

Introduction to Bash scripting language

Day 1

Alessandro Sciarra

Z02 – Software Development Center

07.10.2019

Topics of the day

- | | | | |
|---|--------------------------------|---|----------------------------------|
| 1 | Inception and philosophy | 5 | Variables and special parameters |
| 2 | Commands and arguments | 6 | Shell expansion |
| 3 | Strings and types of commands | 7 | Globs and filename expansion |
| 4 | Quoting and special characters | 8 | Conditional blocks |

Inception and philosophy



Northern Light from the lighthouse of Reykjavík at -8°C

An often mistreated language

- Everybody uses Bash
- It is easy to know a bit of many commands
- Not so many Bash users (have time to) deepen into the details

The nature of Bash

As for many tools, it is common to just get stuff working, no matter how

An often mistreated language

- Everybody uses Bash
- It is easy to know a bit of many commands
- Not so many Bash users (have time to) deepen into the details

The nature of Bash

As for many tools, it is common to just get stuff working, no matter how



Important aspects to **always** keep in mind

- Use a clear, readable layout
- Avoid unnecessary commands
- A small, trivial script today might become large and complex tomorrow

An often mistreated language

- Everybody uses Bash
- It is easy to know a bit of many commands
- Not so many Bash users (have time to) deepen into the details

The nature of Bash

As for many tools, it is common to just get stuff working, no matter how

Before you get too excited

It is key that you remember, bash is a tool, a single tool in a huge toolbox of programs. Bash alone will only let you do basic things with files and other programs. You will need to understand all the other tools in the toolbox of your system. This knowledge is vast and will come slowly, it is important that you take the time to learn them well rather than try to get the basic idea of most and break a leg tomorrow (or more likely, your music archive or collection of family pictures).

Greg's Wiki

Using Bash

Bash in interactive mode: A prompt and a command line

Bash in non-interactive mode: Executing scripts

The prompt

- cool-prompt\$ ← shell compatible with the Bourne shell
- cool-prompt% ← C-shell (which is not covered here)
- cool-prompt# ← shell run as superuser (root)

Using Bash

Bash in interactive mode: A prompt and a command line

Bash in non-interactive mode: Executing scripts

The prompt

cool-prompt\$ ← shell compatible with the Bourne shell
cool-prompt% ← C-shell (which is not covered here)
cool-prompt# ← shell run as superuser (root)

```
1 $ man man      # Learn how to use and read the manual*
2 $ man apropos
3 $ help         # Get help for builtin commands
4 $ help echo
```

* Use  to quit the manual

Using Bash

Bash in interactive mode: A prompt and a command line

Bash in non-interactive mode: Executing scripts

The prompt

cool-prompt\$ ← shell compatible with the Bourne shell
cool-prompt% ← C-shell (which is not covered here)
cool-prompt# ← shell run as superuser (root)

Manual SYNOPSIS

bold text

type exactly as shown.

italic text

replace with appropriate argument.

[-abc]

any or all arguments within [] are optional.

-a|b

options delimited by | cannot be used together.

argument ...

argument is repeatable.

[expression] ...

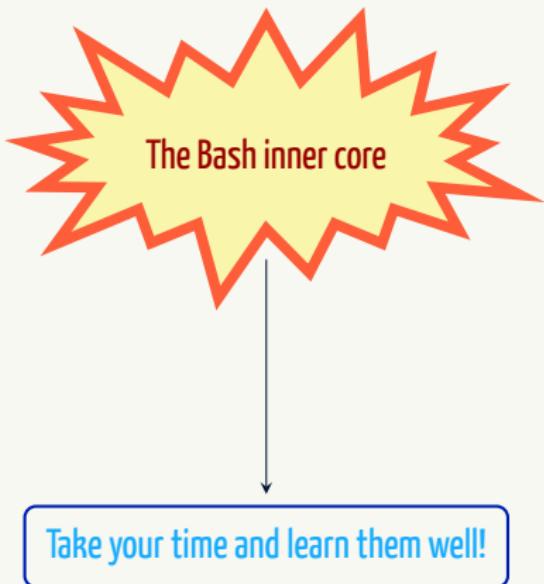
entire expression within [] is repeatable.

The goal of the day

- Arguments
- Quotes
- Shell parameters
- Special variables (e.g. IFS)
- Shell expansion
 - Parameters expansion
 - Word splitting
 - ...
- Globbing
- Regular expressions
- Brace expressions
- `if`, `test` and `[[`

The goal of the day

- Arguments
- Quotes
- Shell parameters
- Special variables (e.g. IFS)
- Shell expansion
 - Parameters expansion
 - Word splitting
 - ...
- Globbing
- Regular expressions
- Brace expressions
- `if`, `test` and `[[`



Commands and arguments



The Skaftafell natural park: Approaching the Vatnajökull

How does Bash interpret a line of code?

Spaces and Tabs



- Bash divides each line into words that are demarcated by a **whitespace character***.
- The first word of the line is the name of the command to be executed.
- All the remaining words become arguments to that command (options, filenames, etc.).

```
1 $ command arg1 arg2 arg3 arg4
2 $ command arg1 arg2 arg3 arg4
```

The amount of whitespace between arguments does not matter!

```
1 $ echo I am Clark Kent
2 I am Clark Kent                                # <- Same output
3 $ echo I      am     Clark                 Kent
4 I am Clark Kent                                # <- as here!
```

The first hurdle: spaces

How can a so innocent, invisible character hurt me? ...well...

- To a shell, whitespace is incredibly important
- Don't be fooled into thinking a space or tab more or less won't make much of a difference
- Whitespace is **vital** to allowing your shell to understand you

```
1 $ ls -1
2 The secret voice.mp3
3 secret
4 $ rm The secret voice.mp3 # rm gets 3 arguments, not 1!
5 rm: cannot remove 'The': No such file or directory
6 rm: cannot remove 'voice.mp3': No such file or directory
7 $ ls
8 The secret voice.mp3          # Where's your file 'secret'!?
```

`ls` and `rm` are commands to list the present working directory content and to remove files, respectively.

We will deepen into Word splitting later!

The first hurdle: spaces

How can a so innocent, invisible character hurt me? ...well...

- To a shell, whitespace is incredibly important
- Don't be fooled into thinking a space or tab more or less won't make much of a difference
- Whitespace is **vital** to allowing your shell to understand you

```
1 $ ls -1
2 The secret voice.mp3
3 secret
4 $ rm The secret voice.mp3 # rm gets 3 arguments, not 1!
5 rm: cannot remove 'The': No such file or directory
6 rm: cannot remove 'voice.mp3': No such file or directory
7 $ ls
8 The secret voice.mp3      # Where's your file 'secret'!?
```

`ls` and `rm` are commands to list the present working directory content and to remove files, respectively.

Side remark

Whitespaces in filenames should be avoided and replaced by underscores.

Everything would be much easier. But they are allowed... so deal with it!

Strings and types of commands



A must in the Golden Circle: Gullfoss

After all it is simple...

A scene from Toy Story featuring Woody and Buzz Lightyear. Woody, on the left, wears his signature brown plaid shirt and blue jeans. Buzz, on the right, wears his green and purple space ranger suit with the "SPACE HERO LIGHTYEAR" patch. Buzz is holding a small yellow star-shaped toy. The background shows a room with a chalkboard and some stars drawn on the wall.

STRINGS

STRINGS EVERYWHERE

Strings, strings everywhere

The term **string** refers to a sequence of characters which is treated as a single unit:

- The command's name is a string
- Each argument of a command is a string
- Variable names are strings
- The contents of variables are strings as well
- A filename is a string
- Most files contain strings

Strings do not have any intrinsic meaning

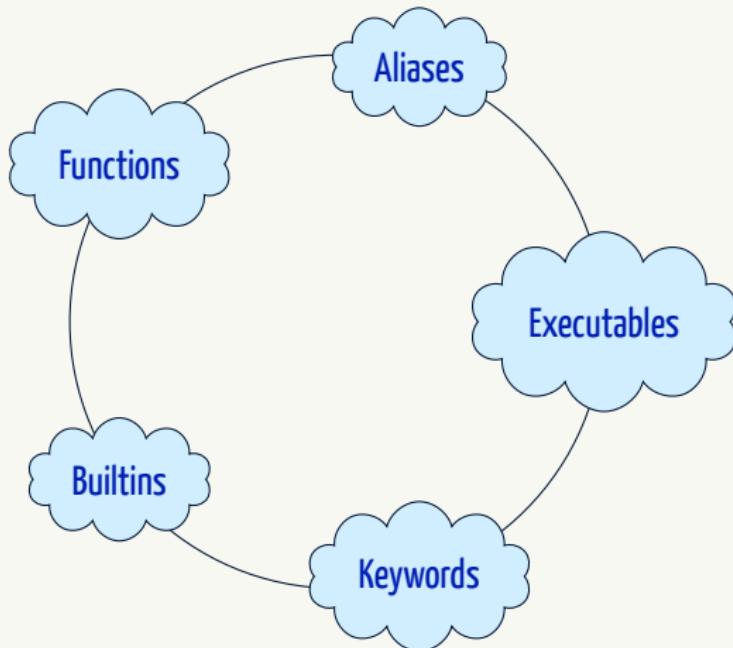
Their meaning is defined by how and where they are used.

