Introduction to Bash scripting language

Day 1

Alessandro Sciarra

Z02 – Software Development Center

17 February 2025

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

AxelKrypton

CRC-TR 211
Strong-interaction matter under extreme conditions

Topics of the day

Prelude

1 Inception and philosophy

2 Commands and arguments

3 Strings and types of commands

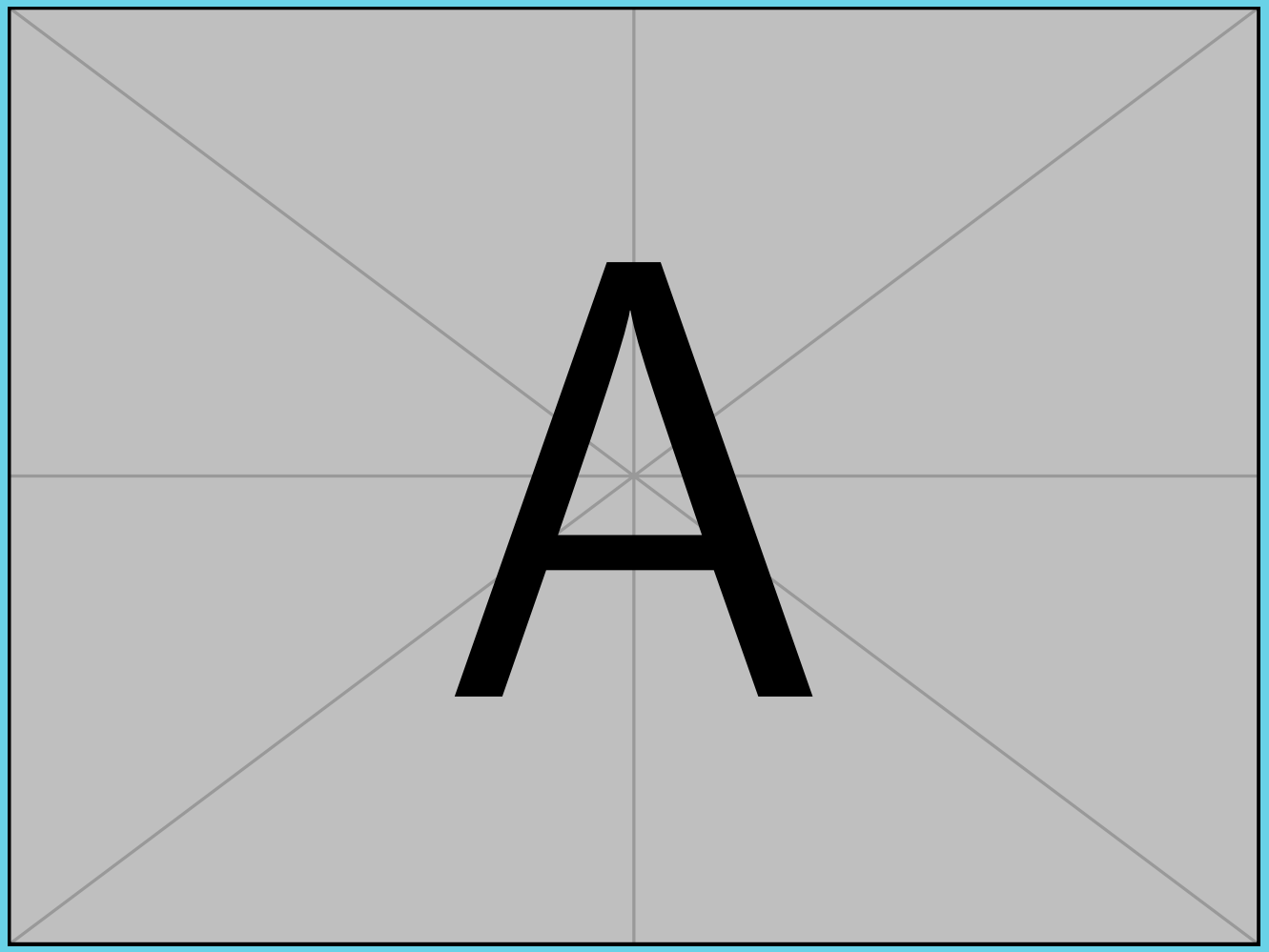
4 Quoting and special characters

5 Variables and special parameters

6 Shell expansion

7 Globs and filename expansion

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) go deeply into the details

