deletes) a file.

You can also remove a directory, but, like cp, you need to use the -r flag. rm is forever, so be very careful with it. A single rm is dangerous. An rm with the -r flag is even more so.

rm doubts

- Try `rmdir` before `rm -r`
- Are you sure you don't want to `cp` a backup first?
- Take a breath before using `rm`.
- Take a deep breath before using `rm -r`.

Because `rm` is forever, you always want to be careful with it. Here's some advice.

DD: ``rm -ri`` is a good alternative. What about ``alias rm=rm -i`` ?

Separation of Concerns

```
copy --delete_source source_file destination_file
```

I've shown you how to create, copy, rename, delete, and list files and directories. Each of these commands is a single program. The practice of creating small programs that do one simple thing well is good design. The design principle is called "Separation of Concerns". Each program is concerned with only one thing. Adding more things increases the likelihood of something going wrong.

You could imagine, for example, that move and copy could be the same thing. Move could have just been a special flag attached to copy that instructed it to delete the source file after the copy was complete. But this is actually bad design. A bug in the copy code could break the moving code, and vice versa. When combined, these two commands become unnecessarily dependent on one another, which is bad.

The Terminal's practice of breaking up an everything your computer can do into lots and lots of small, discrete commands has worked out very well for Unix-based systems (like Mac and Linux) and has inspired many other software designs.

> and cat

```
echo 'Hello world!' > hello_world.txt
```

```
cat hello_world.txt
```

```
Hello world!
```

Eventually, however, you'll want to have some commands talk to other commands. Instead of creating new programs that combine multiple commands, the Terminal app itself allows you to chain multiple commands together, similar to the way you'd chain methods together in Ruby.

For example, let's say you want to take the output of our "Hello World" command and store it in a file. The greater-than sign allows you to take the output of one program and dump it into a file. To see the contents of the file, use the "cat" command followed by the filename.

>> and cat

```
echo 'Hello world!' > hello_world.txt  
echo 'Hello world!' >> hello_world.txt
```

```
cat hello_world.txt  
Hello world!  
Hello world!
```

1 greater-than symbol will overwrite the file. 2 greater-than symbols will append (add to the end) of a file.

>> and cat

```
find . >> hello_world.txt
```

```
cat hello_world.txt
```

```
Hello world!
```

```
Hello world!
```

```
.
```

```
./hello_world.txt
```

Any command that produces output can have that output sent to a file.

less

```
less hello_world.txt
```

```
man less
```

If a file is very large, using the cat command might take too long. You'd have to wait for the entire file to scroll across the screen. Less is an alternative to cat that allows you to scroll through a long file one page at a time.

The man command uses less instead of cat to display the manual for a command. Within a less'd document, you can use the up-arrow or down-arrow to move up or down 1 line. The space bar goes 1 page down. The b key goes one page up. Hitting g takes you to the top. Capital G takes you to the bottom. But the most useful key of all in less is the forward slash (/). It allows you to search the file for some text.

[demo]

So, for example, looking at the synopsis for less, you might be wondering what -q does. You can type forward slash, then -q, then enter. Less will highlight all occurrences of -q for you, and skip down to the first one. Hit the n key to move to the next occurrence and capital-N to move to the previous one.

Type q to quit.

Ctrl+c

yes

[Ctrl] + c

If you type yes, you'll see an infinite cascade of the letter-y. That's a weird command indeed. To escape it's clutches, hold the control key and type c.

Ctrl+c is your escape hatch for when things go wrong.

| (the pipe), head, and tail

```
cat hello_world.txt | head
```

```
cat hello_world.txt | tail
```

```
yes | head
```

The greater-than symbols allows a command to talk to a file. The pipe, located on the far right side of your keyboard above the backslash, allows two commands to talk to each other.

For example, the cat command will output the full contents of a file. If you pipe that to head, head will only output the first 10 lines and ignore the rest.

Tail does the opposite. It outputs the last 10 lines and ignores the rest.

If you want to get that weird “yes” command under control, you can pipe it to head. That will only output 10 y’s and then quit.

head and tail

```
head hello_world.txt
```

```
tail hello_world.txt
```

Head and tail also work on their own if you just give them the name of a file.

head and tail

```
cat hello_world.txt | head  
head < hello_world.txt
```

```
cat hello_world.txt | tail  
tail < hello_world.txt
```

Using “cat” to output the contents of a file to a command is a common pattern, so Terminal has a shortcut. If you flip the greater-than to a less-than, the data flows in the opposite direction, from the file to the command.

So these pairs of commands are equivalent.

cat to head, or file into head.
cat to tail, or file into tail



*

```
ls
```

```
ls some*
```

```
ls *code
```

```
ls *s*
```

The asterisk symbol is called a wildcard. For commands that accept a filename arguments, it allows you to pass multiple arguments with a single word.

For example, if I use `ls`, I should see the `some_code` and `some_other_code` files that I touched earlier, along with everything else. If I just want to see the files that start with the word “some”, I can start typing “some”, then end with an asterisk. That will list all the files starting with some.

If I put the asterisk in the beginning, like in this example, I only get files that end with the word “code”.

