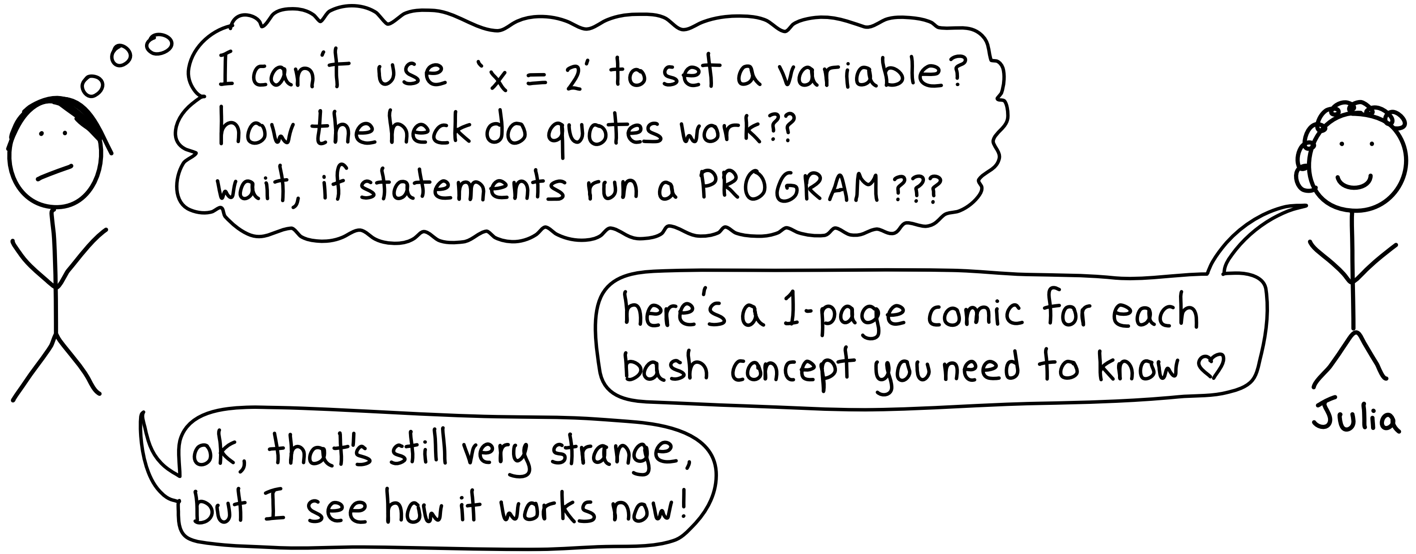
Bite Size Bash

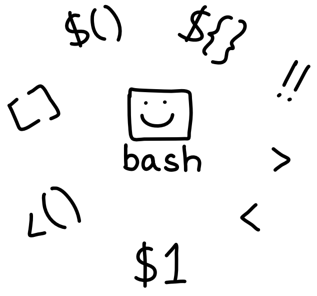
by Julia Evans

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


I can't use `x = 2' to set a variable?
how the heck do quotes work??
wait, if statements run a PROGRAM???
here's a 1-page comic for each bash concept you need to know ♡
Julia
ok, that's still very strange, but I see how it works now!

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

table of contents



why I ♡ bash

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`

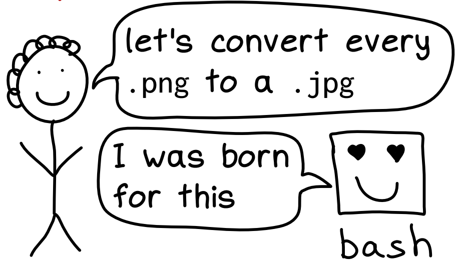
② Run it with `bash hello.sh`

pipes & redirects are super easy

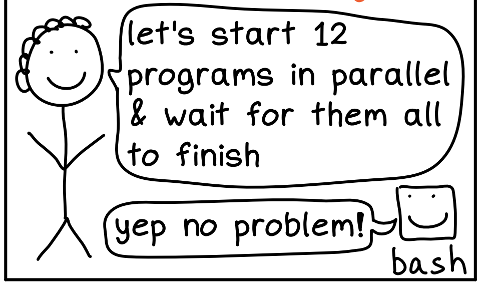
managing pipes in other languages is annoying. in bash, it's just:

```
cmd1 | cmd2
```

batch file operations are easy



it's surprisingly good at concurrency


let's start 12 programs in parallel & wait for them all to finish
yep no problem!
bash