The nature of Bash

As with 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) go deeply into the details

The nature of Bash

As with 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) go deeply into the details

The nature of Bash

As with 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 Q 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**

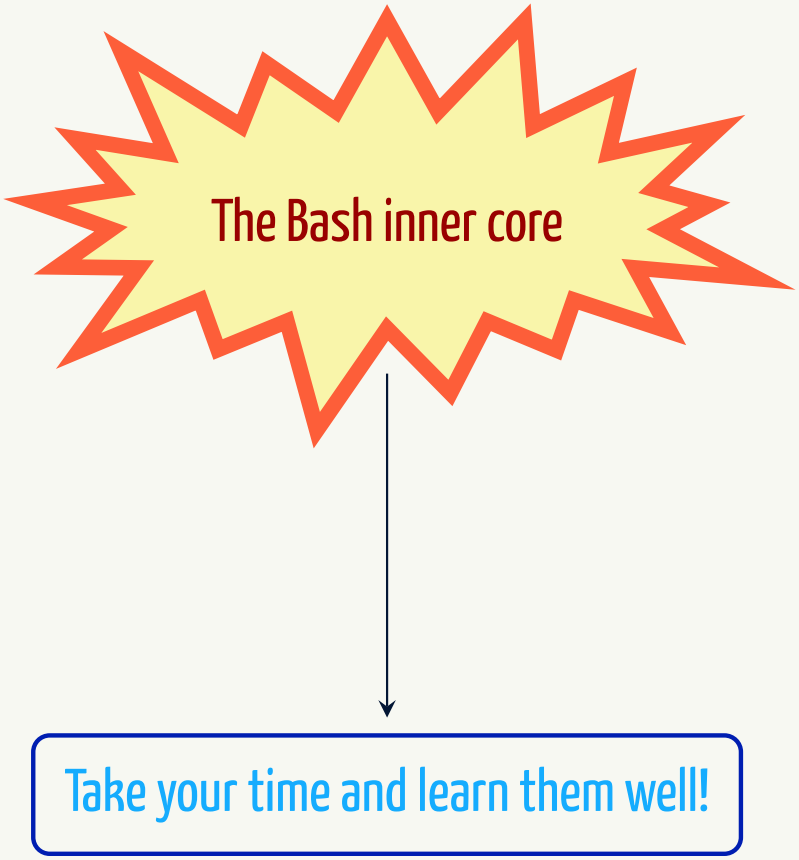
bold text	type exactly as shown.
italic text	replace with appropriate argument.
[-abc]	any or all arguments within [ ] are optional.
-a|-b	options delimited by | cannot be used together.
argument ...	argument is repeatable.
[*expression*] ...	entire expression within [ ] is repeatable.

The goal of the day

- Arguments
- Quotes
- Shell parameters
- Special variables (e.g. `IFS`)
- Shell expansion
  - Brace expansion
  - Parameters expansion
  - ...
  - Word splitting
  - Filename expansion
- Globbing