We have all the responsibility

We need to be sure everything that needs to be separated is separated properly, and everything that needs to stay together stays together properly!

Types of commands

There are basically 5 different classes of commands



Types of commands

1 Aliases 2 Functions 3 Builtins 4 Keywords 5 Executables

- An alias is a rudimentary way of shortening a command
- It is a word that is mapped to a certain string
- They are only used in interactive shells and not in scripts
- They are limited in power; the replacement only happens in the first word
- An alias should effectively not do more than change the default options of a command
- For more complex tasks and more flexibility, use a function

```
1 $ help alias
2 alias: alias [-p] [name[=value] ... ]
3     Define or display aliases.
4
5 # [More information]
6
7 $ alias ls='ls --color=auto'
8 $ ls  # The command executed is, instead: ls --color=auto
```

Types of commands

1 Aliases

2 Functions

3 Builtins

4 Keywords

5 Executables

Functions are tricky!

They will be covered in depth later in the course

For the moment, few hints:

- Functions in Bash are more powerful than aliases and they are often the way to go
- Unlike aliases, they can be used in scripts, i.e. in non-interactive mode
- A function contains shell commands, and acts very much like a small script
- They can even take arguments and create local variables
- When a function is called, the commands in it are executed

Types of commands

1 Aliases

2 Functions

3 Builtins

4 Keywords

5 Executables

- Builtins are basic commands that Bash has built into it
- They can be thought as functions that are already provided
- We will learn about (or at least mention) most of them
- Keywords which are provided as builtins are nevertheless highlighted as keywords

:	complete	export	let	shift	umask
.	compgen	false	local	shopt	unalias
[continue	fc	logout	source	unset
alias	declare	fg	popd	suspend	until
bg	dirs	getopts	printf	test	wait
bind	disown	hash	pushd	times	while
break	echo	help	pwd	trap	
builtin	enable	history	read	true	
case	eval	if	readonly	type	
cd	exec	jobs	return	typeset	
command	exit	kill	set	ulimit	

Types of commands

1 Aliases

2 Functions

3 Builtins

4 Keywords

5 Executables

- Builtins are basic commands that Bash has built into it
- They can be thought as functions that are already provided
- We will learn about (or at least mention) most of them
- Keywords which are provided as builtins are nevertheless highlighted as keywords

:	complete	export	let	shift	umask
.	compgen	false	local	shopt	unalias
[continue	fc	logout	source	unset
alias	declare	fg	popd	suspend	until
bg	dirs	getopts	printf	test	wait
bind	disown	hash	pushd	times	while
break	echo	help	pwd	trap	
builtin	enable	history	read	true	
case	eval	if	readonly	type	
cd	exec	jobs	return	typeset	
command	exit	kill	set	ulimit	

Types of commands

1 Aliases

2 Functions

3 Builtins

4 Keywords

5 Executables

- Builtins are basic commands that Bash has built into it
- They can be thought as functions that are already provided
- We will learn about (or at least mention) most of them
- Keywords which are provided as builtins are nevertheless highlighted as **keywords**

:	complete	export	let	shift	umask
.	compgen	false	local	shopt	unalias
[continue	fc	logout	source	unset
alias	declare	fg	popd	suspend	until
bg	dirs	getopts	printf	test	wait
bind	disown	hash	pushd	times	while
break	echo	help	pwd	trap	
builtin	enable	history	read	true	
case	eval	if	readonly	type	
cd	exec	jobs	return	typeset	
command	exit	kill	set	ulimit	

Types of commands

1 Aliases

2 Functions

3 Builtins

4 Keywords

5 Executables

Keywords are like builtins, but **special parsing rules apply to them**

- Flow control constructs is achieved thanks to keywords
- We will explore all of them in detail {except coproc}

```
if      elif      esac      while      done      time      !      coproc
then    fi       for       until     in        {       [[
else    case     select    do        function   }       ]]]
```

For example:

```
1 $ [ a < b ]    # Ooops, < means here input redirection!
2 -bash: b: No such file or directory
3 $ [[ a < b ]] # The meaning of < changes within [[ ]]
```

Types of commands

1 Aliases 2 Functions 3 Builtins 4 Keywords 5 Executables

- Executable are the last type of command that Bash can execute
- They are also known as external commands or applications
- They are typically invoked by their name, on constraint that Bash is able to find them
- When a command is specified without a file path and it is **not an alias, a function, a builtin or a keyword**, Bash searches through the directories contained in the variable PATH
- The search is done from left to right and the first executable found is run

```
1 $ ls /home/sciarra/.local
2 ...     g++    ... # Version 8.3.0
3 $ ls /usr/bin
4 ...     g++    ... # Version 5.4.0
5 $ echo ${PATH}
6 /home/sciarra/.local/bin:/usr/bin:/bin
7 $ g++ -dumpversion
8 8.3.0
```

Bash scripts

- A script is a sequence of Bash commands in a file
- The commands are read and processed **in order**
- A new command is processed after the previous has **ended** {unless differently required}
- The first line of a script should be reserved for an **interpret directive** also called **hashbang** or **shebang**. This is used when the kernel executes a non-binary file. Use one of the two following alternatives:

```
#!/bin/bash          → or, preferably,          #!/usr/bin/env bash
```

The right directive has the benefit of looking for whatever the default version of the program is in your current environment (i.e. in PATH).

- Please, **do not use** `#!/bin/sh` as shebang, even if you might see it around on the web!
- Avoid giving your scripts a `.sh` extension. Do not use one or use `.bash`!

Sh is NOT Bash!

Bash itself is a sh-compatible shell however, the opposite is not true!

Bash scripts

- A script is a sequence of Bash commands in a file
- The commands are read and processed **in order**
- A new command is processed after the previous has **ended** {unless differently required}
- The first line of a script should be reserved for an **interpret directive** also called **hashbang** or **shebang**. This is used when the kernel executes a non-binary file. Use one of the two following alternatives:

`#!/bin/bash` or, preferably, `#!/usr/bin/env bash`

The right directive has the benefit of looking for whatever the default version of the program is in your current environment (i.e. in PATH).

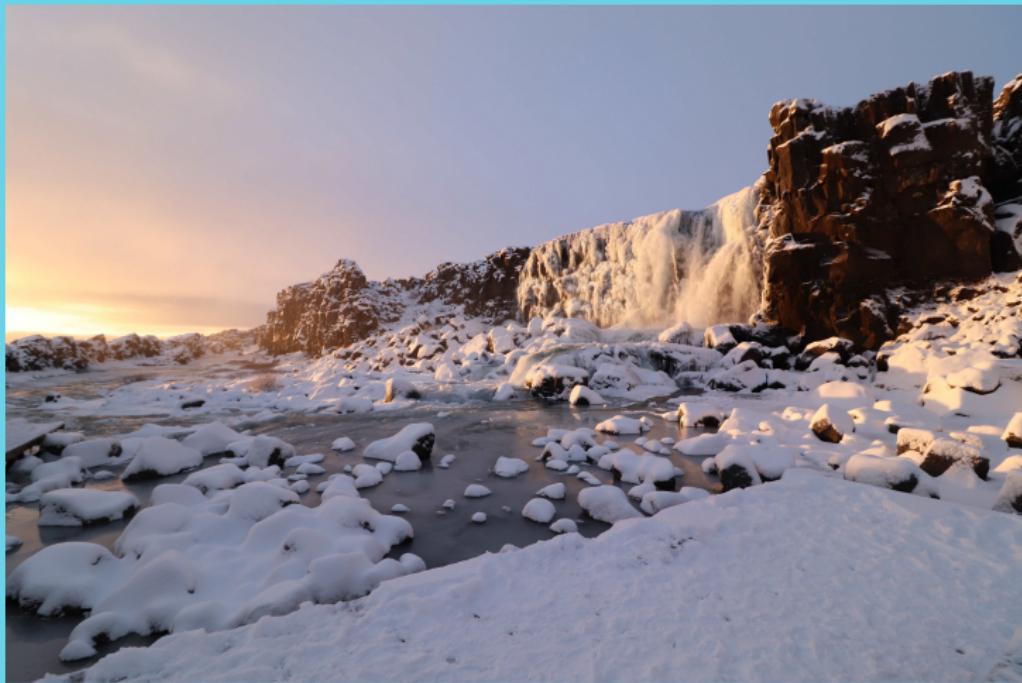
- Execute the script in either of the following ways:

```
1 $ bash myscript #The shebang is treated as a comment!
```

or

```
1 $ chmod +x myscript # Make the file executable  
2 $ ./myscript # Execute it, the shebang is used!
```

Quoting and special characters



The Þingvellir national park: Oxararfoss

Types of quoting

Single quotes Everything inside single quotes becomes a literal string

{The only character that you can't safely enclose in single quotes is a single quote.}

Double quotes Performed actions:

- Every substitution that begins with a dollar sign \$
- Backslash escaping
- The legacy ` ... ` (backticks) command substitution

No word splitting or filename expansion is performed!

