Shellscripting and Buildtools

Joseph Hallett

February 5, 2024

Whats all this about?

I've written a *lot* of code over the years:

Assembly, C and Java as an engineer

Commonlisp for my own projects

Haskell to build compilers

PostScript to draw really efficient diagrams

LaTeX to publish books

...several a dozen other things too

Which language have I written the most code in?

Which language do I use to solve most tasks?

Which language do I like the least?

Shellscripting!

Normally we type commands for the terminal on a commandline...

- But we can automate them and stick them into scripts

Anything you have to do more than once...

Write a script for it!

- Saves a tonne of time
- Often easier than writing a full program

For example...

```
#! /bin/sh
# Solve
GREP=grep
if [ $(uname) = "OpenBSD" ]; then
   # Use GNU Grep on OpenBSD
   GREP=ggrep
fi

${GREP} -Pi "^${1}$" /usr/share/dict/words
```

```
#! /usr/bin/env bash
# Solve Puzzmo's wordbind

DICT=/usr/share/dict/words
REGEX="$(sed -e "s/[ \t]//g" <<<"${*}" \
       | sed -e "s/\(.\)/\1*/g")"
<"${DICT}" awk <<EOS
/^${REGEX}\$/ {
    printf(\"%d\t%s\n\", length(\$0), \$0)
}
EOS \
| sort -n
```

```
~/.local/bin/knotwords 'st[^aeo]pid'
```

- stupid

```
~/.local/bin/wordbind superlite striae
```

- 7 sullies
- 7 suppers
- 7 suppler
- 7 surlier
- 8 peerless
- 8 supplest
- 8 supplier
- 8 supplies
- 8 suppress
- 8 surliest

Or for example...

```
#! /usr/bin/env bash
if [ $1 = "should" -a $2 = "also" -a $3 = "run" ]; then
  shift 3
  gum confirm "Run 'doas $*'?" && doas $*
elif [ $1 = "should" -a $2 = "also" -a $3 = "remove" ]; then
  gum confirm "Delete '$4'?" && doas rm -fr "${4}"
else
  2>&1 printf "[WARNING] You should read the commands you"
  2>&1 printf "paste more carefully\n"
fi
```

Sometimes when I upgrade my computer it tells me to delete some files or run some commands:
`You should also run rcctl restart pf`
Copying and pasting the precise text is a pain...

- Can I just copy the whole line and run that?

(Of course I can... should I though?)

Or for a further example...

```
#! /usr/bin/env bash
# Fix kitty
/usr/local/opt/bin/fix-kitty

# Update sources
cd /usr/src && cvs -q up -Pd -A
cd /usr/ports && cvs -q up -Pd -A
cd /usr/xenocara && cvs -q up -Pd -A
```

After I upgrade my computer I need to run a couple of standard commands.

- I can never remember them
- Batch them up!

So whats this really about?

Shellscripting is about automating all those tedious little jobs

- Byzantine syntax (based on shell commands)
- Awful for debugging
- Requires magical knowledge
- Probably the most useful thing you'll ever learn

Luckily we have help

Shell scripting is somewhat magical, and there are *lots* of gotchas...

https://www.shellcheck.net

Wonderful tool to spot unportable/dangerous things in shell scripts

- Commandline tool available
- Run it on *everything* you ever write
- `shellcheck` is great

```
shellcheck `command -v knotwords`
In /home/joseph/.local/bin/knotwords line 2:
GREP=grep
^--^ SC2209 (warning): Use var=$(command) to assign output (or quote to assign string).


In /home/joseph/.local/bin/knotwords line 3:
if [ $(uname) = "OpenBSD" ]; then
    ^------^ SC2046 (warning): Quote this to prevent word splitting.

For more information:
  https://www.shellcheck.net/wiki/SC2046 -- Quote this to prevent word splitt...
  https://www.shellcheck.net/wiki/SC2209 -- Use var=$(command) to assign outp...
```

So how do you write one?