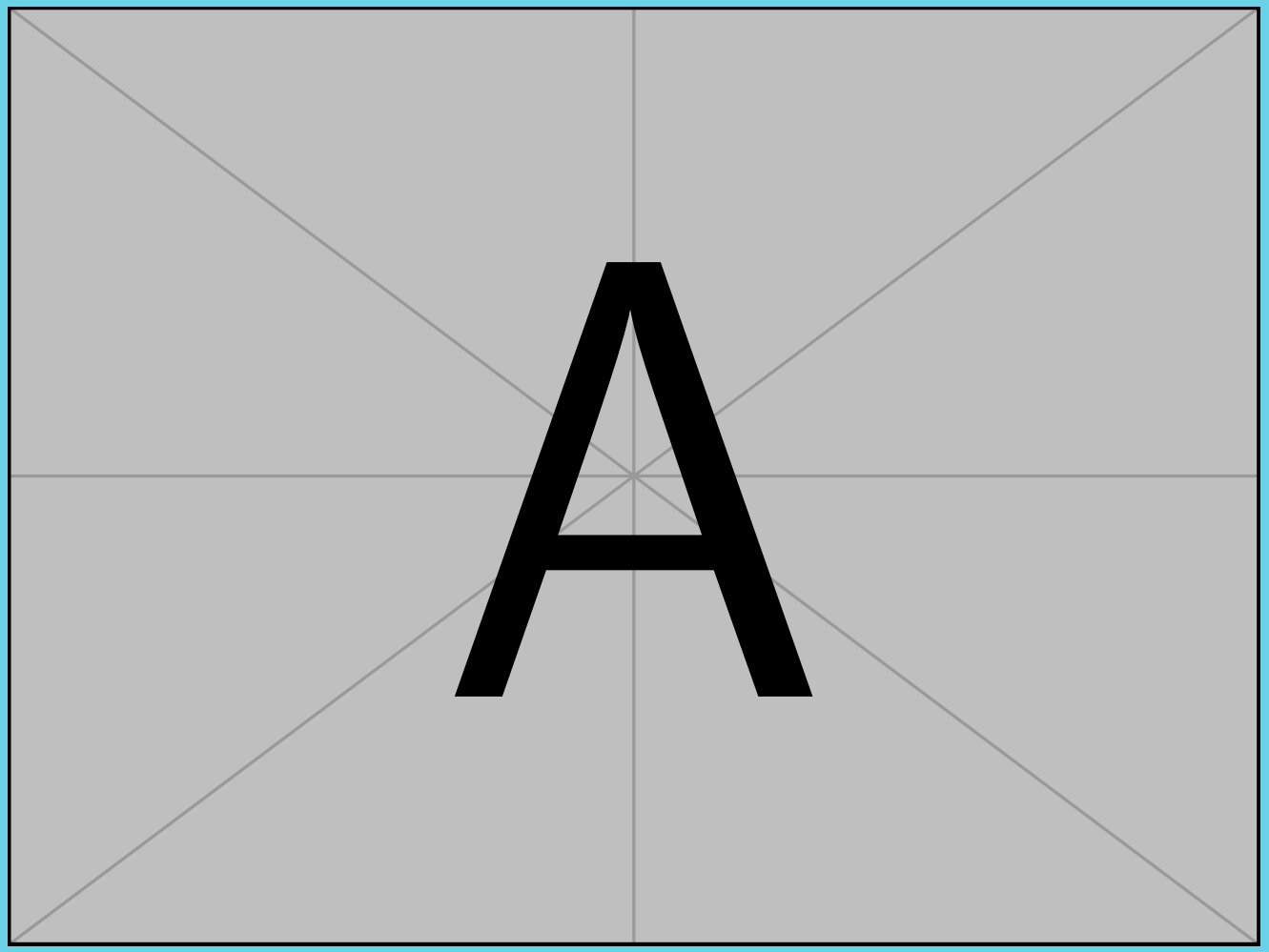
The goal of the day

- Arguments
- Quotes
- Shell parameters
- Special variables (e.g. `IFS`)
- Shell expansion
  - Brace expansion
  - Parameters expansion
  - ...
  - Word splitting
  - Filename expansion
- Globbing



Disclaimer: Some slides will unavoidably refer to later material. Hopefully, everything will be clarified at the end of the day and, for sure, by Friday!

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.).

```
$ command arg1 arg2   arg3       arg4
$ command arg1 arg2 arg3 arg4
```

The amount of whitespace between arguments does not matter!

```
$ echo I am Clark Kent
I am Clark Kent                          # <- Same output
$ echo I    am  Clark         Kent
I am Clark Kent                          # <- as here!
```


* There are a few advanced cases, such as commands that span multiple lines, that have slightly different rules.

