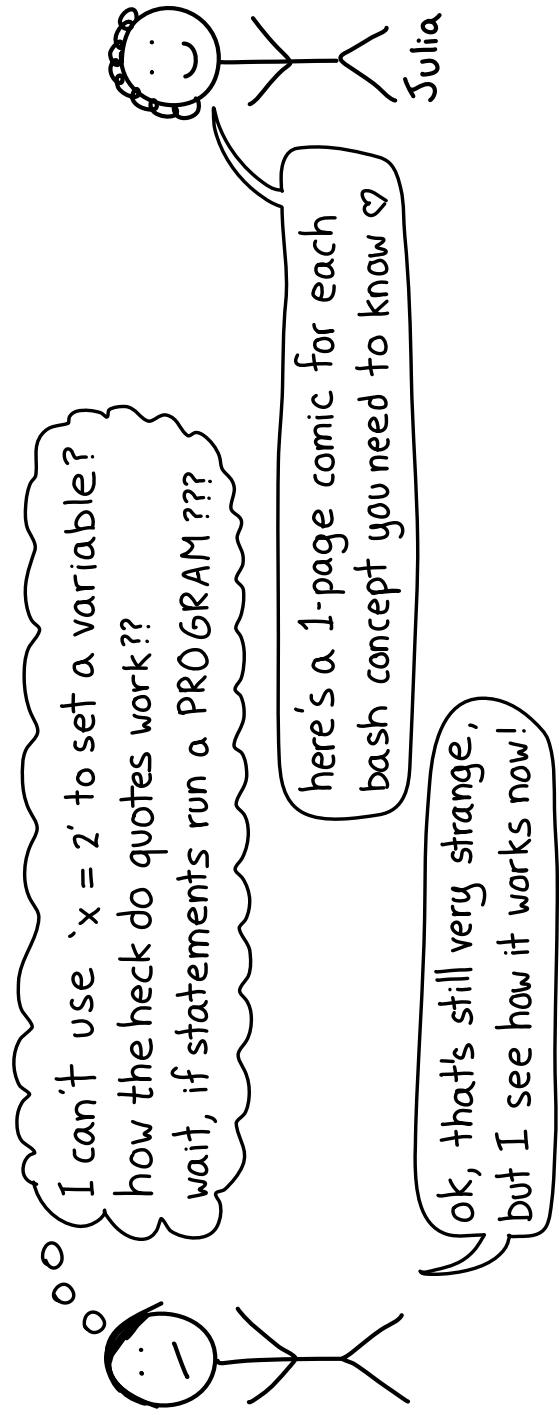


o this?
more at

* wizardzines.com *

hello! we're here because bash* is a very weird programming language.



*most of this zine also applies to other shells, like zsh

thanks for reading

There's more to learn about bash than what's in this zine, but I've written a lot of bash scripts and this is all I've needed so far. If the task is too complicated for my bash skills, I just use a different language.

two pieces of parting advice:

- ① when your bash script does something you don't understand, figure out why! ← ok, this is my advice for literally all programming :)
- ② use shellcheck! And read the shellcheck wiki when it tells you about an error :)

credits

Cover art: Vladimir Kasićović
Editing: Dolly Lanuza, Kamal Marhubi
Copy Editing: Courtney Johnson
and thanks to all 11 beta readers ♥

debugging

our hero: set -x	or bash -x	you can stop before every line
set -x prints out every line of a script as it executes, with all the variables expanded!	\$ bash -x script.sh does the same thing as putting set -x at the top of script.sh	trap read DEBUG the DEBUG "signal" is triggered before every line of code
#!/bin/bash I usually put set -x at the top		
a fancy step debugger trick		
put this at the start of your script to confirm every line before it runs:	trap 'read -p "[\${BASH_SOURCE}: \${LINENO}] \${BASH_COMMAND}"' DEBUG read -p prints a script line next command message, press filename number that will run enter to continue	how to print better error messages this die function: die() { echo \$1 >&2; exit 1; } lets you exit the program and print a message if a command fails, like this: some_command die "oh no!"
basics	cheat sheets (in the middle!)	getting fancy
why I > bash 4	if statements 16	\$() \${{}} !!
POSIX 5	for loops 17	
shellcheck 6	reading input 18	
variables 7	functions 19	
env variables 8	pipes 20	
arguments 9	parameter expansion 21	
builtins 10	background processes 22	
quotes 11	subshells 23	
globs 12	trap 24	
redirects 13	errors 25	
	debugging 26	