♥ 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.

bash is GREAT for some tasks

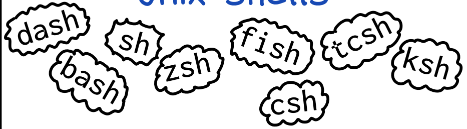
But it's also EXTREMELY BAD at a lot of things.

I don't use bash if I need:

- unit tests
- math (bash barely has numbers!)
- easy-to-read code ☺

POSIX compatibility

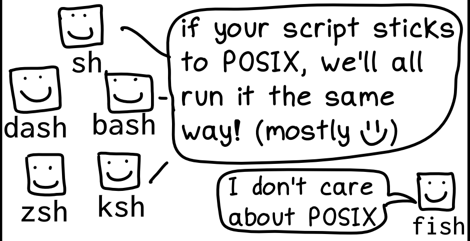
there are lots of Unix shells



you can find out your user's default shell by running:

```
$ echo $SHELL
```

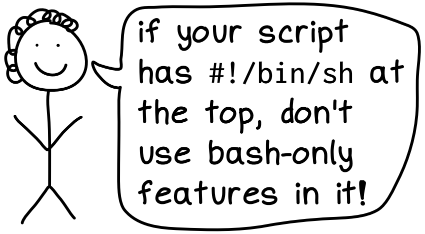
POSIX is a standard that defines how Unix shells should work


sh
dash
bash
zsh
ksh
if your script sticks to POSIX, we'll all run it the same way! (mostly :))
I don't care about POSIX
fish

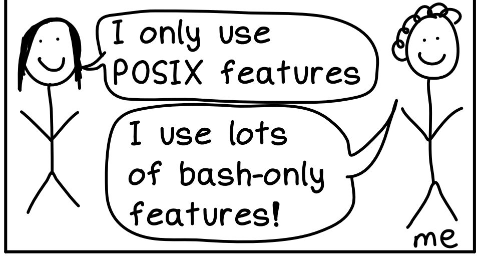
some shells have extra features

bash
zsh
ksh
we have extra features that aren't in POSIX
sh
dash
we keep it simple & just do what POSIX says

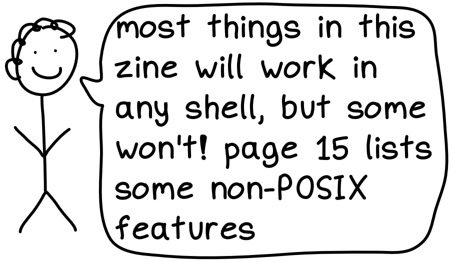
on most systems, /bin/sh only supports POSIX features


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

some people write all their scripts to follow POSIX



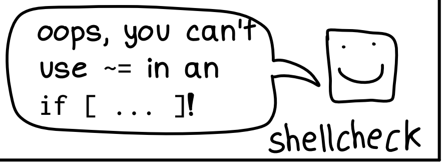
this zine is about bash scripting


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

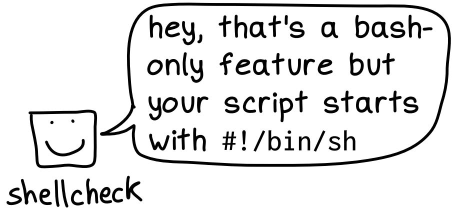
shellcheck

shellcheck finds problems with your shell scripts

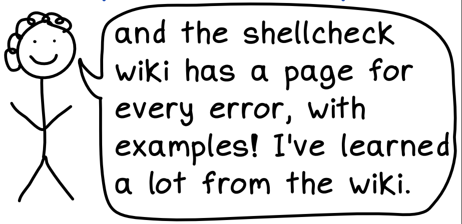
```
$ shellcheck my-script.sh
```



it checks for <u>hundreds</u> of common shell scripting errors


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

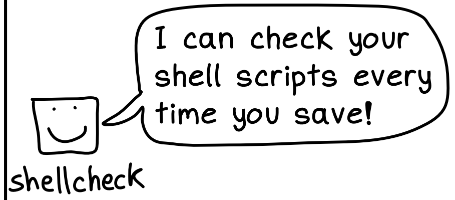
every shellcheck error has a number (like "SC2013")