Start the file with the *shebang* `#!` then the path to the interpreter of the script plus any arguments:

For portable POSIX shellscripts `#! /bin/sh/`

For less portable BASH scripts `#! /usr/bin/env bash`

Then

- `chmod +x my-script.sh`
- `./my-script.sh`

The rest of the file will be run by the interpretter you specified

- or `sh my-script.sh` if you don't want to/can't mark it executable.

(Hey this is also why Python scripts start `#! /usr/bin/env python3`)

Why env?

Hang on, you might be saying, I know that bash is always in `/bin/bash`... can I just put that as my interpretter path?

Yes, but...

In the beginning `/bin` was reserved for just *system* programs

- and `/usr/bin` for admin installed programs
- and `/usr/local/bin` for locally installed programs
- and `/opt/bin` for optional installed programs
- and `/opt/local/bin` for optional locally installed programs
- and `~/.local/bin` for a *users* programs
- ...oh and sometimes they're even mounted on different disks!

This is *kinda* madness.

- So *most* Linux systems said look we'll just stick everything in `/usr/bin` and stop using multiple partitions
- But some said no it should be `/bin`, one said `/Applications/`, and others stuck them in `/usr/bin` but symlinked them to `/bin`
- And on some systems users grew fed up of the outdated system `bash` and compiled their own and installed it in `~/.local/bin` ...
- ...and ever tried using Python `venv`?

env

```
ENV(1)                    General Commands Manual                    ENV(1)

NAME
     env – set and print environment

SYNOPSIS
     env [-i] [name=value ...] [utility [argument ...]]

DESCRIPTION
     env executes utility after modifying the environment as specified on the
     command line.  The option name=value specifies an environment variable,
     name, with a value of value.
```

What `env` does is look through the `PATH` and tries to find the program specified and runs it.

...Path?

There is an environment variable called `PATH` that tells the system where all the programs are:

- Colon separated list of paths

If you want to alter it you can add a line like to your shell's config

```
export PATH="${PATH}:/extra/directory/to/search"
```

Your shells config is *possibly* in `~/.profile` but it often varies... check the `man` page for your `${SHELL}`
Also some shells have different syntax (e.g. `fish`)...

```
$ tr ':' $'\n' <<< $PATH
/home/joseph/.local/share/python/bin
/bin
/usr/bin
/sbin
/usr/sbin
/usr/X11R6/bin
/usr/local/bin
/usr/local/sbin
/home/joseph/.local/bin
/usr/local/opt/bin
/usr/games
/usr/local/games
/usr/local/jdk-17/bin
/home/joseph/.local/share/go/bin
```

Basic Syntax

Shell scripts are written by chaining commands together

- `A; B` run A then run B
- `A | B` run A and feed its output as the input to B
- `A && B` run A and if successful run B
- `A || B` run A and if not successful run B

How does it know if its successful?

Programs *return* a 1 byte exit value (e.g. C main ends with `return 0;`)

- This gets stored into the variable `${?}` after every command runs.
- `0` indicates success (usually)
- `>0` indicates failure (usually)

This can then be used with commands like `test`:

```
do_long_running_command
test $? -eq 0 && printf "Command succeeded\n"
```

Or the slightly shorter:

```
do_long_running_command
[ $? -eq 0 ] && printf "Command succeeded\n"
```

Bonus puzzle

Why is this the case?

```
[ $? -eq 0 ] # works
[$? -eq 0] # doesn't work
```

Variables

All programs have variables... Shell languages are no different:

To create a variable:

```
GREETING="Hello World!"
```

(No spaces around the =)

To use a variable

```
echo "${GREETING}"
```

If you want your variable to exist in the programs you start as an *environment variable*:

```
export GREETING
```

To get rid of a variable

```
unset GREETING
```

Well...

Variables in shell languages *tend* to act more like macro variables.