table of contents

why I ❤️ bash

4

it's SO easy to get started

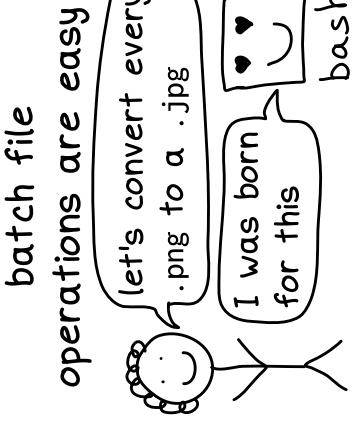
Here's how:

- ① Make a file called hello.sh and put some commands in it, like ls /tmp
- ② Run it with bash hello.sh

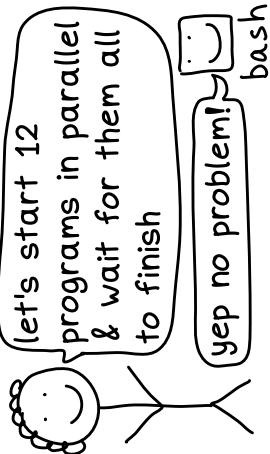
pipes & redirects are super easy

managing pipes in other languages is annoying. in bash, it's just:

cmd1 | cmd2

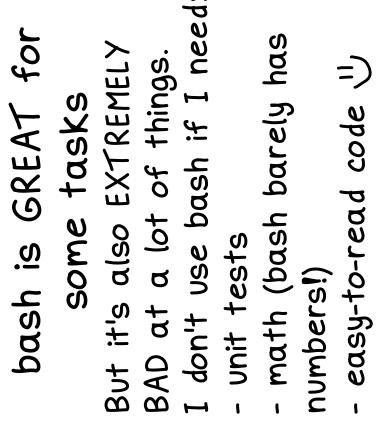
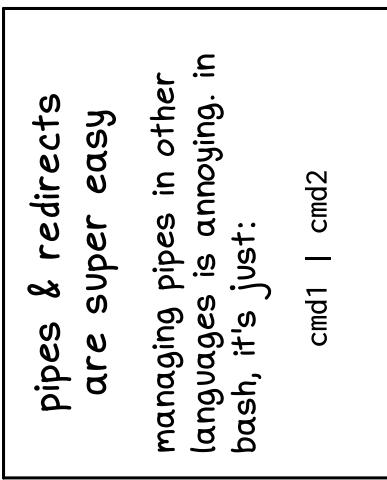


it's surprisingly good at concurrency



♥ it doesn't change ♥

bash is weird and old, but the basics of how it works haven't changed in 30 years. If you learn it now, it'll be the same in 10 years.

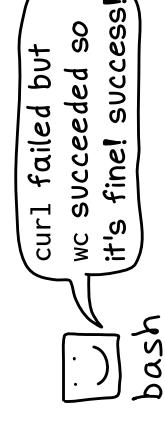


25

errors

by default, a command failing doesn't fail the whole pipeline

curl yxqzq.ca | wc



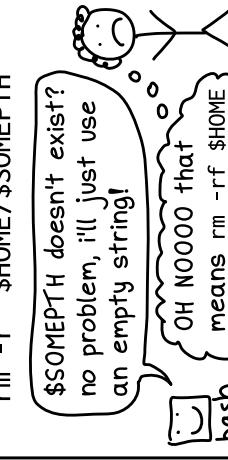
set -o pipefail makes the pipe fail if any command fails

you can combine set -e, set -u, and set -o pipefail into one command I put at the top of all my scripts:
` set -eu pipefail`



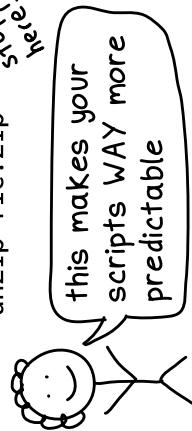
by default, unset variables don't error

rm -r "\$HOME/\$SOMEPTH"



set -u stops the script on unset variables

set -u
rm -r "\$HOME/\$SOMEPTH"



by default, bash will continue after errors

that program's exit status was 1? who cares, let's keep running!!!



