BASH

SECTION 3

HOW BASH PROCESSES COMMAND LINES

SECTION CHEAT SHEET

When Bash receives a command line, it will follow the following 6-step process to interpret its meaning and execute it.

STEP 1: TOKENISATION

During **tokenisation**, bash reads the command line for **unquoted metacharacters**, and uses them to divide the command line into **words** and **operators**.

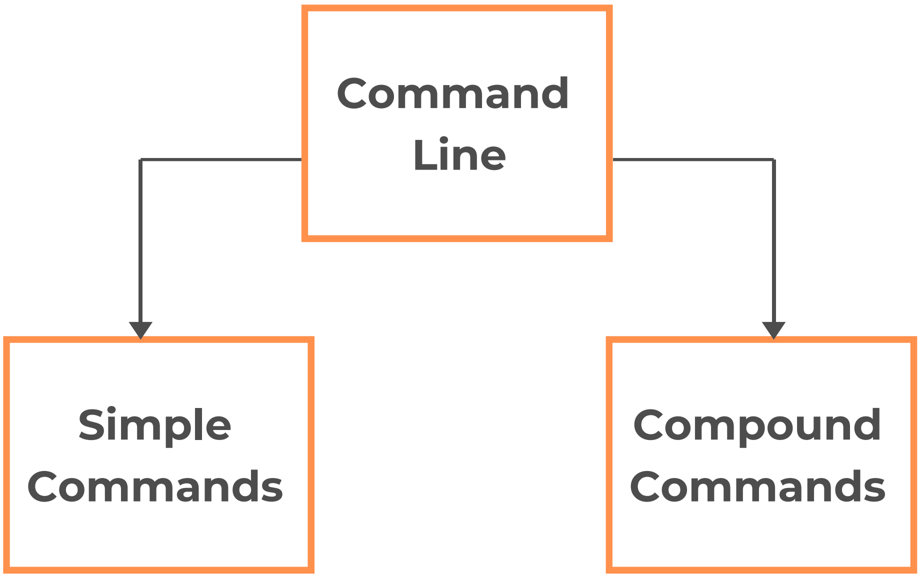
LIST OF METACHARACTERS
Space
Tab
Newline
|
&
;
(
)
<
>

WHAT ARE WORDS?

Words are tokens that **do not** contain any **unquoted metacharacters**

STEP 2: COMMAND IDENTIFICATION

Bash will then break the command line down into simple and compound commands.



SIMPLE COMMANDS

Simple commands are a set of words terminated by a control operator

- The first word is the command name.
- Subsequent words are taken as individual arguments to that command.

WHAT ARE OPERATORS?

Operators are tokens that contain at **least 1 unquoted metacharacter**

LIST OF CONTROL OPERATORS

- Newline
- I
- II
- &
- &&
- ;
- ;;
- ;&
- ;;&
- |&
- (
-)

LIST OF REDIRECTION OPERATORS

- <
- >
- <<
- >>
- <&
- >&
- >|
- <<-
- < >

EXAMPLE:

echo $name > out

echo $name > out -- Identifies unquoted metacharacters

echo $name > out -- Identifies Words & operators

EXAMPLES OF SIMPLE COMMANDS:

Example 1: echo a b c echo 1 2 3

echo a b c echo 1 2 3 - Tokenisation

echo a b c echo 1 2 3 - Interpreted as one simple command because there are no control operators.

Example 2: echo a b c ; echo 1 2 3

echo a b c ; echo 1 2 3

echo a b c ; echo 1 2 3

echo a b c ; echo 1 2 3

This is interpreted as two simple commands because there is a control operator that ends the first command.

Example 3: echo $name > out

echo $name > out -- Tokenisation

echo $name > out -- Found a redirection operator but no control operators

echo $name > out -- All interpreted as one simple command, including redirection operator

COMPOUND COMMANDS

These are bash’s programming constructs. They start with a reserved word and are terminated by the corresponding reserved word

COMPOUND COMMAND EXAMPLE:

Example:

```
if [[ 2 -gt 1 ]]; then
        echo “hello world”
fi
```

Note: We haven’t covered how to use compound commands yet -- we will cover them later in detail.

STEP 3: EXPANSIONS

Note 1: Earlier stages are given higher precedence than later ones.

Note 2: Expansions in the same stage are all given equal precedence and are simply processed from left to right.

THERE ARE 4 STAGES TO PROCESSING EXPANSIONS.