Backticks ` ... ` is the legacy command substitution syntax

Deprecated in favor of \$ (...)

Backslash Putting \ in front of a metacharacter removes its special meaning

{This works inside double quotes, or in the absence of quotes. It does not work inside single quotes.}

\$ ' ... ' A Bash extension that prevents everything except backslash escaping
It also allows special backslash escape sequences like \n for newline

\$ " ... " Bash extension used for [localisation support](#), not covered here

Examples about quoting

```
1 $ echo I      am  Clark          Kent
I am Clark Kent
3 $ echo I"      "am"   "Clark"        "Kent           # Not nice
I      am  Clark          Kent
5 $ echo I\ \ \ \ am\ \ Clark\ \ \ \ \ Kent # Really!?
I      am  Clark          Kent
7 $ echo 'I      am  Clark          Kent'
I      am  Clark          Kent
9 $ echo "I      am  Clark          Kent"
I      am  Clark          Kent
```

```
11 $ echo 'PATH contains ${PATH}'
PATH contains ${PATH}
13 $ echo "PATH contains ${PATH}"
PATH contains /home/sciarra/.local/bin:/usr/bin:/bin
15 $ echo "PATH contains \$${PATH}"
PATH contains ${PATH}
```

Examples about quoting

```
17 $ ls *.tex
    Day_1.tex  Day_2.tex  Day_3.tex
19 $ ls '*.tex'
ls: cannot access '*.tex': No such file or directory
21 $ ls "*.*tex"
ls: cannot access '*.*tex': No such file or directory
23 $ echo "Hello\nWorld"
Hello\nWorld
25 $ echo $'Hello\nWorld'
Hello
27 World
```

I'm Too Lazy to Read, Just Tell Me What to Do

```
$ cp $file $destination          # WRONG
$ cp -- "$file" "$destination"  # Right
```

When in doubt, **double-quote every expansion** in your shell commands.
Generally use **double quotes** unless it makes more sense to use single quotes.

Special characters

Just a brief overview

- A **whitespace** is used by Bash to determine where words begin and end
The first word is the command name and additional words become arguments
 - \$ It introduces various types of **expansion**:
 - parameter expansion \${var}
 - command substitution \$(command)
 - arithmetic expansion \$((expression))
 - ' Single quotes
 - " Double quotes
 - \ Escape symbol
 - # It introduced a **comment** that extends to the end of the line.
Text after it is ignored by the shell. {In single and double quotes, # is just a #.}
- Already discussed
on slide 14

Special characters

Just a brief overview

- = The **assignment** symbol is used to assign a value to a variable
Whitespace is not allowed on either side of the = character

[[]] This **testing keyword** allows to evaluate a conditional expression to determine whether it is "true" or "false"

- ! The **negate keyword** reverses a test or an exit status

> >> < **Redirection** of a command's output or input to a file

- | The **pipeline** sends the output from one command to the input of another command
- ; **Command separator** of multiple commands that are on the same line

{ } An **inline group** allows to treat multiple commands as if they were one command
Convenient to be uses when Bash syntax requires only one command

() Another way to group commands, but in a **subshell group**
However, commands in () are executed in a subshell (a new process)
and this is often the way to avoid side effects on the current shell

Special characters

Just a brief overview

the 4 characters

+ - * /

(()) Within an **arithmetic expression**, mathematical operators are used for calculations

They can be used for

- variable assignments ((a = 1 + 4))
- tests evaluation ((a < b))

\$(()) Comparable to (()), but the expression is replaced with the result of its evaluation

* ? **Globbing** characters are wildcards which match parts of filenames

~ The tilde is a representation of a **home directory**

When alone or followed by a /, it means the current user's home directory

Otherwise, a username must be specified: `ls ~john/`

& When used at the end of a command, run the command in the **background**

The shell does not wait for it to complete

Special characters

Just a brief overview

the 4 characters

+ - * /

(()) Within an **arithmetic expression**, mathematical operators are used for calculations

They can be used for

- variable assignments ((a = 1 + 4))
- tests evaluation ((a < b))

\$(()) Comparable to (()), but the expression is replaced with the result of its evaluation

* ? **Globbing** characters are wildcards which match parts of filenames

~ The tilde is a representation of a **home directory**

When alone or followed by a /, it means the current user's home directory

Otherwise, a username must be specified: `ls ~john/`

& When used at the end of a command, run the command in the **background**

The shell does not wait for it to complete

More details later

We will come back to these special instructions with details and examples

Deprecated special character

- - Command substitution - **use \$() instead** ← *🔗 Advance reading*
- [] An alias for the old test command
Commonly used in POSIX shell scripts
Lacks many features of [[]] ← *🔗 Advanced reading*
- \$[] Arithmetic expression - **use \$(()) instead**
Simply do not use it, it dates back to 1990s!!!

Legacy code and portability

Unless you have very special requirement, try to stick to modern Bash features!

Variables and special parameters

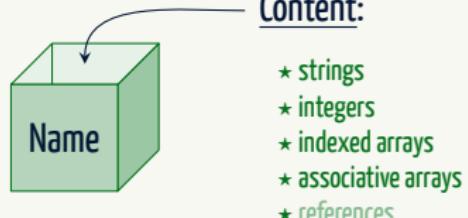


Seljalandsfoss: A 360° cascade

The first building block

Parameters come in two flavours:

- Variables (i.e. parameters with a name)
 - Created and updated by the user
 - Available in the environment
- Special parameters (later)
 - Read-only and pre-set by Bash



Content:

- ★ strings
- ★ integers
- ★ indexed arrays
- ★ associative arrays
- ★ references

The variable name (also referred to as an identifier)

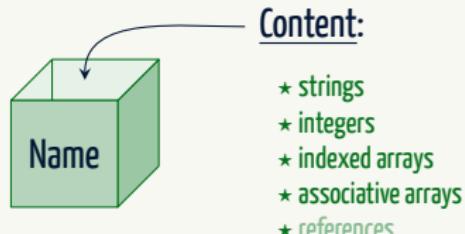
A word consisting only of

- letters, digits and underscores
- and beginning with a letter or an underscore

The first building block

Parameters come in two flavours:

- Variables (i.e. parameters with a name)
 - Created and updated by the user
 - Available in the environment
- Special parameters (later)
 - Read-only and pre-set by Bash



Content:

- ★ strings
- ★ integers
- ★ indexed arrays
- ★ associative arrays
- ★ references

Assignment

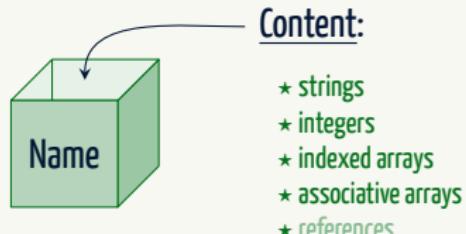
```
$ variableName=variableContent
```

- If not existing, the **global** variable **variableName** is created, and the content **variableContent** is put into it
- If existing, the content of **variableName** is set to **variableContent**
- If **variableName** exists and it is read-only, an error occurs

The first building block

Parameters come in two flavours:

- Variables (i.e. parameters with a name)
 - Created and updated by the user
 - Available in the environment
- Special parameters (later)
 - Read-only and pre-set by Bash



Content:

- ★ strings
- ★ integers
- ★ indexed arrays
- ★ associative arrays
- ★ references

Accessing the content: the parameter expansion

Use the \$ special character to tell Bash that you want to use the content of a variable

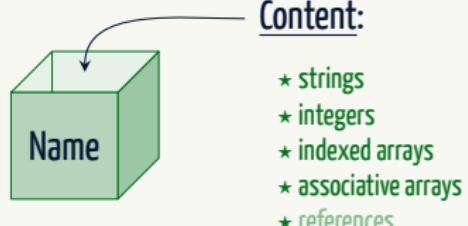
```
$ prefix='Day_'
$ day='Monday'
$ echo "${prefix}0.pdf are the slides for ${day}"
Day_0.pdf are the slides for Monday
```

Always using curly braces \${...} can be considered good programming practice!

The first building block

Parameters come in two flavours:

- Variables (i.e. parameters with a name)
 - Created and updated by the user
 - Available in the environment
- Special parameters (later)
 - Read-only and pre-set by Bash



Content:

- ★ strings
- ★ integers
- ★ indexed arrays
- ★ associative arrays
- ★ references

Expanding undeclared variable

Variables in Bash do not have to be declared, as they do in languages like C!
If you try to read an undeclared variable, the result is the empty string.

```
$ invisibleVariable='Hello'  
$ echo _${invisibelVariable}_  
-- # Is this magic?! Well, no...
```

By default, you get no warnings or errors!

Variables Types

Bash is not a typed language

It does have a few different types of variables, though!

It is more about activate particular rules when acting on variables.