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

it even tells you about misused commands

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

your text editor probably has a shellcheck plugin



basically, you should probably use it

It's available for every operating system!
Try it out at:

https://shellcheck.net

variables

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"`

how to use a variable: "$var"

```
filename=blah.txt
echo "$filename"
```

they're case sensitive. environment variables are traditionally all-caps, like `$HOME`

there are no numbers, only strings

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

always use quotes around variables

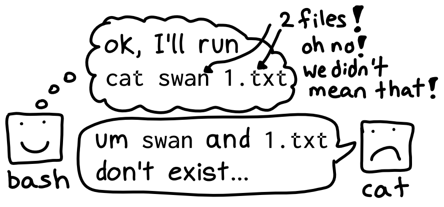
```
$ filename="swan 1.txt"
```

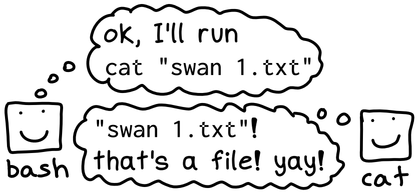
wrong!!

```
$ cat $filename
```

right!

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


ok, I'll run cat swan 1.txt
2 files! oh no! we didn't mean that!
um swan and 1.txt don't exist...
bash
cat


ok, I'll run cat "swan 1.txt"
"swan 1.txt"! that's a file! yay!
bash
cat

${varname}

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

```
$ zoo=panda
$ echo "$zoo2"
$ echo "${zoo}2"
```

`echo "$zoo2"` ← prints "", zoo2 isn't a variable

`echo "${zoo}2"` ← this prints "panda2" like we wanted

environment variables

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`

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
```

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

you can set env vars when starting a program

2 ways to do it (both good!):

① `$ env VAR=panda ./myprogram`

② `$ VAR=panda ./myprogram`

(here bash sets VAR=panda)

arguments

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

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

$0 is "svg2png" (script's name)

$1 is "old.svg"

$2 is "new.png"

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"
```

always quote your variables!

I run it like this:

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

"$@": 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
    ...
done
```

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

shift removes the first argument

```
echo $1
shift
echo $1
```

this prints the script's first argument

this prints the second argument

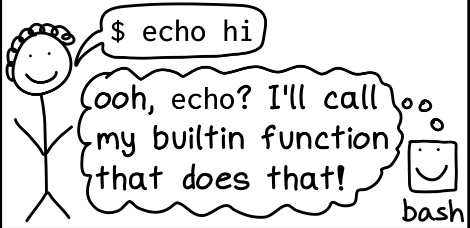
builtins

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` `alias` `declare` `read` `printf` `cd` `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`

quotes

double quotes expand variables, single quotes don't

```
$ echo 'home: $HOME'
home: $HOME
```

single quotes always give you exactly what you typed in

```
$ echo "home: $HOME"
home: /home/bork
```

$HOME got expanded to /home/bork

you can quote multiline strings

```
$ MESSAGE="Usage:

here's an explanation of
how to use this script!"
```

how to concatenate strings

put them next to each other!