The first hurdle: spaces

How can such an 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

```
$ ls -1
The secret voice.mp3
secret
$ rm The secret voice.mp3  # rm gets 3 arguments, not 1!
rm: cannot remove 'The': No such file or directory
rm: cannot remove 'voice.mp3': No such file or directory
$ ls
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 go more deeply into Word splitting later!

The first hurdle: spaces

How can such an 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

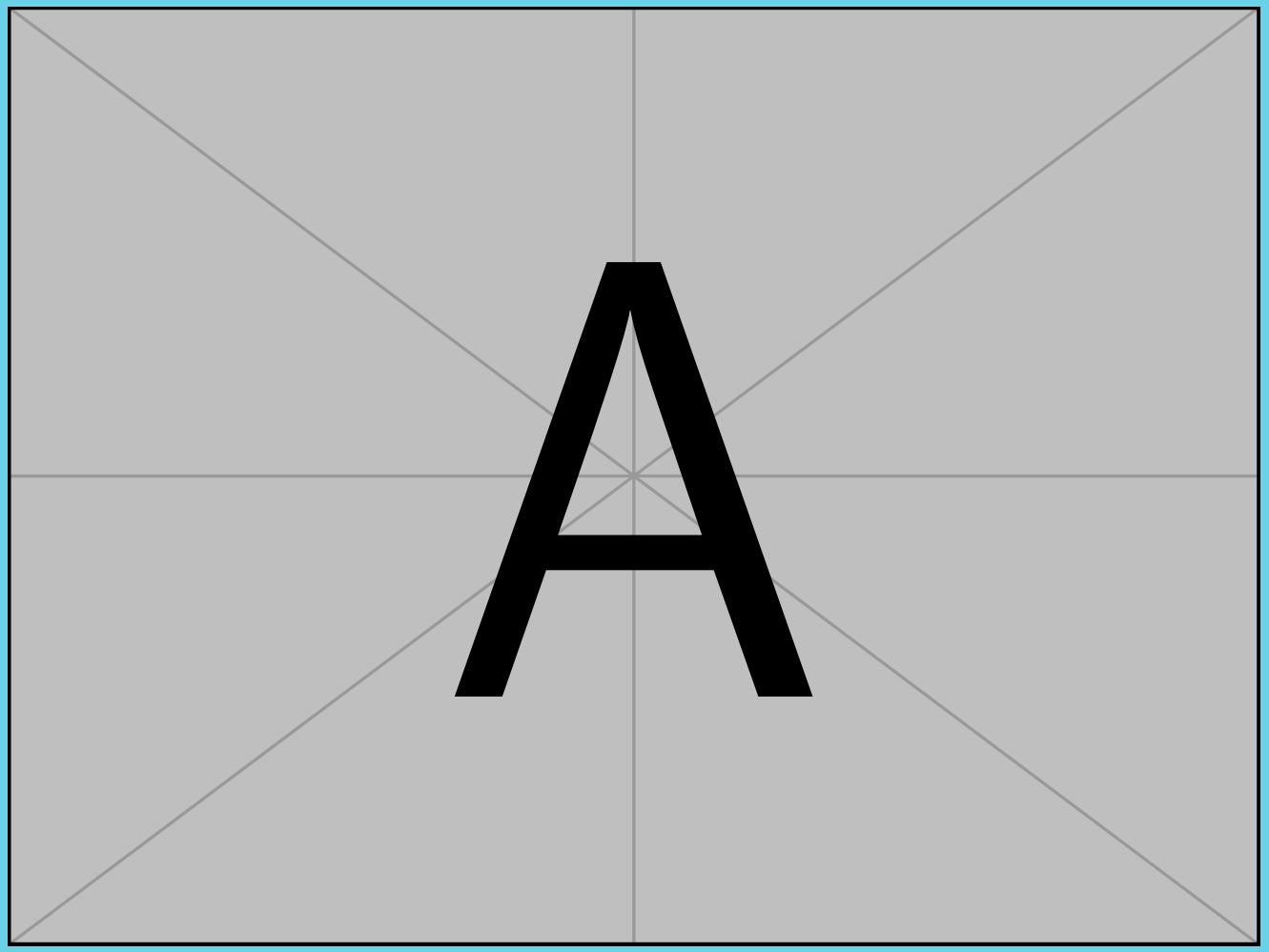
```
$ ls -1
The secret voice.mp3
secret
$ rm The secret voice.mp3  # rm gets 3 arguments, not 1!