Array: `declare -a variableName`

The variable is an array of strings

Associative array: `declare -A variableName`

The variable is an associative array of strings (v4.0 or higher)

Integer: `declare -i variableName`

The variable holds an integer

Assigning values to this variable automatically triggers Arithmetic Evaluation

Read only: `declare -r variableName`

The variable can no longer be modified or unset

Export: `declare -x variableName`

The variable is marked for export, i.e. it will be inherited by any child process

Variables Types

```
1 $ aVar=5; aVar+=2; echo "$aVar"; unset aVar
2 52
3 $ aVar=5; let aVar+=2; echo "$aVar"; unset aVar
4 7
5 $ declare -i aVar=5; aVar+=2; echo "$aVar"; unset aVar
6 7
7 $ aVar=5+2; echo "$aVar"; unset aVar
8 5+2
9 $ declare -i aVar=5+2; echo "$aVar"; unset aVar
10 7
11 $ declare -i aVar=5; aVar+=aVar; echo "$aVar"; unset aVar
12 10
13 $ declare -i aVar=5; aVar+="foo"; echo "$aVar"; unset aVar
14 10
15 # "foo" refers to the variable foo in arithmetic evaluation
```

The use of integer variables is exceedingly rare!

Most experienced shell programmers prefer to use explicit arithmetic commands
(with `((...))` or `let`) when they want to perform arithmetic!

Crucial to know (I)

...and not to forget!

```
#This is WRONG
$ variableName=variableContent    # spaces around = sign!
bash: variableName: command not found
```

- 1 Bash will not know that you are attempting to assign something
- 2 The parser will see `variableName` with no `=` and treat it as a command name
- 3 `=` and `variableContent` are then passed to it as arguments

If you think about it for a moment, it makes sense!

Crucial to know (II)

...and not to forget!

Attention!

After parameter expansion, Bash may still perform additional manipulations on the result!

```
1 $ today=Monday
2 $ echo "Today is ${today}"
3 Today is Monday
4 # Bash takes the content of the variable today
5 # and replaces ${today} by Monday. Equivalent to:
6 $ echo Today is Monday
7 Today is Monday
```

Everything seems to work as expected... but:

Crucial to know (II)

...and not to forget!

Attention!

After parameter expansion, Bash may still perform additional manipulations on the result!

```
1 $ today=Monday
2 $ echo "Today is ${today}"
3 Today is Monday
4 # Bash takes the content of the variable today
5 # and replaces ${today} by Monday. Equivalent to:
6 $ echo Today is Monday
7 Today is Monday
```

Everything seems to work as expected... but:

```
1 #This is probably not what you would like to do
2 $ song="My song.mp3"
3 $ rm ${song}
4 rm: My: No such file or directory
5 rm: song.mp3: No such file or directory
```

Crucial to know (II)

...and not to forget!

Attention!

After parameter expansion, Bash may still perform additional manipulations on the result!

Why did it not work?

```
1 #This is probably not what you would like to do
2 $ song="My song.mp3"
3 $ rm ${song}
4 rm: My: No such file or directory
5 rm: song.mp3: No such file or directory
```

Crucial to know (II)

...and not to forget!

Attention!

After parameter expansion, Bash may still perform additional manipulations on the result!

Why did it not work?

- 1 Bash replaced \${song} by its content
- 2 Word splitting occurred before the command was executed!
- 3 rm was run with 2 arguments (there is white space between them and it is not quoted!)

```
$ rm My song.mp3
```

```
1 #This is probably not what you would like to do
2 $ song="My song.mp3"
3 $ rm ${song}
4 rm: My: No such file or directory
5 rm: song.mp3: No such file or directory
```

Crucial to know (II)

...and not to forget!

Attention!

After parameter expansion, Bash may still perform additional manipulations on the result!

```
1 # Please, do not try to put quotes in variables!
2 $ song="\"My song.mp3\""
3 $ rm ${song}
4 rm: "My: No such file or directory
5 rm: song.mp3": No such file or directory
6 # Here the quotes contained in the variable song
7 # are literal characters and they are not interpreted
8 # as quotes when the rm command is run!!
```

How do we fix this?

Crucial to know (II)

...and not to forget!

Attention!

After parameter expansion, Bash may still perform additional manipulations on the result!

```
1 # Please, do not try to put quotes in variables!
2 $ song="\"My song.mp3\""
3 $ rm ${song}
4 rm: "My: No such file or directory
5 rm: song.mp3": No such file or directory
6 # Here the quotes contained in the variable song
7 # are literal characters and they are not interpreted
8 # as quotes when the rm command is run!!
9
10 # CORRECT WAY TO DO IT:
$ rm "${song}"
```

How do we fix this?

Remember to put double quotes around every parameter expansion!

Available Variables

🔗 Bash manual v5.0 pages 73-84

```
# Bourne Shell Variables (For some, Bash sets a default)
CDPATH HOME IFS MAIL MAILPATH OPTARG OPTIND PATH PS1 PS2
# Bash Variables (i.e. variables that are set or used by Bash)
BASH COMP_POINT HISTFILESIZE OSTYPE
BASHOPTS COMP_TYPE HISTIGNORE PIPESTATUS
BASHPID COMP_KEY HISTSIZE POSIXLY_CORRECT
BASH_ALIASES COMP_WORDS_BREAKS HISTTIMEFORMAT PPID
BASH_ARGC COMP_WORDS HOSTFILE PROMPT_COMMAND
BASH_ARGV COMPREPLY HOSTNAME PROMPT_DIRTRIM
BASH_ARGVO COPROC HOSTTYPE PS0
BASH_CMDS DIRSTACK IGNOREEOF PS3
BASH_COMMAND EMACS INPUTRC PS4
BASH_COMPAT ENV INSIDE_EMACS PWD
BASH_ENV EPOCHREALTIME LANG RANDOM
BASH_EXECUTION_STRING EPOCHSECONDS LC_ALL READLINE_LINE
BASH_LINENO EUID LC_COLLATE READLINE_POINT
BASH_LOADABLES_PATH EXECIGNORE LC_CTYPE REPLY
BASH_REMATCH FCEDIT LC_MESSAGES SECONDS
BASH_SOURCE FIGNORE LC_NUMERIC SHELL
BASH_SUBSHELL FUNCNAME LC_TIME SHELOPTS
BASH_VERSINFO FUNCNEST LINENO SHlvl
BASH_VERSION GLOBIGNORE LINES TIMEFORMAT
BASH_XTRACEFD GROUPS MACH_TYPE TMOUT
CHILD_MAX histchars MAILCHECK TMPDIR
COLUMNS HISTCMD MAPFILE UID
COMP_CWORD HISTCONTROL OLDPWD # And few more
COMP_LINE HISTFILE OPTERR
```

The Internal Field Separator

A very important Bash special variable {More on it later when we discuss word splitting in detail}

- By default, IFS is unset and acts as if it were set to <space><tab><newline>

```
#The following two lines have the same effect
$ IFS=$' \t\n'
$ unset IFS
```

- It is used to determine what characters to use as word splitting delimiters
- Therefore, you can use any of these forms of white space to delimit words
- The IFS variable comes into play also in special parameter \$* expansion!

Attention

It is crucial to understand the role of the IFS variable!

The Internal Field Separator

A very important Bash special variable {More on it later when we discuss word splitting in detail}

- By default, `IFS` is unset and acts as if it were set to `<space><tab><newline>`

```
#The following two lines have the same effect
$ IFS=$' \t\n'
$ unset IFS
```

- It is used to determine what characters to use as word splitting delimiters
- Therefore, you can use any of these forms of white space to delimit words
- The `IFS` variable comes into play also in special parameter `$*` expansion!

🔗 When and why not to use a custom `IFS`

It is important that you understand the danger of changing the way the shell behaves.

If you modify `IFS`, word splitting will happen in a non-default manner henceforth.

Some will recommend you save `IFS` and reset it to the default later on. Others will recommend to unset `IFS` after you're done with your custom word splitting.

Personally, I prefer to recommend you **to NOT modify `IFS` on the script level. Ever!**

Greg's wiki

The Special Parameters

🔗 Bash manual v5.0 section 3.4.2

- Contains the name, or the path, of the script. Do not rely on it or read the manual carefully!
- 1 2 ... **Positional Parameters** contain the arguments passed to the current script or function
 - * Expands to all the words of all the positional parameters
Double quoted, it expands to a single string containing them all, separated by the first character of the `IFS` variable
 - Expands to all the words of all the positional parameters
Double quoted, it expands to a list of them all as individual words
 - # Expands to the number of positional parameters that are currently set
 - Expands to the current shell option flags
 - ? Expands to the exit code of the most recently completed foreground command
 - \$ Expands to the PID (process ID number) of the current shell
 - ! Expands to the PID of the command most recently executed in the background
 - _ Expands to the last argument of the last command that was executed

The Special Parameters

🔗 Bash manual v5.0 section 3.4.2

- Contains the name, or the path, of the script. Do not rely on it or read the manual carefully!
- 1 2 ... **Positional Parameters** contain the arguments passed to the current script or function
 - * Expands to all the words of all the positional parameters
Double quoted, it expands to a single string containing them all, separated by the first character of the `IFS` variable
 - Expands to all the words of all the positional parameters
Double quoted, it expands to a list of them all as individual words
 - # Expands to the number of positional parameters that are currently set
 - Expands to the current shell option flags
 - ? Expands to the exit code of the most recently completed foreground command
 - \$ Expands to the PID (process ID number) of the current shell
 - ! Expands to the PID of the command most recently executed in the background
 - _ Expands to the last argument of the last command that was executed

Exercise
Sheet 1

Shell expansion



Danger: Sneaker waves!