STAGE 1	Brace Expansion
STAGE 2	Parameter expansion Arithmetic expansion Command substitution Tilde expansion
STAGE 3	Word Splitting
STAGE 4	Globbing (aka filename expansion)

STAGE 1 - BRACE EXPANSION

Note: Brace expansion is processed as discussed in section 2. Please see the section 2 cheat sheet for more information

STAGE 2

- Parameter expansion
- Arithmetic expansion
- Command substitution
- Tilde expansion

Note: Each of these is processed as discussed in section 2. Please see the section 2 cheat sheet for more information

STAGE 3: WORD SPLITTING

After processing the preceding expansions, the shell will try to split the results of **unquoted parameter expansions, unquoted arithmetic expansions** and **unquoted command substitutions** into individual words.

Note 1: Word splitting is a very important step, because each word provided to a command is considered as an individual argument to the command (see the "Command Identification" step above).

Note 2: Word splitting doesn’t occur on the results of expansions that occurred inside double quotes.

Example 1 (Word Splitting)	Example 2 (No Word Splitting)
numbers=”1 2 3 4 5” touch $numbers touch 1 2 3 4 5 Result: 5 files created	numbers=”1 2 3 4 5” touch “$numbers” touch “1 2 3 4 5” Result: 1 file created called “1 2 3 4 5”

Note 3: Bash will split a word using the characters stored in the IFS variable, which by default contains space, tab, and newline.

Note 4: You can modify the IFS variable just like any other variable.

Note 5: Use **echo "${IFS@Q}"** to view what characters the IFS variable currently contains

For more information, see: GNU Bash Manual - Word Splitting

STAGE 4: GLOBBING (AKA FILENAME EXPANSION)

Upon reaching the globbing stage, bash scans each word for unquoted special pattern characters. These special pattern characters are *, ? and [.

Any word containing one of these characters is interpreted as a pattern, and replaced with a list of alphabetically-sorted filenames that match the pattern (if they exist).

BASIC GLOBBING PATTERN CHARACTER

? Matches any single character, but requires a character to be there.

***** Matches any string, regardless of length or content. Also matches empty strings

[Matches any one of the enclosed characters, but requires a character to be there.

[! Matches any single character *except* those enclosed in the brackets, but requires a character to be there.

Examples of Basic Globbing Patterns:

Consider the following example files:
filea.txt, fileb.txt, filec.txt, file1.txt, file2.txt, file3.txt, fileabc.txt, file123.txt

ls file?.txt	Will match all files with exactly one character between "file" and ".txt". In this case, this pattern will match all files except fileabc.txt and file123.txt
ls file??.txt	Will match all files with exactly two character between "file" and ".txt". In this case, this pattern would match none of the files
ls file???.txt	Will match all files with exactly three character between "file" and ".txt". In this case, this pattern will match fileabc.txt and file123.txt
ls file[abc].txt	Will match all files that include either a single “a”, “b”, or “c” between “file” and “.txt”. In this case, the pattern will match filea.txt, fileb.txt, and filec.txt
ls file[123].txt	Will match all files that include either a single “1”, “2”, or “3” between “file” and “.txt”. In this case, the pattern will match file1.txt, file2.txt, and file3.txt
	Will match all files that include any single character between “file” and “.txt” except an “a”, “b”, or “c” . In this case, the pattern will match file1.txt, file2.txt, and file3.txt
ls file*.txt	Will match all files with any number of characters (even none) between “file” and “.txt”. In this case, the pattern would match all of the example files.

CHARACTER CLASSES

To make it easier to construct ranges of characters within square brackets, bash makes several character classes available for use.

Note: When used in a pattern, character classes must themselves be placed inside square brackets

[:alpha:]✗ [[:alpha:]]✓

[:alpha:]

Includes all the letters of the alphabet, in both upper and lowercase

[:lower:]

Includes just lowercase letters

[:upper:]

Includes just uppercase letters

[:digit:]

Includes the numbers 0–9

[:alnum:]

Includes the numbers 0–9, and all upper and lowercase letters

[:punct:]

Includes all forms of punctuation

[:space:]

Includes all forms of whitespace, such as tab and space characters

[:word:]

Includes all uppercase and lowercase letters, as well as “_”

Examples of Character Class usage:

Consider the following example files:
filea.txt, fileb.txt, filec.txt, file1.txt, file2.txt, file3.txt, fileabc.txt, file123.txt

ls file[[:lower:]].txt	Will match all files with exactly one lowercase letter character between “file” and “.txt” In this case, this pattern will match filea.txt, fileb.txt and filec.txt
ls file[[:alnum:]].txt	Will match all files with exactly one character between “file” and “.txt” that is either an uppercase letter, a lowercase letter, or a number from 0-9. In this case, this pattern will match filea.txt, fileb.txt, filec.txt, file1.txt, file2.txt and file3.txt
ls file[[:digit:]].txt	Will match all files with exactly one character between “file” and “.txt” that is a number from 0-9. In this case, this pattern will match file1.txt, file2.txt and file3.txt
ls file[![:digit:]].txt	Will match all files with exactly one character between “file” and “.txt” that is not a number from 0-9. In this case, this pattern will match filea.txt, fileb.txt and filec.txt

Extended Globbing Patterns

Important: Use **shopt -s extglob** to enable extended globbing in bash scripts

Extended Pattern General Form:

```
S(pattern1 | pattern2 | ... | patternN)
```

Where **S** is one of the symbols below:

@	Happy if the pattern list matches once
+	Happy if the pattern list matches 1 or more times.
?	Happy if the pattern list matches 0 or one time.
*	Happy if the pattern list matches 0 or more times.
!	Happy if something except the pattern list matches.

Examples of Extended Globbing Patterns:

Consider the following example files:

```
touch london_july_2001_{001..100}.jpeg,
touch london_march_2004_{001..100}.png
touch paris_sept_2007_{001..100}.jpg
touch newyork_dec_2012_{001..100}.jpeg
```

ls *.@(jpeg|png|jpg)	This will list each file that has one of the given file extensions (jpeg, png & jpg) In this case, this pattern will match all the image files
ls @(london|paris)*.@(jpeg|png|jpg)	This will list each file that has one of the given file extensions (jpeg, png & jpg) and that were taken in london and paris. In this case, this pattern will match all the image files with the words “london” or “paris” at the beginning.
ls !(london|paris)*.@(jpeg|png|jpg)	This will list each file that has one of the given file extensions (jpeg, png & jpg) and that do not start with the word london or paris In this case, this pattern will match all the image files with the word newyork at the beginning.
paris+([[:word:]]).@(jpeg|png|jpg)	This will list each file that starts with the word “paris”, then is followed by a series of characters containing only numbers, letters and the “_” character, and finally ends with a file extension of .jpeg, .png or .jpg In this case, this pattern will match all the image files with the word paris at the beginning.

For more information, see: GNU Bash Manual - Pattern Matching

STEP 4: QUOTE REMOVAL

During quote removal, the shell removes all **unquoted backslashes, single quote** characters**,** and **double quote** characters **that did NOT result** from a shell expansion.

echo “hello”	The double quotes are removed because they are not quoted and do not result from a expansion Result: echo hello
echo ‘ “hello” ’	The backslashes are removed, because they are unquoted and do not result from an expansion. The double quotes are retained, however, because they are quoted by the single quotes that surround them. Result: echo “hello”
echo \”hello\”	The backslashes are removed, because they are unquoted and do not result from an expansion. The double quotes are retained, however, because they are quoted by their preceding backslashes. Result: echo “hello”
path=”C:\Users\Karen\Documents” echo $path	On line 2, the backslashes in the path are retained because they result from an expansion (i.e. the parameter expansion of the $path variable). Result: echo C:\Users\Karen\Documents

STEP 5: REDIRECTION

The shell then processes any redirection operators to determine where the standard input, standard output and standard error data streams for the command should connect to

Note 1: Not all commands use every data stream. The best way to find out what streams a command uses is to try it out, or to read its manual page

Note 2: A data stream can only connect to one location at a time

Note 3: Redirections are processed from left to right

Example Redirection Operators

command < file	Redirects the contents of file to the standard input of command.
command > file	Truncates file and then redirects standard output of command to it
command >> file	Appends standard output of command to file.
command 2> file	Truncates file and then redirects standard error of command to it
command 2>> file	Appends standard error of command to file
command &> file	Truncates file, and then redirects both standard output and standard error of command to it.
command &>> file	Appends both standard output and standard error of command to file.

For more information, see: GNU Bash Manual - Redirections

STEP 6: EXECUTE

At this stage the shell has completed its processing of the command line and it now executes the command that have resulted from all the above steps.

And You’re Done!!

CONGRATULATIONS