```
$ echo "hi ""there"
hi there
```

x + y <u>doesn't</u> add strings:

```
$ echo "hi" + " there"
hi + there
```

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

get bash to do it for you!

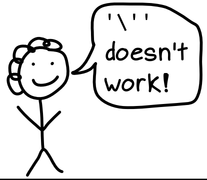
```
$ # He said "that's $5"
$ !:q:p
'# He said "that'\''s $5"'
```

this only works in bash, not zsh.
! is an "event designator" and :q:p is a "modifier"

escaping ' and "

here are a few ways to get a ' or ":

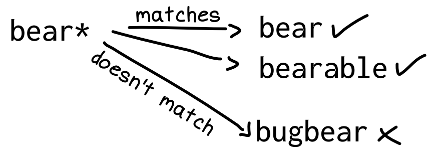
```
\' and \"
"'" and '"'
$'\''
"\""
```



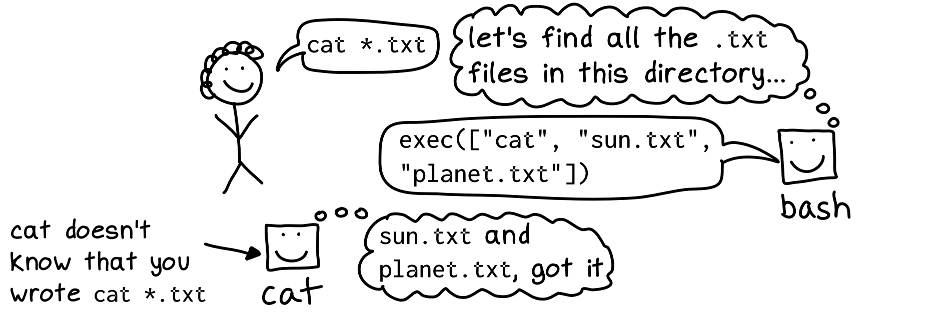
globs

globs are a way to match strings

beware: the * and the ? in a glob are different than * and ? in a regular expression!!!



bash expands globs to match filenames


cat *.txt
let's find all the .txt files in this directory...
exec(["cat", "sun.txt", "planet.txt"])
bash
cat doesn't know that you wrote cat *.txt
cat
sun.txt and planet.txt, got it

there are just 3 special characters

* matches 0+ characters
? matches 1 character
[abc] matches a or b or c

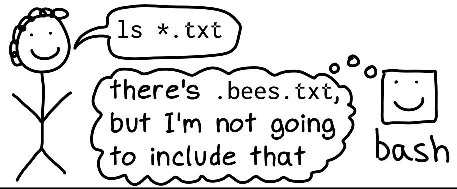
use quotes to pass a literal '*' to a command

```
$ 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*



> redirects <

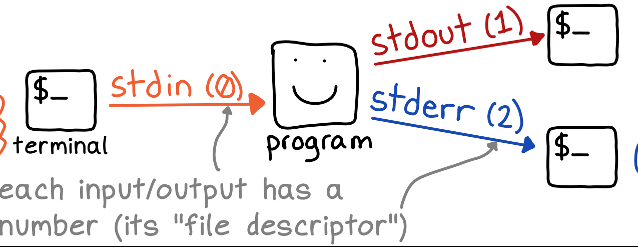
unix programs have 1 input and 2 outputs

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



> redirects stdout

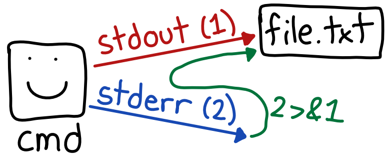
```
$ cmd > file.txt
```

2> redirects stderr

```
$ cmd 2> file.txt
```

2>&1 redirects stderr to stdout

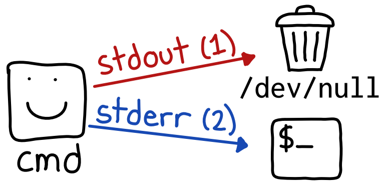
```
$ cmd > file.txt 2>&1
```



/dev/null

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

```
$ cmd > /dev/null
```



sudo doesn't affect redirects

your bash shell opens a file to redirect to it, and it's running as you. So

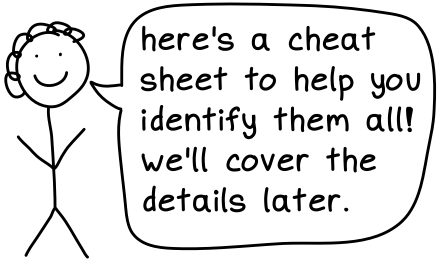
```
$ sudo echo x > /etc/xyz
```

won't work. do this instead:

```
$ echo x | sudo tee /etc/xyz
```

brackets cheat sheet

shell scripts have a lot of brackets


here's a cheat sheet to help you identify them all! we'll cover the details later.