Several kind of expansions

🔗 Bash manual v5.0 section 3.5

A line is split into tokens using unquoted metacharacters:

↳ \t \n | & ; () < >



1



2

At the same time!



3



4



Quote Removal

After the preceding expansions, all unquoted occurrences of the characters \, ', and " that did not result from one of the above expansions are removed.

Brace expansion

- It comes in two forms:
 - 1 [preamble]{comma separated list}[postscript]
 - 2 [preamble]{X..Y[.increment]}[postscript]
- [preamble] and [postscript] are optional
- [preamble] and [postscript] might contain other brace expansions
- Brace expansions can be nested and, if so, they work from the outside in
- X and Y are either characters or numbers
- increment is an optional integer (if omitted, it is +1 or -1 as appropriate)
- If X and Y are numbers, leading 0 are respected to force each term to have the same width
- If the brace expansion syntax is not respected, then brace expansion is not performed!

Result of the expansion

A space separated list of all combinations of [preamble] and [postscript] with the elements in the brace. In 2, the sequence in braces is at first completed going back to 1. Order is respected left to right. Multiple non-nested braces expand to all combinations.

Brace expansion

- It comes in two forms:

- 1 [preamble]{comma separated list}[postscript]
- 2 [preamble]{X..Y[..increment]}[postscript]

```
1 $ echo {a,b,c}
2   a b c
3 $ echo {a,b,c}.tex
4   a.tex b.tex c.tex
5 $ echo image.{jpg,png,pdf}
6   image.jpg image.png image.pdf
7 $ echo {1..8}
8   1 2 3 4 5 6 7 8
9 $ echo {8..1}
10  8 7 6 5 4 3 2 1
11 $ echo {1..8..3}
12  1 4 7
13 $ echo {a..e}
14  a b c d e
15 $ echo {e..a..2}
16  e c a
17 $ echo file_{01..3}.pdf
18  file_01.pdf file_02.pdf file_03.pdf #Note leading zeros!
```

Brace expansion

- It comes in two forms:

- 1 [preamble]{comma separated list}[postscript]
- 2 [preamble]{X..Y[.increment]}[postscript]

```
19 $ echo {A..C}{0..2}
AO A1 A2 BO B1 B2 CO C1 C2
21 $ echo {A..C}{{0..2}} # Here two independent expansions!
A B C 0 1 2
23 $ echo {in,out}{go,com}ing
ingoing incoming outgoing outcoming
25 $ echo {{A,E,I,O,U},{0..9}}
A E I O U 0 1 2 3 4 5 6 7 8 9
27 $ echo {a..z..x}
{a..z..x} # No brace expansion!
29 $ aVar=1; echo ${aVar..5}; unset aVar
{1..5} # No brace expansion!
31 $ echo {b,1..5}
b 1..5 # Not surprising, right?
33 $ echo {b,{1..5}}
b 1 2 3 4 5
35 $ echo {b,{1..5}}}} # Which of the last braces is kept?
b} 1} 2} 3} 4} 5}
```

Parameter expansion: Overview

- It is the most used form of expansion!
- Its basic syntax consists of a single dollar sign alone
- Braces after the dollar are only sometimes necessary
→ it is good practice to always use them: `$(...)`
- Basic form: `$(parameter)` → The value of parameter is substituted
- It is plenty of incredibly powerful modified forms of parameter expansion → You will learn them by using, but **keep in mind they exist!**
- General form: `$([prefix]parameter [postfix])`
- Not all (particular) cases are discussed here immediately
- More on parameter expansion in the future
 - namerefs and indirect expansion
 - arrays and associative arrays

Parameter expansion: Checking variables state

In each of the cases below, word is subject to tilde expansion, parameter expansion, command substitution, and arithmetic expansion!

`${parameter:-word}`

Use Default Value

Parameter unset or null: expansion of word is substituted

Otherwise: the value of parameter is substituted

`${parameter:=word}`

Assign Default Value

Parameter unset or null: expansion of word is assigned to parameter

Otherwise: the value of parameter remains unchanged

The value of parameter is then substituted

{ Positional parameters and special parameters may not be assigned to in this way }

`${parameter:?word}`

Exit with message

Parameter is null or unset: the expansion of word to the standard error

The shell, then, if it is not interactive, exits

Otherwise, the value of parameter is substituted

{ If word is not present, a default message is produced }

`${parameter:+word}`

Use Alternative value

Parameter is null or unset: nothing is substituted

Otherwise, the expansion of word is substituted

A different variant

Omitting the colon results in a test only for a parameter that is unset

Put another way, if the colon is omitted, the operator tests only for existence

Parameter expansion: Checking variables state

```
1 $ aVar="Hello"
# The variable aVar is set and not empty/null
3 $ echo "_${aVar}_"
_Hello_
5 $ echo "_${aVar-Goodbye}_ _${aVar:-Goodbye}_" # Use default Value
_Hello_ _Hello_
7 $ echo "_${aVar=Goodbye}_ _${aVar:=Goodbye}_" # Assign default Value
_Hello_ _Hello_
9 $ echo "_${aVar?Goodbye}_ _${aVar:?Goodbye}_" # Exit with message
_Hello_ _Hello_
11 $ echo "_${aVar+Goodbye}_ _${aVar:+Goodbye}_" # Use alternative value
_Goodbye_ _Goodbye_
13 $ aVar=""
# The variable aVar is now set BUT empty/null
15 $ echo "_${aVar}_"

--
17 $ echo "_${aVar-Goodbye}_ _${aVar:-Goodbye}_"
-- _Goodbye_
19 $ echo "_${aVar=Goodbye}_ _${aVar:=Goodbye}_"; aVar=""
-- _Goodbye_
21 $ echo "_${aVar?Goodbye}_ _${aVar:?Goodbye}_"
bash: aVar: Goodbye # Shell STOPS at second expansion and prints message!
23 $ echo "_${aVar?Goodbye}_"

--
25 $ echo "_${aVar+Goodbye}_ _${aVar:+Goodbye}_"
_Goodbye_ --
```

Parameter expansion: Checking variables state

```
27 $ unset aVar  
# The variable aVar is now UNSET  
29 $ echo "_${aVar}_"  
  
--  
31 $ echo "_${aVar-Goodbye}_ _${aVar:-Goodbye}_"  
_Goodbye_ _Goodbye_  
33 $ echo "_${aVar=Goodbye}_"; unset aVar  
_Goodbye_  
35 $ echo "_${aVar:=Goodbye}_"; unset aVar  
_Goodbye_  
37 $ echo "_${aVar?Goodbye}_"  
bash: aVar: Goodbye # Shell STOPS at expansion and prints message!  
39 $ echo "_${aVar:?Goodbye}_"  
bash: aVar: Goodbye # Shell STOPS at expansion and prints message!  
41 $ echo "_${aVar+Goodbye}_ _${aVar:+Goodbye}_"  
-- --
```

Ok, but is all this useful? When?

Ohhh yes! These kind of expansion are useful e.g. when it comes to check whether a variable is set, unset or set but empty. Sometimes there are alternatives, but sometimes you need exactly this!

Parameter expansion: String manipulation (I)

String Length: \${#parameter}

The length in characters of the value of parameter is substituted

Substring Expansion: \${parameter:offset:length}

Expands to up to length characters of parameter starting at the character specified by offset (0-indexed)

If :length is omitted, go all the way to the end

{If offset is negative, count backward from the end of parameter instead of forward from the beginning }

Parameter expansion: String manipulation (I)

String Length: \${#parameter}

The length in characters of the value of parameter is substituted

Substring Expansion: \${parameter:offset:length}

Expands to up to length characters of parameter starting at the character specified by offset (0-indexed)

If :length is omitted, go all the way to the end

{If offset is negative, count backward from the end of parameter instead of forward from the beginning }

```
1 $ aVar='01234567890abcdefg'
# Let us demonstrate a bit
3 $ echo "String ${aVar} is ${#aVar} characters long"
String 01234567890abcdefg is 19 characters long
5 $ echo "${aVar:7} and 0 length: \"${aVar:7:0}\""
7890abcdefg and 0 length: ""
7 $ echo "${aVar:7:2} and ${aVar:7:-2}"
78 and 7890abcdef
9 $ echo "${aVar: -7}" # Why do you need a space?
bcdefgh
11 $ echo "${aVar: -7:2} and ${aVar: -7:0} and ${aVar: -7:-2}"
bc and and bcdef
```

Parameter expansion: String manipulation (II)

Remove Smallest Prefix: `${parameter#pattern}`

The pattern is matched against the **beginning** of parameter
which is expanded with the **shortest** match deleted

Remove Largest Prefix: `${parameter##pattern}` As the previous, but the **longest** match is deleted

Remove Smallest Suffix: `${parameter%pattern}`

The pattern is matched against the **end** of parameter
which is expanded with the **shortest** match deleted