rm: cannot remove 'The': No such file or directory
rm: cannot remove 'voice.mp3': No such file or directory
$ ls
The secret voice.mp3       # Where's your file 'secret'!?
```

`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...

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.

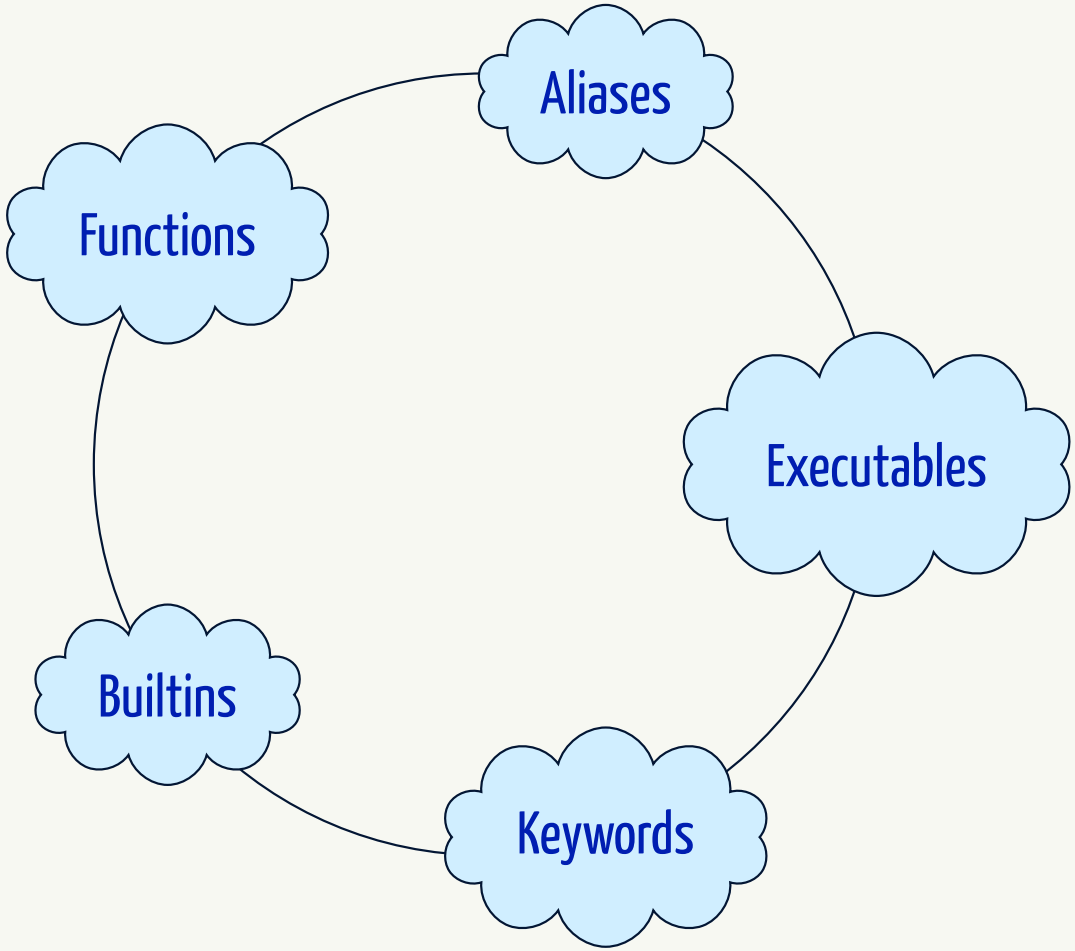
We have all the responsibility

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

I will loosely use the term **string** throughout the lecture, mostly referring to a portion of text contained in a variable.

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