- There's no penalty for using one thats not defined.

```
NAME='Joe'
unset NAME
echo "Hello, '${NAME}'"
```

```
Hello, ''
```

If this bothers you:

```
set -o nounset
echo "${NAME:? variable 1 passed to program}"
```

(There are a *bunch* of these shell parameter expansion tricks beyond `:?` which can do search and replace, string length and various magic...)

Standard variables

`${0}` Name of the script

`${1}, ${2}, ${3}...` Arguments passed to your script

`${#}` The number of arguments passed to your script

`${@}` and `${*}` All the arguments

Control flow

If statements and *for* loops, with *globbing*, are available:

```
# Or [ -x myscript.sh ];
# Or [[ -x myscript.sh ]]; if using Bash
if test -x myscript.sh; then
   ./myscript.sh
fi

for file in *.py; do
   python "${file}"
done
```

Other loops

Well...okay you only have `for` really... but you can do other things with it:

```
for n in 1 2 3 4 5; do
   echo -n "${n} "
done
```

1 2 3 4 5

```
seq 5
```

1 2 3 4 5

```
for n in $(seq 5); do
   echo -n "${n} "
done
```

1 2 3 4 5

```
seq -s, 5
```

1,2,3,4,5

```
# IFS = In Field Separator
IFS=','
for n in $(seq -s, 5); do
   echo -n "${n} "
done
```

1 2 3 4 5

Case statements too!

```
# Remove everything upto the last / from ${SHELL}
case "${SHELL##*/}" in
   bash) echo "I'm using bash!" ;;
   zsh) echo "Ooh fancy a zsh user!" ;;
   fish) echo "Something's fishy!" ;;
   *)   echo "Ooh something else!" ;;
esac
```

Basename and Dirname

In the previous example I used the `"${VAR##*/}"` trick to remove everything up to the last /...
Which gives you the name of the file neatly...
...but I have to look this up everytime I use it.
Instead we can use `$(basename "${shell}")` to get the same info.

```
echo "${SHELL}"
echo "${SHELL##*/}"
echo "$(basename "${SHELL}")"
echo "$(dirname "${SHELL}")"
```

You can even use it to remove *file extensions*:

```
for f in *.jpg; do
  convert "${f}" "$(basename "${f}" .jpg).png"
done
```

Pipelines

As part of shell scripting, its often useful to build commands out of chains of other commands. For example I can use `ps` to list all the processes on my computer and `grep` to search.

- How many processes is *Firefox* using?

```
ps -A | grep -i firefox
```

```
44179   ??   SpU   0:29.59   firefox
60731   ??   Ip    0:00.08   /usr/local/lib/firefox/firefox   -contentproc   -appDir
57651   ??   IpU   0:00.30   /usr/local/lib/firefox/firefox   -contentproc   {e3aaae0
78402   ??   SpU   0:08.66   /usr/local/lib/firefox/firefox   -contentproc   {1ddabe3
53121   ??   SpU   0:01.79   /usr/local/lib/firefox/firefox   -contentproc   {5f676d2
79118   ??   IpU   0:00.21   /usr/local/lib/firefox/firefox   -contentproc   {40690c1
38067   ??   IpU   0:00.20   /usr/local/lib/firefox/firefox   -contentproc   {6be551d
33456   ??   IpU   0:00.20   /usr/local/lib/firefox/firefox   -contentproc   {8c295ac
82061   ??   R/3   0:00.00   grep                             -i             firefox
```

Too much info!

Lets use the awk command to cut it to just the first and fourth columns!

```
ps -A | grep -i firefox | awk '{print $1, $4}'
```

```
44179   firefox
60731   /usr/local/lib/firefox/firefox
57651   /usr/local/lib/firefox/firefox
78402   /usr/local/lib/firefox/firefox
53121   /usr/local/lib/firefox/firefox
79118   /usr/local/lib/firefox/firefox
38067   /usr/local/lib/firefox/firefox
33456   /usr/local/lib/firefox/firefox
24087   grep
```

Why is grep in there?