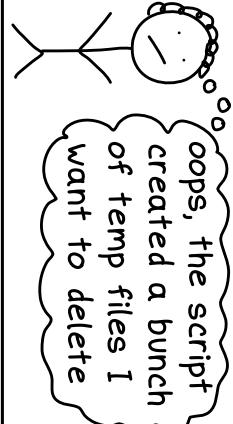
set -e stops the script on errors

set -e
unzip file.zip



trap

when your script exits, sometimes you need to clean up



events you can trap

- unix signals (INT, TERM, etc)
- the script exiting (EXIT)
- every line of code (DEBUG)
- function returns (RETURN)

trap COMMAND EVENT
what to run when to run the command to run

bash runs COMMAND when EVENT happens

```
trap "echo 'hi!!!'" INT
      <--- sends SIGINT signal>
      <--- ok, time to print out 'hi!!!'
bash
```

example: cleanup files when the script exits

```
function cleanup() {
    rm -rf $TEMPDIR
    rm $TEMPFILE
}
trap cleanup EXIT
```

POSIX compatibility

5

some shells have extra features

we have extra features that aren't in POSIX

sh
zsh
ksh
dash

we keep it simple & just do what POSIX says

sh
zsh
ksh
dash

this zine is about bash scripting

most things in this zine will work in any shell, but some won't! page 15 lists some non-POSIX features

on most systems, /bin/sh only supports POSIX features

if your script has #!/bin/sh at the top, don't use bash-only features in it!

POSIX is a standard that defines how Unix shells should work

if your script sticks to POSIX, we'll all run it the same way! (mostly :)

sh
zsh
ksh
dash
fish

I don't care about POSIX

some people write all their scripts to follow POSIX

I only use POSIX features

I use lots of bash-only features!

me

shellcheck

6

shellcheck finds problems with your shell scripts

\$ shellcheck my-script.sh
oops, you can't use =~ in an if [...]!

it checks for hundreds of common shell scripting errors

hey, that's a bash-only feature but your script starts with #!/bin/sh



it even tells you about misused commands

hey, it looks like you're not using grep correctly here
wow, I'm not! thanks!

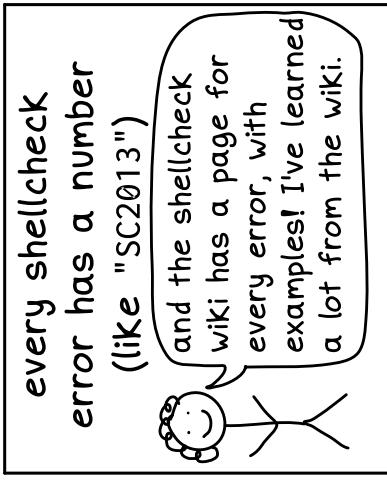
your text editor probably has a shellcheck plugin

I can check your shell scripts every time you save!



every shellcheck error has a number (like "SC2013")

and the shellcheck wiki has a page for every error, with examples! I've learned a lot from the wiki.