`(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

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

$(()) does arithmetic

`if [ ... ]`

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

`<(COMMAND)`

"process substitution": an alternative to pipes

`a{.png,.svg}`

this expands to a.png a.svg it's called "brace expansion"

`if [[ ... ]]`

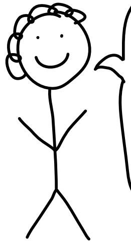
[[is bash syntax. it's more powerful than [

`${var//search/replace}`

see page 21 for more about ${...}!

non-POSIX features

some bash features aren't in the POSIX spec



here are some examples! These won't work in POSIX shells like dash and sh.

`[[ ... ]]`

POSIX alternative:
`[ ... ]`

`a.{png,svg}`

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

`diff <(./cmd1) <(./cmd2)`

this is called "process substitution", you can use named pipes instead

`{1..5}`

POSIX alternative:
`$(seq 1 5)`

arrays

POSIX shells only have one array: `$@` for arguments

the `local` keyword

in POSIX shells, all variables are global

`$'\n'`

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

`[[ $DIR = /home/* ]]`

POSIX alternative:
match strings with `grep`

`for ((i=0; i <3; i++))`

sh only has `for x in ...` loops, not C-style loops

`${var//search/replace}`

POSIX alternative: pipe to `sed`

if statements

in bash, every command has an exit status

0	=	success
any other number	=	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

```
grep THING x.txt
```

will exit with status:

1 if THING isn't in x.txt
2 if x.txt doesn't exist

bash if statements test if a command succeeds

```
if COMMAND; then
    # do a thing
fi
```

this:

① runs COMMAND
② if COMMAND returns 0 (success), then do the thing

[vs [[

there are 2 commands often used in if statements: [and [[

```
if [ -e file.txt ]
```

/usr/bin/[(aka test) is a program* that returns 0 if the test you pass it succeeds

```
if [[ -e file.txt ]]
```

[[is built into bash. It treats asterisks differently:

```
[[ $filename = *.png ]]
```

doesn't expand *.png into files ending with .png

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

true

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!

for loops

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!

for loops loop over words, not lines

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

loops over every word in the file, __NOT__ every line (see page 18 for how to change this!)

while loop syntax

```
while COMMAND
do
   ...
done
```

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

how to loop 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

reading input

read -r var reads stdin into a variable

```
$ read -r greeting
  hello there!
$ echo "$greeting"
hello there!
```

type here and press enter

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

set IFS='' to avoid stripping whitespace

empty string

```
$ IFS='' read -r greeting
    hi there!
$ echo "$greeting"
    hi there!
```

the spaces are still there!

more IFS uses: loop over every line of a file

by default, for loops will loop over every <u>word</u> of a file (not every line). Set IFS='' to loop over every line instead!

don't forget to unset IFS when you're done!

```
IFS=''
for line in $(cat file.txt)
do
    echo $line
done
```

functions

defining functions is easy

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

... and so is calling them

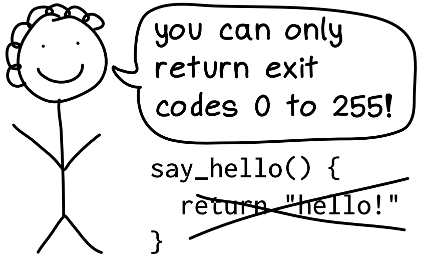
```
say_hello
```
← no parentheses!

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



```
say_hello() {
  ~~return "hello!"~~
}
```

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

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

the local keyword declares local variables

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

y=$(date) ← global

local x=VALUE suppresses errors

```
local x=$(asdf)
```
← never fails, even if asdf doesn't exist

```
local x
x=$(asdf)
```
← this one will fail

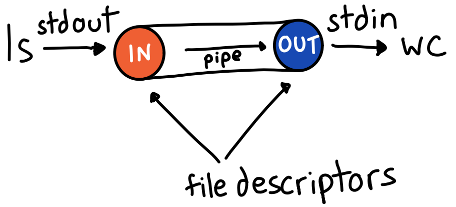
pipes

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

```
$ ls | wc -l
53
```

↖ 53 files!

a pipe is a pair of 2 magical file descriptors



when ls does

```
write(IN, "hi")
```

wc can read it!