Oh yes... when we search for *firefox* we create a new process with *firefox* in its commandline. Lets drop the last line

```
ps -A | grep -i firefox | awk '{print $1, $4}' | head -n -1
```

```
44179   firefox
60731   /usr/local/lib/firefox/firefox
57651   /usr/local/lib/firefox/firefox
78402   /usr/local/lib/firefox/firefox
53121   /usr/local/lib/firefox/firefox
35192   /usr/local/lib/firefox/firefox
46680   /usr/local/lib/firefox/firefox
 9850   /usr/local/lib/firefox/firefox
40081   /usr/local/lib/firefox/firefox
44225   /usr/local/lib/firefox/firefox
 3307   /usr/local/lib/firefox/firefox
```

And really I'd just like a count of the number of processes

```
ps -A | grep -i firefox | awk '{print $1, $4}' | head -n -1 | wc -l
```

11

Other piping techniques

- The | pipe copies *standard output* to *standard input*...
- The > pipe copies *standard output* to a named file... (e.g. `ps -A >processes.txt`, see also the `tee` command)
- The >> pipe **appends** *standard output* to a *named file*...
- The < pipe reads a file *into* standard input... (e.g. `grep firefox <processes.txt`)
- The <<< pipe takes a string and *places it on standard input*
- You can even copy and merge streams if you know their file descriptors (e.g. appending `2>&1` to a command will run it with *standard error* merged into *standard output*)

- The `<(echo hello)` pipe is completely magical...
  - It runs the command in the parentheses outputting to a temporary file (descriptor)...
  - It replaces itself with the path to that file

So you can do things like:

```
diff <(echo Hello World) \
    <(echo Hello World | tr r R)
```

```
1c1
<     Hello   World
---
>     Hello   WoRld
```

Different shells

(Just use bash unless you care about *extreme* portability in which case use POSIX sh)

Typical Shells

sh	POSIX shell
bash	Bourne Again shell (default on Linux)
zsh	Z Shell (default on Macs), like bash but with more features
ksh	Korne shell (default on BSD)

Other shells

dash	simplified faster bash, used for booting on Linux
Busybox sh	simplified bash you find on embedded systems

Weird shells

fish	More usable shell (but different incompatible syntax)
elvish	Nicer syntax for scripting (but incompatible with POSIX)
nushell	Nicer output (but incompatible, and weird)

One last thing...

Suppose you want to run a shellscript as a network service.

- *Normally* you'd have to run a webserver and do some socket programming
- None of which is ever usually available for shell scripts

Inetd

`inetd` is *super server* that can launch any program when someone connects to a socket

- `stdin` will be the input from the network
- `stdout` will be sent back on the same socket

Check out `fingerd` for an ancient social network built like this...

Suppose we wanted to build some code

An *awful* lot of the things we do with a computer are about *format shifting*
We do this when we compile code:

- `cc -c library.c -o library.o`
- `cc hello.c library.o -o hello`

When we archive files:

- `zip -r coursework.zip coursework`

When we draw figures:

- `dot -Tpdf flowchart.dot -O flowchart.pdf`

Can we automate this?

YES!

We *could* write a shellscript and stick all the tasks in one place...

```
#! /usr/bin/env bash
cc -c library.c -o library.o
cc hello.c library.o -o hello
zip -r coursework.zip coursework
dot -Tpdf flowchart.dot -O flowchart.pdf
```

But can we do better than this?

- Do we really need to recompile the C program if only our flowchart has changed?
- Can we generalise build patterns?

Make

Make in an *ancient* tool for automating builds.

- Developed by *Stuart Feldman* in 1976
- Takes *rules* which tell you how to build files
- Then follows them to build the things you need!