23

subshells

a subshell is a child shell process

hey, can you run this bash code for me?
sure thing! other bash process

(var=3
(var=2
echo \$var

some ways to create a subshell

- ① put code in parentheses (...) (cd \$DIR; ls)
runs in subshell
- ② put code in \$(...) var=\$(cat file.txt)
runs in subshell
- ③ pipe/redirect to a code block cat x.txt | while read line...
piping to a loop makes the loop run in a subshell
- ④ + lots more for example, process substitution <() creates a subshell

cd in a subshell doesn't cd in the parent shell

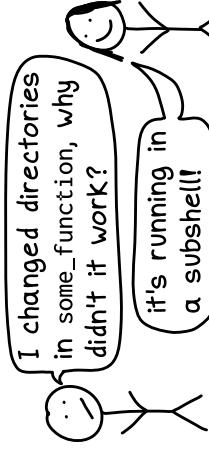
(cd subdir/
mv x.txt y.txt
) I like to do this so I don't have to remember to cd back at the end!

setting a variable in a subshell doesn't update it in the main shell

var=3
(var=2
echo \$var
this prints 3, not 2

it's easy to create a subshell and not notice

x=\$(some_function)
I changed directories in some_function, why didn't it work?
it's running in a subshell!



background processes

scripts can run many processes in parallel

```
python -m http.server &
curl localhost:8080
```

& starts python in the "background", so it keeps running while curl runs

background processes
sometimes exit when you close your terminal

you can keep them running with nohup or by using tmux/screen.

```
$ nohup ./command &
```

wait waits for all background processes to finish

```
command1 &
command2 &
wait
```

this waits for both command1 and command2 to finish

jobs, fg, bg, and disown let you juggle many processes in the same terminal, but I almost always just use multiple terminals instead

concurrency is easy* in bash
in other languages:
thing1 & thing2 & wait
*(if you keep it very simple)

jobs disown list shell's like nohup, background but after processes started move process to fg and bg foreground/background

variables

there are no numbers, only strings

a=2
a="2"
"2"
both of these are the string

technically bash can do arithmetic but I avoid it

how to set a variable

var=value^{right!}
(no spaces!)

var = value^{wrong}

var = value will try to run the program var with the arguments "==" and "value"

always use quotes around variables

```
wrong"
$ cat $filename
```

```
right!
$ cat "$filename"
```

ok, I'll run 2 files!
oh no!
we didn't mean that!

um swan and 1.txt
don't exist...
cat

that's a file! yay!

To add a suffix to a variable like "2", you have to use \${varname}. Here's why:

```
$ zoo=panda
$ echo "$zoo2"
```

prints "",
zoo2 isn't a variable

```
$ echo "${zoo}2"
```

this prints "panda2" like we wanted

Environment variables

8

every process has environment variables printing out your shell's environment variables is easy, just run:

```
$ env
```

shell scripts have 2 kinds of variables
1. environment variables
2. shell variables
unlike in other languages, in bash you access both of these in the exact same way: \$VARIABLE

child processes inherit environment variables
this is why the variables set in your .bashrc are set in all programs you start from the terminal.
They're all child processes of your bash shell!

shell variables aren't inherited
var=panda
\$var only gets set in this process, not in child processes

export sets environment variables
how to set an environment variable:
export ANIMAL=panda
or turn a shell variable into an environment variable
ANIMAL=panda
export ANIMAL

you can set env vars when starting a program
2 ways to do it (both good):
① \$ env VAR=panda ./myprogram
OK! I'll set VAR to :panda and then start ./myprogram
env
② \$ VAR=panda ./myprogram
(here bash sets VAR=panda)

\${}: "parameter expansion" 21

\${...} is really powerful
it can do a lot of string operations!
my favorite is search/replace.

\$[var]
\${var}
\${var/panda}
\${var//panda/panda}
/ replaces first instance,
// replaces every instance
search & replace example:
\$ x="I'm a bearbear!"
\$ echo \${x/bear/panda}
I'm a pandabear!

\$[var:-\$othervar]
use a default value like
\$othervar if var is unset/null

\$[var:?some error]
prints "some error" and exits if var is unset/null
get a substring of var

\$[var:offset:length]
there are LOTS more, look up "bash parameter expansion"!