Remove Largest Suffix: `${parameter%%pattern}` As the previous, but the **longest** match is deleted

Parameter expansion: String manipulation (II)

Remove Smallest Prefix: `${parameter#pattern}`

The pattern is matched against the **beginning** of parameter
which is expanded with the **shortest** match deleted

Remove Largest Prefix: `${parameter##pattern}` As the previous, but the **longest** match is deleted

Remove Smallest Suffix: `${parameter%pattern}`

The pattern is matched against the **end** of parameter
which is expanded with the **shortest** match deleted

Remove Largest Suffix: `${parameter%%pattern}` As the previous, but the **longest** match is deleted

```
1 $ aVar='b1.2345_s0000_s1111'
  # Let us demonstrate a bit (* matches any characters)
3 $ echo "${aVar#*_} and ${aVar##*_}"
   s0000_s1111 and s1111
5 $ echo "${aVar%_*} and ${aVar%%_*}"
   b1.2345_s0000 and b1.2345
7 $ echo "${aVar#NoMatch} and ${aVar%NoMatch}"
   b1.2345_s0000_s1111 and b1.2345_s0000_s1111
```

Parameter expansion: String manipulation (III)

Replace first: \${parameter/pattern/string}

Results in the expanded value of parameter with the first (unanchored) match of pattern replaced by string

{ Assume null string when the /string part is absent }

Replace all: \${parameter//pattern/string}

As the previous, but **every** match is replaced.

Replace at start: \${parameter/#pattern/string}

As the first, but matched against the **beginning**

Replace at end: \${parameter/%pattern/string}

As the first, but matched against the **end**

Parameter expansion: String manipulation (III)

Replace first: \${parameter/pattern/string}

Results in the expanded value of parameter with the first (unanchored) match of pattern replaced by string

{ Assume null string when the /string part is absent }

Replace all: \${parameter//pattern/string}

As the previous, but **every** match is replaced.

Replace at start: \${parameter/#pattern/string}

As the first, but matched against the **beginning**

Replace at end: \${parameter/%pattern/string}

As the first, but matched against the **end**

```
1 $ aVar='123aa000aa321'
2 # Let us demonstrate a bit
3 $ echo "${aVar/aa/_}" and ${aVar//aa/_}"
4 123_000aa321 and 123_000_321
5 $ echo "${aVar/#1/_}" and ${aVar/%1/_}"
6 _23aa000aa321 and 123aa000aa32_
7 $ echo "${aVar/#/prefix}" and ${aVar%/postfix}"
8 prefix123aa000aa321 and 123aa000aa321postfix
```

Parameter expansion: String manipulation (IV)

Since Bash v4.0 (2009)

First uppercase: \${parameter^}

Results in the expanded value of parameter with the first character capitalised

All uppercase: \${parameter^^}

As the previous, but all characters are capitalised.

First lowercase: \${parameter,}

Results in the expanded value of parameter with the first character to lowercase

All lowercase: \${parameter,,}

As the previous, but all characters are transformed to lowercase.

Parameter expansion: String manipulation (IV)

Since Bash v4.0 (2009)

First uppercase: \${parameter^}

Results in the expanded value of parameter with the first character capitalised

All uppercase: \${parameter^^}

As the previous, but all characters are capitalised.

First lowercase: \${parameter,}

Results in the expanded value of parameter with the first character to lowercase

All lowercase: \${parameter,,}

As the previous, but all characters are transformed to lowercase.

```
1 $ aVar='hELLO, World!'
  # Let us demonstrate a bit
3 $ echo "${aVar^} - ${aVar^^}"
  HELLO, World! - HELLO, WORLD!
5 $ echo "${aVar,} - ${aVar,,}"
  hELLO, World! - hello, world!
```

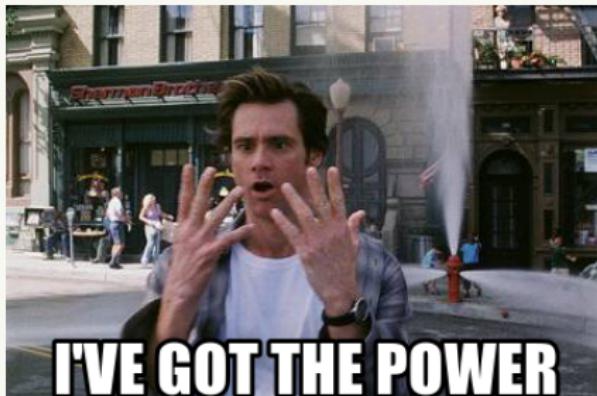
These types of expansion apply also to arrays: We will come back to this!

Parameter expansion: String manipulation summary

Good practice:

You may be tempted to use external applications such as `sed`, `awk`, `cut`, `perl` or others to modify your strings. Be aware that all of these require an extra process to be started, which in some cases can cause slowdowns.

Parameter Expansions are the perfect alternative!



Arithmetic expansion

- Arithmetic expansion syntax is `$((...))` and it can be nested
- Bash is only capable of integer arithmetic
- If you need to do a lot of non-integer arithmetic, then Bash is the wrong tool!
- Shell variables are allowed as operands
 - parameter expansion is performed before the expression is evaluated!
- Within an expression, shell variables may also be referenced by name without `${...}`
 - ⇒ strings are recursively interpreted as variable name until an integer value is found!
- A shell variable that is null or unset evaluates to
 - 0 when referenced by name without using the parameter expansion syntax
 - '' when referenced by name using the parameter expansion syntax
- Overflow is not checked but division by 0 gives an error
- Operators and their precedence are the same as in the C language
 -  Bash manual v5.0 section 6.5

Arithmetic expansion

```
1 $ echo $(( 22 / 3 ))
2
3 $ aVar=22; echo $(( ${aVar} / 3 )); unset aVar
4
5 $ aVar=22; echo $(( aVar / 3 )); unset aVar
6
7 $ echo $(( aVar / 3 ))      # aVar is unset, no ${} => 0 is used: $((0/3))
8
9 $ echo $(( ${aVar} / 3 ))  # aVar is unset, no ${} => '' is used: $(( /3 ))
bash: / 3 : syntax error: operand expected (error token is "/ 3 ")
10 $ echo $(( ))
11
12 $ aVar='Hello'; echo $(( aVar )) $(( ${aVar} )); unset aVar
13 0 0 # No variable 'Hello' existing
14 $ aVar='bVar'; bVar=5; echo $(( aVar )) $(( ${aVar} )); unset aVar bVar
15 5 5 # aVar is expanded to bVar, which is defined to 5
16 $ aVar='HelloMyLove'; echo $(( aVar )) $(( ${aVar} )); unset aVar
17 0 0 # No variable 'HelloMyLove' existing
18 $ aVar='Hello <3'; echo $(( aVar )) $(( ${aVar} )); unset aVar
19 1 1 # It makes sense, doesn't it?
20 $ echo $(( 2**51 * 4096 ))
21 -9223372036854775808 # No error, no overflow check!
```

Word splitting

...the evil of Bash!

When does it occur?

The shell scans the results of parameter expansion, command substitution, and arithmetic expansion that did not occur within double quotes for word splitting!

- The shell treats each character of `IFS` as a delimiter, and splits the results of the other expansions into words using these characters as field **terminators** (these are all dropped)
- Whitespace characters `\t\n` contained in `IFS` are called **IFS whitespaces**
- **IFS whitespaces at the beginning/end** of the results of previous expansions **are ignored**
- **Adjacent IFS whitespaces** are considered all together as **single field terminator**
- **Any other character** in `IFS` that is not an IFS whitespace **always delimits a field**
- If the value of `IFS` is null, no word splitting occurs
- Implicit null arguments, resulting from the expansion of empty parameters, are removed
- **Note that if no expansion occurs, no splitting is performed**

Word splitting

...the evil of Bash!

```
1 # The args command puts the arguments it gets in <...>
# NOT standard, I implemented it!
3 $ args Hello world "I am Alessandro"
3 args: <Hello> <world> <I am Alessandro>
5 $ aVar="This is a variable"; args ${aVar}
4 args: <This> <is> <a> <variable>
7 $ args "${aVar}"
1 args: <This is a variable>
9 $ ls
Day1.tex Day2.tex Day3.tex
11 $ args $(ls)
3 args: <Day1.tex> <Day2.tex> <Day3.tex>
13 $ aVar=" This is a variable "; args ${aVar}
4 args: <This> <is> <a> <variable>
15 $ aVar=" This is a variable "; args ${aVar}
4 args: <This> <is> <a> <variable>
17 $ aVar="/home/sciarra/Documents/Seminars"; args ${aVar}
1 args: </home/sciarra/Documents/Seminars/>
19 $ IFS='/'; args ${aVar}; unset IFS
5 args: <> <home> <sciarra> <Documents> <Seminars>
21 # Why is there an empty argument at the beginning?
# Why isn't there one at the end?
```

Word splitting

...the evil of Bash!

```
23 # Let us have a look to more complex situations
$ aVar="/home/sciarra/Documents/Seminars"
25 $ IFS='/' ; args ${aVar}; unset IFS
5 args: <      > <home> <sciarra> <Documents> <Seminars>
27 $ IFS=' \t'; args ${aVar}; unset IFS # Note the space in IFS
5 args: <> <home> <sciarra> <Documents> <Seminars>
29 # Remember $'\' special quoting to allow backslash escaping
$ aVar=$'\n  \n  xxx \t  \t xx  \t'
31 $ args ${aVar}
2 args: <xxx> <xx>
33 $ IFS=''; args ${aVar}; unset IFS
1 args: <

35
      xxxx      xx      >
37 $ IFS=$'\n'; args ${aVar}; unset IFS
2 args: <      > <  xxxx      xx      >
39 $ IFS=$'\n\t'; args ${aVar}; unset IFS
4 args: <      > <  xxxx > <      > <  xx      >
41 $ IFS=$'\n\t x'; args ${aVar}; unset IFS
5 args: <> <> <> <> <>
43 $ IFS=':'; args aaa:bbb; unset IFS aVar
1 args: <aaa:bbb> # It makes sense, right?
```

Word splitting: conclusion

When does NOT it occur?