Two main dialects of it (nowadays):

BSD Make More old fashioned, POSIX

GNU Make More featureful, default on Linux

In practice, unless your developing a BSD *every one* uses GNU Make

- If you're on a Mac, or BSD box install GNU Make and try `gmake` if things don't work

Makefiles

Rules for Make are placed into a *Makefile* and look like the following:

```
hello: hello.c library.o
	cc -o hello hello.c library.o

library.o: library.c
	cc -c -o library.o library.c

coursework.zip: coursework
	zip -r coursework.zip coursework

flowchart.pdf: flowchart.dot
	dot -Tpdf flowchart.dot -O flowchart.pdf
```

If you ask make to build `hello` it will figure out what it needs to do:

```
$ make hello
cc -c -o library.o library.c
cc -o hello hello.c library.o
```

First line specifies how to build *what* from *which* source files

- The rest of the **TAB** indented block is a shellscript (ish)

Making changes

If you alter files... Make is smart enough to only rerun the steps you need:
For example if you edit `hello.c` and rebuild:

```
$ make hello
cc -o hello hello.c library.o
```

But if you edit `library.c` it can figure out it needs to rebuild *everything*

```
$ make hello
cc -c -o library.o library.c
cc -o hello hello.c library.o
```

Phony targets

As well as rules for how to make files you can have *phony* targets that don't depend on files but just tell make what to do when they're run

Often a `Makefile` will include a phony:

`all` typically first rule in a file (or marked `.default`): depends on everything you'd like to build

`clean` deletes all generated files

`install` installs the program

```
$ make
cc -c -o library.o library.c
cc -o hello.o hello.c library.o
zip -r coursework.zip coursework
dot -Tpdf flowchart.dot -O flowchart.pdf
```

```
.PHONY: all clean

all: hello coursework.zip flowchart.pdf

clean:
	git clean -dfx

hello: hello.c library.o
	cc -o hello hello.c library.o

library.o: library.c
	cc -c -o library.o library.c

coursework.zip: coursework
	zip -r coursework.zip coursework

flowchart.pdf: flowchart.dot
	dot -Tpdf flowchart.dot -O flowchart.pdf
```

Pattern rules

(So far, everything *should* have worked in GNU and BSD Make... here on out we're in GNU land) What if we wanted to add an extra library to our `hello` programs? We could go and update the `Makefile` but its better to generalise!

```
CC=clang
CFLAGS=-Wall -O3

.PHONY: all clean

all: hello coursework.zip flowchart.pdf
clean:
	git clean -dfx

hello: hello.c library.o extra-library.o

%.o: %.c
	$(CC) $(CFLAGS) -c -o $@ $<

%: %.c
	$(CC) $(CFLAGS) -o $@ $<

%.zip: %
	zip -r $@ $<

%.pdf: %.dot
	dot -Tpdf $< -O $@
```

Implicit pattern rules

Actually because Make is so old, it knows about compiling C (and Fortran/Pascal...) code already:

```
.PHONY: all clean

all: hello coursework.zip flowchart.pdf
clean:
	git clean -dfx

hello: hello.c library.o extra-library.o

%.zip: %
	zip -r $@ $<

%.pdf: %.dot
	dot -Tpdf $< -O $@
```

Lets get even more general!

Suppose we wanted to add more figures... we could add dependencies on `all` to build them or...

```
.PHONY: all clean
figures=$(patsubst .dot,.pdf,$(wildcard *.dot))

all: hello coursework.zip ${figures}
clean:
	git clean -dfx

hello: hello.c library.o extra-library.o

%.zip: %
	zip -r $@ $<

%.pdf: %.dot
	dot -Tpdf $< -O $@
```

Make is crazy powerful

I love Make...

- I abuse it for compiling everything
- For distributing reproducible science studies
- For building and deploying websites

Pattern rules and the advanced stuff is neat...

- ...but if you never use it I won't be offended
- Make is one of those tools that you'll come back to ***again and again*** over your careers.
- ...and there's a ***bunch*** of tricks I haven't shown you `;-)`

Go and read the *GNU Make Manual*

- Its pretty good for a technical document

Just type `make`

When you get a bit of software... and you find a `Makefile` in there...
Just type `make`!

- (and make sure your projects build in the same way!)

(Actually often you'll have to type `./configure` then `make`)

- No I'm not going to teach you *autotools* don't worry!