\$[var#pattern]
\$[var%pattern]
remove the prefix/suffix pattern from var. Example:
\$ x=motorcycle.svg
\$ echo "\${x%.svg}"
motorcycle

Pipes

sometimes you want to send the output of one process to the input of another

```
$ ls | wc -l
53 ↪ 53 files!
```

the OS creates a buffer for each pipe

IN data waiting OUT
to be read

when the buffer gets full:
write(**IN**, "...")
it's full! I'm going
to pause you until
there's room again

OS

a pipe is a pair of 2 magical file descriptors

```
ls → IN ← pipe → OUT → wc
file descriptors
```

named pipes

you can create a file that acts like a pipe with mkfifo

```
$ mkfifo mypipe
$ ls > mypipe &
$ wc < mypipe
```

this does the same thing as ls | wc

When ls does
write(**IN**, "hi")
wc can read it!
read(**OUT**) → "hi"
Pipes are one way. →
You can't write to **OUT**

arguments

q

get a script's arguments with \$0, \$1, \$2, etc

```
$ svg2png old.svg new.png
```

\$0 is \$1 is \$2 is

"svg2png" "old.svg" "new.png"
(script's name)

arguments are great for making simple scripts

Here's a 1-line svg2png script I use to convert SVGs to PNGs:

```
#!/bin/bash
inkscape "$1" -b white --export-png="$2"
```

I run it like this:
\$ svg2png old.svg new.png

always quote your variables!

"\$@": all arguments

\$@ is an array of all the arguments except \$0.

This script passes all its arguments to ls --color:

```
#!/bin/bash
ls --color "$@"
```

you can loop over arguments

```
for i in "$@"
do
  ...
```

in our svg2png example, this would loop over old.svg and new.png

shift removes the first argument

echo \$1 ← this prints the script's first shift argument
echo \$1 ← this prints the second argument

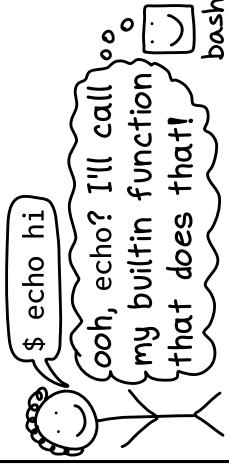
builtins

10

**most bash commands
are programs**

You can run which to find out which binary is being used for a program:
\$ which ls
/bin/ls

but some commands
are functions inside
the bash program



type tells you if a command is a builtin

```
$ type grep  
grep is /bin/grep  
$ type echo  
echo is a builtin  
$ type cd  
cd is a builtin
```

examples of builtins

type source
declare
alias
read
cd
printf
echo

a useful builtin:

alias

alias lets you set up shorthand commands, like:
alias gc="git commit"
~/.bashrc runs when bash starts, put aliases there!

a useful builtin:

source

bash script.sh runs script.sh in a subprocess, so you can't use its variables / functions.
source script.sh is like pasting the contents of script.sh

19

functions

**defining functions
is easy**

```
say_hello() {  
    echo "hello!"  
}
```

... and so is calling them
say_hello()
no parentheses!

**functions have
exit codes**

```
failing_function() {  
    return 1  
}
```

0 is success, everything else is a failure. A program's exit codes work the same way.

**you can't return
a string**

you can only
return exit
codes 0 to 255!
say_hello() {
 return "Hello!"
}

**local x=VALUE
suppresses errors**

local x=\$asdf
never fails,
even if asdf
doesn't exist
local x
x=\$asdf
this one
will fail
I have NO IDEA why
it's like this, bash is
weird sometimes

**the local keyword
declares local variables**

```
say_hello() {  
    local x  
    x=$(date)  
    local  
    y=$(date)  
}
```

**arguments are
\$1, \$2, \$3, etc**

```
say_hello() {  
    echo "Hello $1!"  
}  
say_hello "Ahmed"  
not say_hello("Ahmed")!
```

reading input

read -r var
reads stdin into
a variable

```
$ read -r greeting
hello there! ← type here
echo "$greeting" and press
enter
hello there!
```

set IFS='' to avoid
stripping whitespace

```
$ IFS=''
IFS='empty string
hi there!
read -r greeting
echo "$greeting"
hi there!
the spaces are
still there!
```

you can also read
into multiple variables

```
$ read -r name1 name2
ahmed fatima
$ echo "$name2"
fatima
```

by default, read
strips whitespace

" a b c " -> "a b c"
it uses the IFS ("Input
Field Separator") variable
to decide what to strip