If I put asterisks on both sides, I get files or directories that contain the letter “s” anywhere in their name.

grep

```
find . | grep slash
```

```
grep 'Hello' *
```

```
grep -r 'Hello' .
```

grep is a very handy command. It stands for “globally search a regular expression and print”. Think of it as a search tool.

If you pipe to grep, it'll filter out all the lines that don't contain the argument. In this first command, I'm using grep to just give me the list of files that contain the word “slash” in their name. You can do this same thing just using find, but this is sometimes quicker to type.

On it's own, grep accepts two arguments. The first is the pattern, and the second is a list of files to search. In this format, grep will search inside each file for text that matches the pattern.

If you'd like grep to search inside both files and directories, use the -r (recursive) flag, and pass it a directory name.

set variables

```
HELLO="World"  
echo $HELLO  
  
set  
set | grep HELLO
```

The Terminal supports variables. By convention, Terminal variables are in ALL-CAPS. When you want to refer to a variable you've defined, prefix it with a dollar sign.

You will hardly ever need to define Terminal variables. Any time you find the need to, just use Ruby instead. However, it is important to realize that there are a ton of variables that come pre-defined for you. You can see the list if you use the "set" command.

env variables

```
HELLO="World"  
export HELLO  
echo $HELLO
```

```
env  
env | grep HELLO  
set | grep HELLO
```

By default, variables are private. They can only be used in the Terminal. When you export a variable, it becomes public. That means other programs can access it.

The “env” command outputs a list of public variables. Just setting a variable adds it to set, but not to env. Exporting a variable adds it to both.

env variables

```
export HELLO="World"
```

```
unset HELLO
```

You'll commonly see the export and the set happen on a single line like this. This sets the variable `*and*` makes it public.

To delete a variable, you can unset it. That'll remove it from set and from env.

Variables won't survive when you close and re-open the Terminal app either.

exit

exit
logout

[ctrl] + [d]

Speaking of closing and re-opening the Terminal, you can exit the Terminal using the commands `exit` or `logout`.

Holding the [control] key and pressing the d key is a shortcut for closing the Terminal as well.

permissions

chown

chgrp

chmod

All the permission settings you see when you “ls -al” can be modified using the chown, chgrp, and chmod commands. We’ll talk about this more some other day.

ruby

```
echo puts \"Hello world\\!\\\" | ruby
```

```
ruby -e "puts \"Hello world\\!\\\""
```

```
ruby hello_world.rb
```

Ruby itself is a command (as long as you have Ruby installed, which should be all of you).

You can pipe Ruby code into the ruby command.

The -e flag lets you pass a line of Ruby code as an argument.

But the most common usage of the ruby command will be to run our programs. You just pass the name of the Ruby file as an argument.

version convention

```
ruby -v
```

```
ruby --version
```

Ruby, like many commands that don't automatically come with your computer's operating system, uses some conventional flags.

So, for example, if you'd like to get the version of something you've installed, you can typically use the `-v` or `--version` flags.

help convention

```
ruby -h
```

```
ruby --help
```

```
ruby -?
```

```
man -?
```

Ruby also uses the help convention. Typically passing `-h` or `--help` will display a helpful, shorter version of the man page.

Less common is the `-?` flag, which some programs support, but Ruby doesn't.

Some commands, like `man`, will display the help if you try to use a flag it doesn't recognize. But I don't recommend depending on that. Typing random commands you don't understand into the Terminal is a good way to get yourself into trouble. If you aren't sure what something does, try passing the help flags first.

and so much more!

- permissions
- booleans
- creating your own commands
- text editors
- job control
- monitoring

There are a bunch of other commands available in the terminal that we won't talk about today. Advanced terminal usage crosses over into a whole other branch of software called "system administration". Those are the guys that, for example, set up the physical hardware in the datacenter or configure your servers for you.

To use a metaphor, we just want you to be able to drive the car, not repair it. Over time you'll pick up little tricks here and there. It's going to be more memorable to bring up those tricks when something breaks rather than try to jam them all into your head at once.

Terminal, a.k.a

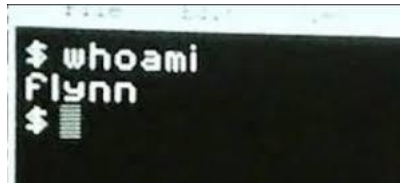
- Terminal, iTerm2
- terminal
- tty
- console
- shell
- command line
- BASH (Bourne-Again SHell)

To end this lecture, let's clear up some terminology. Terminal (capital-T) is the application that runs a terminal (lowercase t) on Mac. That application may be called different things on different operating systems. An alternative to capital-T Terminal on the Mac is iTerm2, which you can download and use for free if you'd like.

terminal, tty, console, shell, and command line are technically different things, but are commonly used interchangeably.

While capital-T Terminal and iTerm2 are GUIs for accessing the lowercase-t terminal, the command language itself is usually called BASH, which stands for the Bourne-Again SHell, named for British computer scientist Steve Bourne. When you see a long prompt followed by a dollar-sign, you're typically at a BASH prompt. BASH is a REPL program.

Mystery Shells



If you don't see a lot of text before the dollar-sign, like in this shot from Tron Legacy, you may be at some other type of shell program - one that may not support all the commands I've shown you today.

In this case, you can usually type the command "bash" to launch a Bash shell within this mystery shell.