Word splitting is not performed

- on expansions inside Bash keywords such as `[[...]]` and `case`;
- on expansions in assignments.

Thus, one does not need to quote anything in a command like these:

```
aVar=${bVar}; aVar=$(date)  
PATH=/usr/local/bin:${PATH}
```

Quoting anyway will not break anything, so if in doubt, quote!

Remember:

Word Splitting is performed after Parameter Expansion has been performed!

That means that it is absolutely vital that we quote our Parameter Expansions, in case they may expand values that contain syntactical whitespace which will then be word-split.

Other expansions

Before Word Splitting:

Tilde expansion: Easy to understand → [🔗 Bash manual v5.0 section 3.5.2](#)

Process substitution: Useful, but not discussed here → [🔗 Bash manual v5.0 section 3.5.6](#)
It takes the form <(...)> or >(...)

Command substitution: It occurs when a command is enclosed as \$(...)
Do not use the deprecated syntax ````

Try to read about them alone

The Bash manual is a very good source, you can start reading there!

[🔗 More on process substitution](#)

After Word Splitting:

Filename expansion: Discussed in a separate section about globbing

Globs and filename expansion



One of the edges of the Vatnajökull (the largest glacier in Europe)

Patterns in Bash

Definition

A pattern is a string with a special format designed to match filenames, or to check, classify or validate data strings.

Bash offers three different kinds of **pattern matching**:

- 1 **Globs** {Used also in patterns of Parameter Expansion}
- 2 **Extended Globs**
- 3 **Regular expression** {Since Bash v3.0}



**Pattern matching done
in Filename expansion**

{ Cannot be used to select filenames! }

Patterns in Bash

Definition

A pattern is a string with a special format designed to match filenames, or to check, classify or validate data strings.

Bash offers three different kinds of **pattern matching**:

- 1 Globs {Used also in patterns of Parameter Expansion}
- 2 Extended Globs
- 3 Regular expression {Since Bash v3.0}



Pattern matching done
in Filename expansion

{ Cannot be used to select filenames! }



We will discuss this later with conditional blocks

Filename Expansion

Glob patterns

How does it work?

If a character among *, ?, [appears, then the word is regarded as a pattern, and replaced with an alphabetically sorted list of filenames matching the pattern

- Metacharacters which might be used in glob patterns and undergo Filename Expansion:
 - * Matches any string, including the null string
 - ? Matches any single character
 - [...] Matches any one of the enclosed characters
- Globs are implicitly **anchored at both ends**, i.e. a glob must match a whole string
- When matching filenames, the * and ? characters cannot match a slash (/)
- When matching patterns, the / restriction is removed

The [...] pattern

- There are some special rules which it might be interesting to know
- A pair of characters separated by a hyphen denotes a range expression
 - any character that falls between those two characters, inclusive, is matched
- If the first character following the [is a ! or a ^ then any character not enclosed is matched
- A - may be matched by including it as the first or last character in the set
- A] may be matched by including it as the first character in the set
- Within [...], character classes can be specified using the syntax [:class:], where class is one of the *character classes* defined in the POSIX standard:

alnum	ascii	cntrl	graph	print	space	word
alpha	blank	digit	lower	punct	upper	xdigit

For example, [[:digit:]] or [[:blank:]]

Globs examples

```
1 $ ls
Bash.pdf      Day_1.tex      Day_2.tex      Day_3.tex      Day_extra.tex
3 $ echo *
Bash.pdf Day_1.tex Day_2.tex Day_3.tex Day_extra.tex
5 $ echo Day*
Day_1.tex Day_2.tex Day_3.tex Day_extra.tex
7 $ echo *.pdf
Bash.pdf
9 $ echo *_?.tex
Day_1.tex Day_2.tex Day_3.tex
11 $ echo *_[12]*
Day_1.tex Day_2.tex
13 $ echo *_[0-9]*
Day_1.tex Day_2.tex Day_3.tex
15 $ echo *_[:digit:]*
Day_1.tex Day_2.tex Day_3.tex
17 $ echo '*' # As already said, quotes prevent globbing
*
19 $ echo "*_[:digit:]*"
*_[:digit:]*
```

Globs examples

```
21 $ aVar='Day3_Day2_Day1'; echo "${aVar##*[!2]}"; unset aVar  
_Day1  
23 $ aVar='Day3_Day2_Day1'; echo "${aVar%*[!123]}"; unset aVar  
Day  
25 $ echo /*/bin    # For filenames, * does not match /  
/usr/bin  
27 $ echo /*/*/bin  
/usr/local/bin  
29 $ aVar='/home/sciarra/Documents'; echo "${aVar##*/}"  
Documents      # For strings, it does!  
31 $ echo "${aVar%/*}"; unset aVar  
/home/sciarra  
33 # This different behaviour is quite natural, indeed
```

Good practice:

You should always use globs instead of `ls` (or similar) to enumerate files. Globs will always expand safely and minimise the risk for bugs. You can sometimes end up with some very weird filenames. Most scripts aren't tested against all the odd cases that they may end up being used with. Don't let your script be one of those!

Globs examples

```
33 # Bash performs Filename Expansions AFTER Word Splitting
# => filenames generated by a glob will not be split!
35 $ touch "a b.txt"; ls
a b.txt
37 $ rm *; ls # Here * is expanded into "a b.txt"
              # => rm run correctly with one argument only!
39 $ touch "a b.txt"; rm $(ls) # WRONG!
rm: cannot remove 'a': No such file or directory
41 rm: cannot remove 'b.txt': No such file or directory
# Please, do not be tempted by using quotes here...
43 $ rm "$(ls)" # AAAARGHH! It works, but try this:
$ touch "a b.txt" c.txt; rm "$(ls)"
45 rm: cannot remove 'a b.txt'$'\n'c.txt': No such file or dir
```

Good practice:

You should always use globs instead of `ls` (or similar) to enumerate files. Globs will always expand safely and minimise the risk for bugs. You can sometimes end up with some very weird filenames. Most scripts aren't tested against all the odd cases that they may end up being used with. Don't let your script be one of those!

Extended Globs

This feature is turned off by default (in scripts)

You can turn it on with the `shopt` command, which is used to toggle shell options:

```
$ shopt -s extglob
```

In the following description, a pattern-list is a list of one or more patterns separated by a | :

- ?(pattern-list) Matches zero or one occurrence of the given patterns
- *(pattern-list) Matches zero or more occurrences of the given patterns
- +(pattern-list) Matches one or more occurrences of the given patterns
- @(pattern-list) Matches one of the given patterns
- !(pattern-list) Matches anything except one of the given patterns

Extended Globs

This feature is turned off by default (in scripts)

You can turn it on with the `shopt` command, which is used to toggle shell options:

```
$ shopt -s extglob
```

In the following description, a pattern-list is a list of one or more patterns separated by a `|`:

- ?(pattern-list) Matches zero or one occurrence of the given patterns
- *(pattern-list) Matches zero or more occurrences of the given patterns
- +(pattern-list) Matches one or more occurrences of the given patterns
- @(pattern-list) Matches one of the given patterns
- !(pattern-list) Matches anything except one of the given patterns

```
1 $ ls
names.txt    tokyo.jpg    california.bmp
3 $ echo "!(*jpg|*bmp)"
names.txt
```

Conditional blocks



The bridge between two continents

Exit code

Every command in Bash terminates with an exit code:

`$?` Shows the exit code of the last foreground process that terminated

It is a 8-bit integer $\$? \in \{0, \dots, 255\}$

{Indeed, only the least significant 8 bits count}

`0` Denotes success

`$\neq 0$` Denotes failure, in general the meaning is up to the command

`1` Miscellaneous errors

`2` Misuse of shell builtins

`126` Command invoked cannot execute

`127` "command not found" error

`128` Invalid argument to exit

`128 + n` Fatal error signal "`n`" {For example, `130` means that the script terminated by Control-C}

Which exit code should I use?

Avoid reserved one, be consistent!