Limitations of Make

I love `Make` but it has one *big* weakness

- Modern development makes extensive use of external libraries…

But *Make* is rubbish at dealing with them:

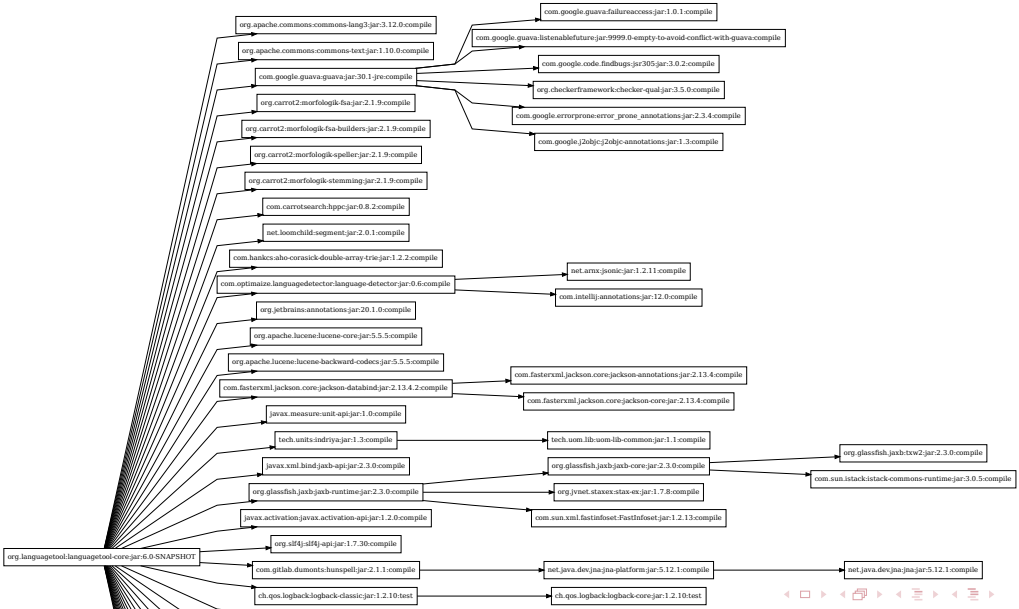
- Doesn't know how to fetch dependencies
- Doesn't track versions beyond *source is newer than object*

LanguageTool is a cool little Java grammar checker:

- How many libraries does *just the core* of the tool make use of?

```
mvn dependency:tree -D outputType=dot | dot -Tpdf
```

This is surely too many?



In the old days...

Traditionally you'd have to go download all the dependencies by hand...

- And then compile and install them
- Very tedious and error prone

So we automated it!

Modern build tooling

(Almost) every language comes with its own library management tooling

- Lets developers specify dependencies
- Tells compiler how to rebuild the project

...which means *for every language you use you need to learn its build tools*...

- Yay?

(Honestly, I still use *Make* but I'm old and cantankerous)

So now we have...

Commonlisp	ASDF and Quicklisp
Go	Gobuild
Haskell	Cabal
Java	Ant, Maven, Gradle...
JavaScript	NPM
Perl	CPAN
Python	Distutils and `requirements.txt`
R	CRAN
Ruby	Gem
Rust	Cargo
LaTeX	CTAN and TeXlive

...and *many* more.

And they're all different

Very little similarity between *any* of them.

- You need to learn the ones you use.
- We'll play in the labs with *Maven* for Java a little bit

Maven Quickstart

```
mkdir /tmp/src
cd /tmp/src
mvn archetype:generate \
   -DgroupId=uk.ac.bristol.cs \
   -DartifactId=hello \
   -DarchetypeArtifactId=maven-archetype-quickstart \
   -DinteractiveMode=false
```