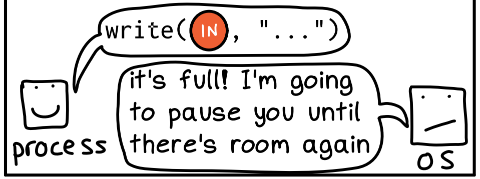
```
read(OUT) ⟶ "hi"
```

Pipes are one way. ⟶
You can't write to OUT

the OS creates a buffer for each pipe

IN
data waiting to be read
OUT

when the buffer gets full:



named pipes

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

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

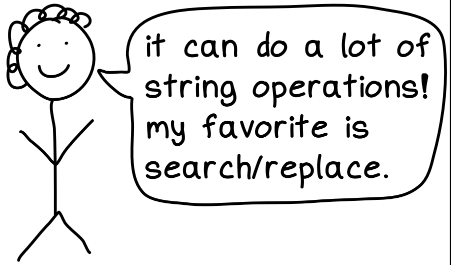
this does the same thing as `ls | wc`

you can use pipes in other languages!

only shell has the syntax `process1 | process2` but you can create pipes in basically any language!

${}: "parameter expansion"

${...} is really powerful


it can do a lot of string operations! my favorite is search/replace.

${var}

see page 7 for when to use this instead of `$var`

${#var}

length of the string or array `var`

${var/bear/panda}
${var//bear/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

${var#pattern}
${var%pattern}

remove the prefix/suffix `pattern` from `var`. Example:

```
$ x=motorcycle.svg
$ echo "${x%.svg}"
motorcycle
```

${var:offset:length}

get a substring of `var`

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

wait waits for all background processes to finish

```
command1 &
command2 &
wait
```

this waits for both command1 and command2 to finish

concurrency is easy* in bash

in other languages:

in bash:

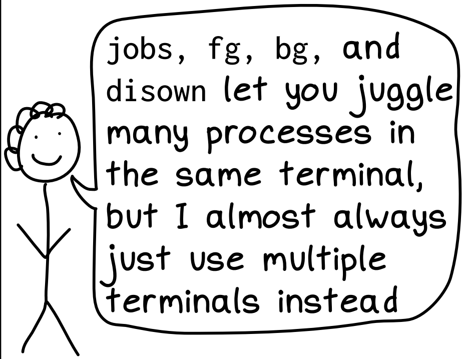
```
thing1 &
thing2 &
wait
```

*(if you keep it very simple)

background processes sometimes exit when you close your terminal

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

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


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

jobs

list shell's background processes

disown

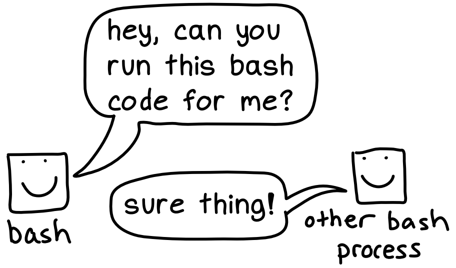
like nohup, but after process has started

fg and bg

move process to foreground/background

subshells

a subshell is a child shell process



some ways to create a subshell

① put code in parentheses (...)

$\underbrace{\texttt{(cd \$DIR; ls)}}_{\text{runs in subshell}}$

② put code in $(...)

`var=$(`$\underbrace{\texttt{cat file.txt}}_{\text{runs in subshell}}$`)`

③ pipe/redirect to a code block

`cat x.txt |` $\underbrace{\texttt{while read line...}}_{\text{piping to a loop makes the loop run in a subshell}}$

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!