more IFS uses: loop over every line of a file
by default, for loops will loop over every word of a file
(not every line). Set IFS=''

to loop over every line instead!

```
IFS=''
don't forget → for line in $(cat file.txt)
to unset IFS
when you're
do
echo $line
done!
```

quotes

double quotes expand variables,
single quotes don't

```
$ echo 'home: $HOME'      $ echo "home: $HOME"
home: $HOME               home: /home/bork
single quotes always      ↗
give you exactly what    $HOME got expanded
you typed in              to /home/bork
```

how to concatenate
strings

put them next to each other!

```
$ echo "hi ""there"
hi there
x + y doesn't add strings:
$ echo "hi" ± " there"
hi ± there
```

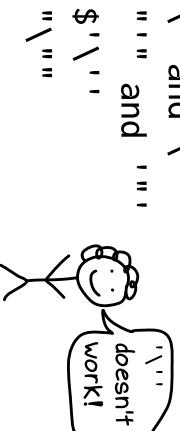
||

you can quote
multiline strings

```
$ MESSAGE="Usage:
here's an explanation of
how to use this script!"
```

a trick to escape
any string: !:q:p

get bash to do it for you!
\$ # He said "that's \$5"
\$!:q:p
He said "that\"'s \$5"
this only works in bash, not zsh.
! is an "event designator" and
:q:p is a "modifier"



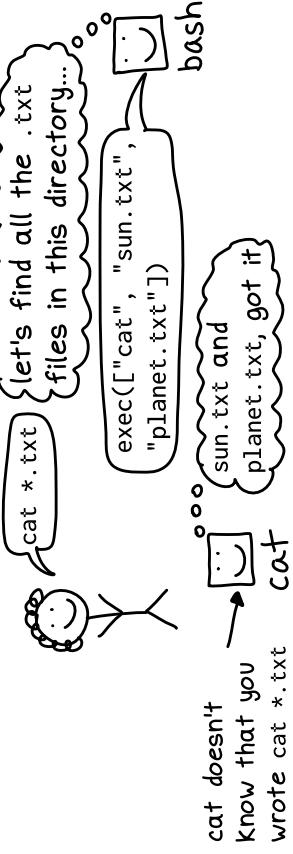
globs

12

globs are a way to match strings

beware: the * and the ? in a glob are different than * and ? in a regular expression!!!
bear* → matches, bear → doesn't match ↗ bugbear x

bash expands globs to match filenames



there are just 3 special characters

* matches 0+ characters
? matches 1 character
[abc] matches a or b or c
I usually just use * in my globs

use quotes to pass a literal '*' to a command

```
$ egrep 'b.*' file.txt
```

the regexp 'b.*' needs to be quoted so that bash won't translate it into a list of files with b. at the start

filenames starting with a dot don't match

... unless the glob starts with a dot, like .bash*
ls *.txt

there's .bees.txt, but I'm not going to include that bash

for loops

17

for loop syntax

```
for i in panda swan
do
  echo "$i"
done
```

the semicolons are weird
usually in bash you can always replace a newline with a semicolon. But not with for loops!
for i in a b; do ...; done
you need semicolons before do and done but it's a syntax error to put one after do

looping over files is easy

```
for i in *.png
do
  convert "$i" "${i/png/jpg}"
done
```

this converts all png files to jpgs!

looping over a range of numbers

3 ways:

```
for i in $(seq 1 5)
for i in {1..5}
for ((i=1; i<6; i++))
```

these two only work in bash, not sh

for loops loop over words, not lines

```
for word in $(cat file.txt)
do
  ...
done
```

loops over every word in the file, NOT every line (see page 18 for how to change this!)

while loop syntax

```
while COMMAND
do
```

like an if statement, runs COMMAND and checks if it returns 0 (success)

if statements

in bash, every command has an exit status

0 = success
any other = failure

bash puts the exit status of the last command in a special variable called \$?

why is 0 success?