The if keyword

- Since it is a keyword, it requires a precise syntax (not a surprise)
- It executes a command (or a set of commands) and checks that command's exit code to see whether it was successful
- Different layouts possible, choose a readable one!

```
if(COMMAND
then
    # COMMAND's exit code was 0
else
    # COMMAND's exit code was different from 0
fi

if(COMMAND; then
    # COMMAND's exit code was 0
elif(ANOTHER_COMMAND
    # ANOTHER_COMMAND's exit code was 0
else
    # ANOTHER_COMMAND's exit code was different from 0
fi
```

The command in a conditional block

- In principle it can be any command
- There are specifically designed commands to test things:
 - `test` A normal command that reads its arguments and does some checks with them. The `[]` variant requires a `] as last argument.`
 - `[[` A special shell keyword that offers more versatility like `pattern matching` and `regex support`
- Multiple commands can be concatenated using control operators `&&` and `||`

Which syntax should I prefer?

Whenever you are making a `Bash` script, you should always use `[[` rather than `[]`. If portability is an issue, e.g. you are writing a `shell` script, you should use `[]`, because it is far more portable.

The test command and its friend [

- e FILE True if file exists
- f FILE True if file is a regular file (not a directory or device file)
- d FILE True if file is a directory
- s FILE True if file exists and is not empty
- z STRING True if the string is empty (it's length is zero)
- n STRING True if the string is not empty (it's length is not zero)
- v VARIABLE True if the shell variable is set (has been assigned a value)*

```
1 $ ls -d */  
TeX  
3 $ if [ -d 'TeX' ]; then echo 'YES'; else echo 'NO'; fi  
YES  
5 $ if [ -e 'TeX' ]; then echo 'YES'; else echo 'NO'; fi  
YES  
7 $ if [ -s 'TeX' ]; then echo 'YES'; else echo 'NO'; fi  
YES  
9 $ if [ -f 'TeX' ]; then echo 'YES'; else echo 'NO'; fi  
NO
```

The test command and its friend [

STRING = STRING True if the first string is identical to the second
STRING != STRING True if the first string is not identical to the second
STRING < STRING True if the first string sorts before the second
STRING > STRING True if the first string sorts after the second
! EXPR Inverts the result of the expression (logical NOT)

```
1 $ aVar="Kal El"
$ bVar="Clark Kent"
3 $ [ ${aVar} = ${aVar} ]
bash: [: too many arguments
5 $ if [ "${aVar}" = "${aVar}" ]; then echo 'YES'; else echo 'NO'; fi
YES
7 $ if [ "${aVar}" > "${bVar}" ]; then echo 'YES'; else echo 'NO'; fi
NO
9 $ if [ 319 < 7 ]; then echo 'YES'; else echo 'NO'; fi
YES # 319 is < than 7 but it is not less than 7...
11 $ unset aVar bVar
```

The test command and its friend [

INT -eq INT True if both integers are identical.

INT -ne INT True if the integers are not identical.

INT -lt INT True if the first integer is less than the second.

INT -gt INT True if the first integer is greater than the second.

INT -le INT True if the first integer is less than or equal to the second.

INT -ge INT True if the first integer is greater than or equal to the second.

```
1 $ if [ 319 -lt 7 ]; then echo 'YES'; else echo 'NO'; fi
  NO
2 $ if [ 7 -ne 7 ]; then echo 'YES'; else echo 'NO'; fi
  NO
3 $ if [ 7 -eq 7 ]; then echo 'YES'; else echo 'NO'; fi
  YES
4 $ if [ 7 -gt 7 ]; then echo 'YES'; else echo 'NO'; fi
  NO
5 $ if [ 7 -ge 7 ]; then echo 'YES'; else echo 'NO'; fi
  YES
```

The [[keyword

- It supports the tests supported by the `test` and `[]` commands*
- String equality (`=` or `==`) and inequality (`!=`) comparison is changed to **pattern matching** by default; use quotes to perform a string comparison
- **It does not allow word-splitting of its arguments!**
- However, be aware that simple strings still have to be quoted properly
- Few more additional tests supported:

`STRING =~ REGEX` True if the string matches the regex pattern

`(EXPR)` Parentheses can be used to change the evaluation precedence

`EXPR && EXPR` True if both expressions are true (logical AND)

{ it does not evaluate the second expression if the first already turns out to be false }

`EXPR || EXPR` True if either expression is true (logical OR)

{ it does not evaluate the second expression if the first already turns out to be true }

- The alphabetically sorted test operators (`<` and `>`) do not need to be escaped

* Except `EXPR -a EXPR` and `EXPR -o EXPR` which should in any case not be used!

The [[keyword

```
1 $ aVar="Day_1.tex"; bVar='Hello world'
# Significative examples
3 $ if [[ ${aVar} = *.tex ]]; then echo 'YES'; else echo 'NO'; fi
YES
5 $ if [[ ${aVar} = ".*.tex" ]]; then echo 'YES'; else echo 'NO'; fi
NO
7 $ if [[ ${aVar} = *.tex && ${aVar} = Day_?.tex ]]; then echo 'YES'; else echo 'NO'; fi
YES
9 # Simple strings still have to be quoted properly!
$ if [[ ${bVar} = Hello world ]]; then echo 'YES'; else echo 'NO'; fi
11 bash: syntax error in conditional expression
bash: syntax error near `world'
13 $ if [[ ${bVar} = 'Hello world' ]]; then echo 'YES'; else echo 'NO'; fi
YES # No word splitting occurred!
15 $ if [[ ${emptyVar} = '' ]]; then echo 'YES'; else echo 'NO'; fi
YES
17 $ if [ ${emptyVar} = '' ]; then echo 'YES'; else echo 'NO'; fi
bash: [: =: unary operator expected
19 NO
# Quotes necessary to use test [ command to check for empty variables
21 $ if [ "${emptyVar}" = '' ]; then echo 'YES'; else echo 'NO'; fi
YES
23 $ unset aVar bVar
```

Control Operators

`&&` Used to build AND lists: Run one command only if another exited successfully

```
COMMAND_1 && COMMAND_2  
# Equivalent to  
if COMMAND_1; then COMMAND_2; fi
```

`||` Used to build OR lists: Run one command only if another exited unsuccessfully

```
COMMAND_1 || COMMAND_2  
# Equivalent to  
if ! COMMAND_1; then COMMAND_2; fi
```

Do not get overzealous with conditional operators

They can make your script hard to understand and sometimes you need fancy grouping to get your logic right!

Control Operators

```
# The use of control operators is handy in simple cases
[[ $PATH ]] && echo 'PATH variable set and non empty!'
# Mostly they are used in script
rm file || echo 'Could not delete file!'
mkdir TeX && cd TeX
source AuxiliaryOps.bash || exit 1
# Avoid complicate one-liners which might easily be wrong!
grep -q goodword "$file" && ! grep -q badword "$file" && rm
"$file" || echo "Couldn't delete: $file"
```

What happens if the first `grep` fails (sets the exit status to 1)?

Do not get overzealous with conditional operators

They can make your script hard to understand and sometimes you need fancy grouping to get your logic right!

Control Operators

```
# The use of control operators is handy in simple cases
[[ $PATH ]] && echo 'PATH variable set and non empty!'
# Mostly they are used in script
rm file || echo 'Could not delete file!'
mkdir TeX && cd TeX
source AuxiliaryOps.bash || exit 1
# Avoid complicate one-liners which might easily be wrong!
grep -q goodword "$file" && ! grep -q badword "$file" && rm
"$file" || echo "Couldn't delete: $file"
```

What happens if the first grep fails (sets the exit status to 1)?

```
grep -q goodword "$file" && ! grep -q badword "$file" && {
    rm "$file" || echo "Couldn't delete: $file"; }
```

Do not get overzealous with conditional operators

They can make your script hard to understand and sometimes you need fancy grouping to get your logic right!

Regular expressions in Bash

- Regular expressions are similar to Glob Patterns, but not for filename matching
- Since v3.0, Bash supports the `=~` operator to the `[]` keyword
- This operator matches the string that comes before it against the regular expression pattern that follows it and it returns
 - 0 when the string matches the pattern
 - 1 If the string does not match the pattern
 - 2 In case the pattern's syntax is invalid
- Bash uses the Extended Regular Expression dialect
 - o  POSIX standard
 - o  The Premier website about Regular Expressions
- Regular Expression patterns that use capturing groups (parentheses) will have their captured strings assigned to the `BASH_REMATCH` variable (array) for later retrieval

Regular expressions in Bash

```
1 $ echo "${LANG}"
en_GB.utf8
3 $ langRegex='(..)_(..)'
$ if [[ ${LANG} =~ ${langRegex} ]]
5 > then
>     echo "Your country code (ISO 3166-1-alpha-2) is ${BASH_REMATCH[2]}."
7 >     echo "Your language code (ISO 639-1) is ${BASH_REMATCH[1]}."
> else
9 >     echo "Your locale was not recognised"
> fi
11 Your country code (ISO 3166-1-alpha-2) is GB.
Your language code (ISO 639-1) is en.
```

General remarks

- The best way to always be compatible is to put your regex in a variable and expand that variable in `[[` without quotes, as we showed above.
- The pattern matching achieved by the `=` and `!=` operators can be achieved using regex: **Find your way!**