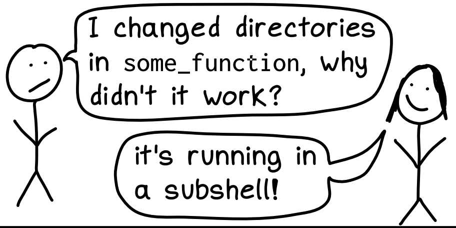
setting a variable in a subshell doesn't update it in the main shell

```
var=3
(var=2)
echo $var
```

this prints 3, not 2

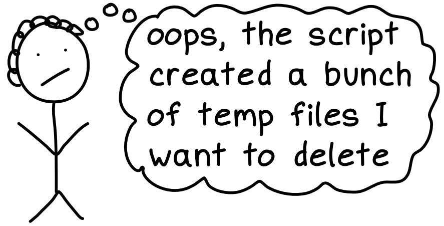
it's easy to create a subshell and not notice

```
x=$(some_function)
```


I changed directories in some_function, why didn't it work?
it's running in a subshell!

trap

when your script exits, sometimes you need to clean up


oops, the script created a bunch of temp files I want to delete

trap sets up callbacks

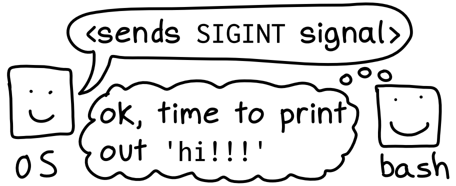
```
trap COMMAND EVENT
```

what command to run → COMMAND

when to run the command → EVENT

bash runs COMMAND when EVENT happens

```
trap "echo 'hi!!!'" INT
```



events you can trap

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

example: kill all background processes when Ctrl+C is pressed

```
trap 'kill $(jobs -p)' INT
```

important: single quotes!

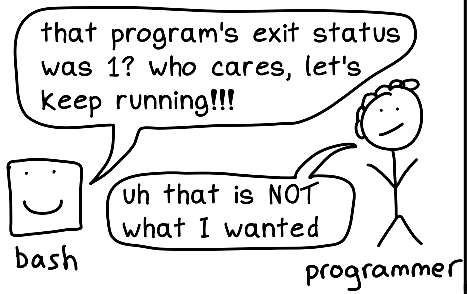
when you press CTRL+C, the OS sends the script a SIGINT signal

example: cleanup files when the script exits

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

errors

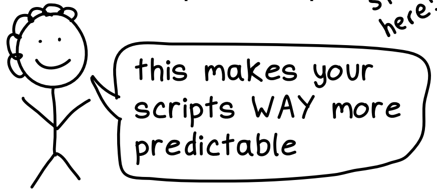
by default, bash will continue after errors


that program's exit status was 1? who cares, let's keep running!!!
uh that is NOT what I wanted
bash
programmer

set -e stops the script on errors

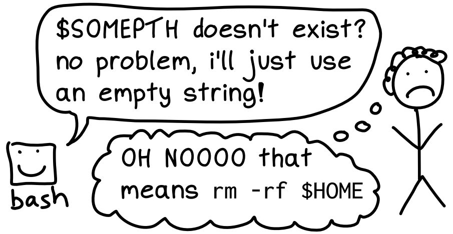
```
set -e
unzip fle.zip
```

typo! script stops here!



by default, unset variables don't error

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


$SOMEPTH doesn't exist? no problem, i'll just use an empty string!
OH NOOOO that means rm -rf $HOME
bash

set -u stops the script on unset variables

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

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

```
curl yxqzq.ca | wc
```

curl failed but wc succeeded so it's fine! success!
bash

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 -euo pipefail
```

debugging

our hero: set -x

`set -x` prints out every line of a script as it executes, with all the variables expanded!

```
#!/bin/bash
set -x
```

I usually put `set -x` at the top

or bash -x

```
$ bash -x script.sh
```

does the same thing as putting `set -x` at the top of `script.sh`

you can stop before every line

```
trap read DEBUG
```

the DEBUG "signal" is triggered before every line of code

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 message, press enter to continue

script filename

line number

next command that will run

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!"
```

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 Kašiković
Editing: Dolly Lanuza, Kamal Marhubi
Copy Editing: Courtney Johnson
and thanks to all 11 beta readers ♥