there's only one way to succeed, but there are LOTS of ways to fail. For example

will exit with status:
1 if THING isn't in x.txt
2 if x.txt doesn't exist

grep THING x.txt

[vs [[

there are 2 commands often used in if statements: [and [[
if [-e file.txt] if [[-e file.txt]]
/usr/bin/[(aka test) is [[is built into bash. It
a program* that returns 0 if the test you pass it succeeds

*in bash, [is a builtin that acts like /usr/bin/[

true is a command that always succeeds, not a boolean

combine with && and ||
if [-e file1] && [-e file2]
man test for more on [[
you can do a lot!

bash if statements test if a command succeeds

if COMMAND; then
do a thing

this:

① runs COMMAND

② if COMMAND returns 0 (success), then do the thing

> redirects <

13

unix programs have 1 input and 2 outputs

When you run a command from a terminal, they all go to/from the terminal by default.

< redirects stdin
\$ wc < file.txt
\$ cat file.txt | wc
these both read file.txt to wc's stdin

stdout (1) → \$-
cat file.txt | wc → \$-
terminal → program

each input/output has a number (its "file descriptor")

2>&1 redirects stderr to stdout

\$ cmd > file.txt 2>&1

cmd → stdout (1) → file.txt → stderr (2) → 2>&1

/dev/null
your operating system ignores all writes to /dev/null.

\$ cmd > /dev/null

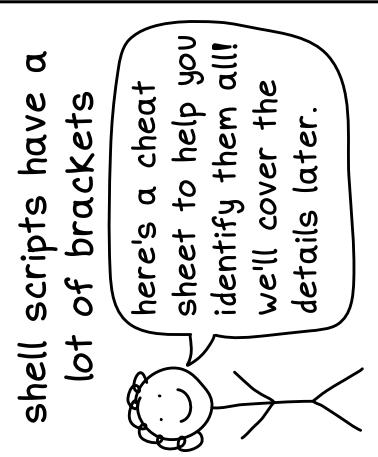
cmd → stdout (1) → /dev/null → stderr (2) → \$-

sudo doesn't affect redirects

your bash shell opens a file to redirect to it, and it's running as you. So \$ sudo echo x > /etc/xyz won't work. do this instead: \$ echo x | sudo tee /etc/xyz

brackets cheat sheet

| 4



(cd ~/music; pwd)
(...) runs commands in a
subshell.

```
VAR=$(cat file.txt)  
$(COMMAND) is equal to  
COMMAND's stdout
```

```
{ cd ~/music; pwd }  
{} groups commands.  
runs in the same process.
```

```
x=(1 2 3)
```

x=(...) creates an array

```
<(COMMAND)
```

"process substitution":
an alternative to pipes

```
$[var//search/replace] see page 21 for more  
about ${...}!
```

```
if [ ... ]
```

/usr/bin/[is a program
that evaluates statements

```
if [[ ... ]]  
[] is bash syntax. it's  
more powerful than [
```

```
x=$((2+2))
```

\$() does arithmetic

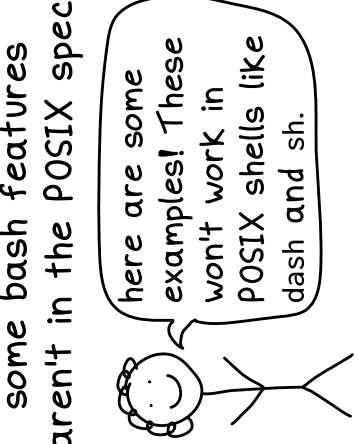
```
a{.png,.svg}
```

this expands to a.png a.svg

it's called "brace expansion"

non-POSIX features

| 5



[[...]]
POSIX alternative:
[...]

```
diff <./cmd1> <./cmd2>  
this is called "process  
substitution", you can use  
named pipes instead
```

```
a.{png,svg}
```

you'll have to type
a.png a.svg

```
{1..5}
```

POSIX alternative:
\$(seq 1 5)

```
$' \n'
```

POSIX alternative:
\$(printf "\n")

```
for ((i=0; i <3; i++))  
sh only has for x in ...  
loops, not C-style loops
```

```
[[ $DIR = /home/* ]]  
POSIX alternative:  
match strings with grep  
to sed
```