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------< org.apache.maven:standalone-pom >-------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] --------------------------------[ pom ]---------------------------------
[INFO]
[INFO] >>> maven-archetype-plugin:3.2.1:generate (default-cli) > generate-sources @ standalone-pom >>>
[INFO]
[INFO] <<< maven-archetype-plugin:3.2.1:generate (default-cli) < generate-sources @ standalone-pom <<<
[INFO]
[INFO]
[INFO] --- maven-archetype-plugin:3.2.1:generate (default-cli) @ standalone-pom ---
[INFO] Generating project in Batch mode
[INFO] ----------------------------------------------------------------------------
[INFO] Using following parameters for creating project from Old (1.x) Archetype: maven-archetype-quickstart:1.0
[INFO] ----------------------------------------------------------------------------
[INFO] Parameter: basedir, Value: /tmp/src
[INFO] Parameter: package, Value: uk.ac.bristol.cs
[INFO] Parameter: groupId, Value: uk.ac.bristol.cs
[INFO] Parameter: artifactId, Value: hello
[INFO] Parameter: packageName, Value: uk.ac.bristol.cs
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] project created from Old (1.x) Archetype in dir: /tmp/src/hello
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  4.256 s
```

...and after spewing all that...

```
find /tmp/src -type f
```

- /tmp/src/hello/pom.xml
- /tmp/src/hello/src/main/java/uk/ac/bristol/cs/App.java
- /tmp/src/hello/src/test/java/uk/ac/bristol/cs/AppTest.java
- /tmp/src/hello/target/maven-status/maven-compiler-plugin/compile/default-compile/createdFiles.lst
- /tmp/src/hello/target/maven-status/maven-compiler-plugin/compile/default-compile/inputFiles.lst
- /tmp/src/hello/target/maven-status/maven-compiler-plugin/testCompile/default-testCompile/createdFiles.lst
- /tmp/src/hello/target/maven-status/maven-compiler-plugin/testCompile/default-testCompile/inputFiles.lst
- /tmp/src/hello/target/classes/uk/ac/bristol/cs/App.class
- /tmp/src/hello/target/test-classes/uk/ac/bristol/cs/AppTest.class
- /tmp/src/hello/target/surefire-reports/uk.ac.bristol.cs.AppTest.txt
- /tmp/src/hello/target/surefire-reports/TEST-uk.ac.bristol.cs.AppTest.xml
- /tmp/src/hello/target/maven-archiver/pom.properties
- /tmp/src/hello/target/hello-1.0-SNAPSHOT.jar

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instanc
       xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd"
  <modelVersion>4.0.0</modelVersion>
  <groupId>uk.ac.bristol.cs</groupId>
  <artifactId>hello</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>hello</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

And if we try and build...

```
mvn package
```

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------< uk.ac.bristol.cs:hello >------------------------
[INFO] Building hello 1.0-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ hello ---
[WARNING] Using platform encoding (UTF-8 actually) to copy filtered resources, i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory /tmp/src/hello/src/main/resources
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ hello ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ hello ---
[WARNING] Using platform encoding (UTF-8 actually) to copy filtered resources, i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory /tmp/src/hello/src/test/resources
[INFO]
[INFO] --- maven-compiler-plugin:3.1:testCompile (default-testCompile) @ hello ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ hello ---
[INFO] Surefire report directory: /tmp/src/hello/target/surefire-reports

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running uk.ac.bristol.cs.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.018 sec

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ hello ---
```

Other useful commands

`mvn test` run the test suite

`mvn install` install the JAR into your local JAR packages

`mvn clean` delete everything

And if I'm being a bit snarky…

`https://gradle.org` A *better* Java build tool

(That doesn't work everywhere and is much worse than Maven when you try and do more complex things…)

Wrap up

Language specific build tools exist

- You should probably use them
- (but I still use good ol' `make` a lot more)

Makefiles let you shift things between different filetypes

- And avoid rebuilding things that don't need to be rebuilt

Shellscripts let you automate *everything*

- You're gonna end up writing them in anger
- Laziness is good
- Run everything through shellcheck

Aside

Sometimes you'll find you pull a project and it uses a certain build system and you just know you're going to have to spend a day fighting it.
...please don't use CMake.