```
$ help alias
alias: alias [-p] [name[=value] ... ]
    Define or display aliases.

# [More information]

$ alias ls='ls --color=auto'
$ ls  # The command executed is, instead:  ls --color=auto
```


Bash stores the values of aliases in an array called `BASH_ALIASES`.

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, a few hints:

- Functions in Bash are more powerful than aliases and the 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 ]    # Oops, < 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

- Executables 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 the 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

```
$ ls /home/sciarra/.local/bin
...    g++    ... # Version 8.3.0
$ ls /usr/bin
...    g++    ... # Version 5.4.0
$ echo ${PATH}
/home/sciarra/.local/bin:/usr/bin:/bin
$ g++ -dumpversion
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 **env**ironment (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 **env**ironment (i.e. in `PATH`).

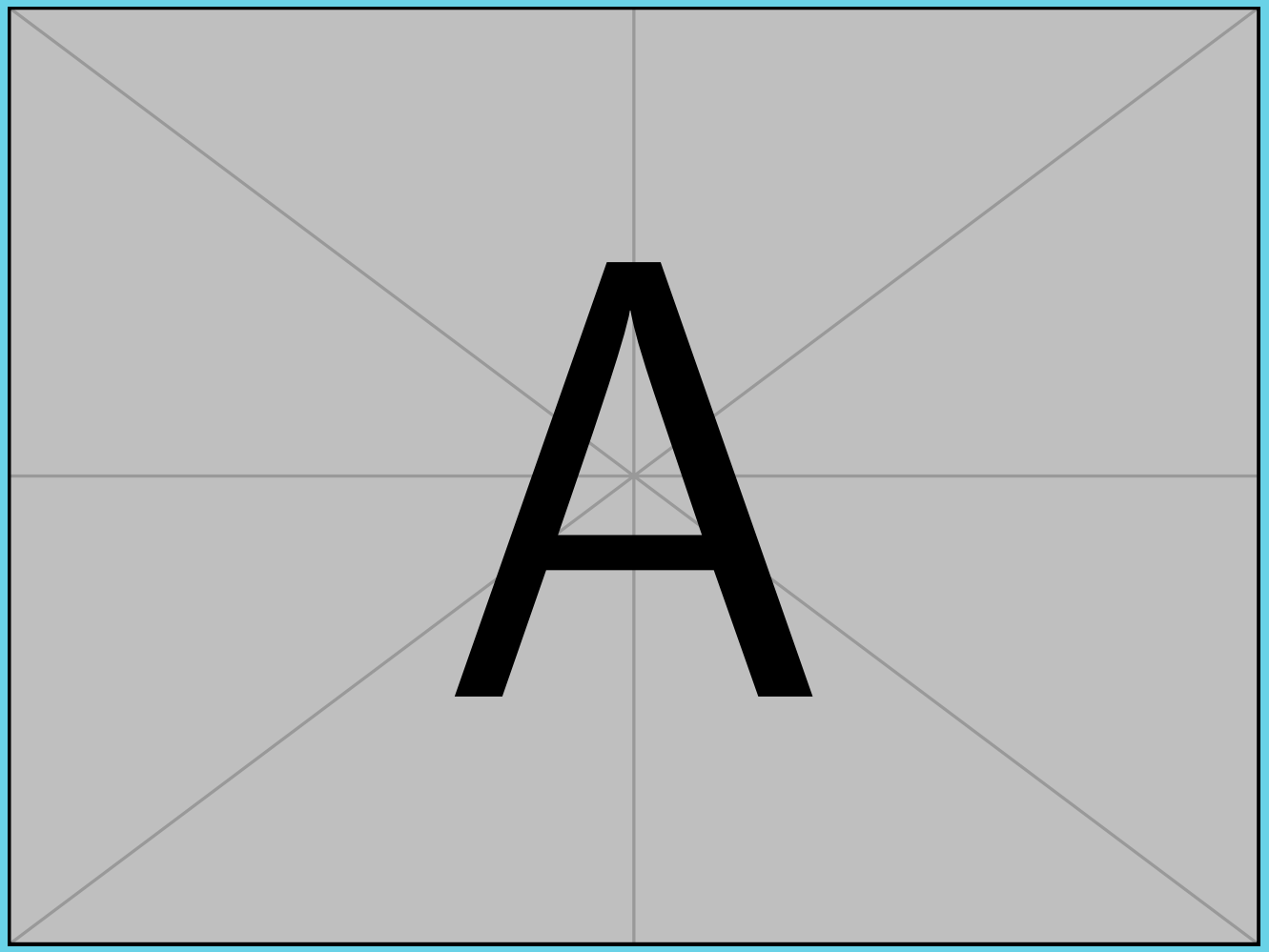
- Execute the script in either of the following ways:

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

or

```
$ chmod +x myscript  # Make the file executable
$ ./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

This is the short story. The verbose version is available at the end of 🔗this page .

Examples about quoting

```
$ echo I    am  Clark       Kent
I am Clark Kent
$ echo I"    "am"  "Clark"       "Kent         # Not nice
I    am  Clark       Kent
$ echo I\ \ \ \ am\ \ Clark\ \ \ \ \ \ \ Kent # Really!?
I    am  Clark       Kent
$ echo 'I    am  Clark       Kent'
I    am  Clark       Kent
$ echo "I    am  Clark       Kent"
I    am  Clark       Kent
```

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

Examples about quoting

```
$ ls *.tex
Day_1.tex   Day_2.tex   Day_3.tex
$ ls '*.tex'
ls: cannot access '*.tex': No such file or directory
$ ls "*.tex"
ls: cannot access '*.tex': No such file or directory
$ echo "Hello\nWorld"
Hello\nWorld
$ echo $'Hello\nWorld'
Hello
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 space, tab, newline, carriage return, vertical tab or form feed }

␣ 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

} Already discussed on slide 14

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 #. }

! The negate keyword reverses a test or an exit status

{ When history expansion is active, as by default in interactive shells, ! invokes history expansion, not covered here. }

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”

`> >> <` 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

`(( ))` 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

`(( ))` 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 ← Advanced reading

`[ ]` An alias for the old test command
Commonly used in POSIX shell scripts
Lacks many features of `[[ ]]` ← Advanced reading

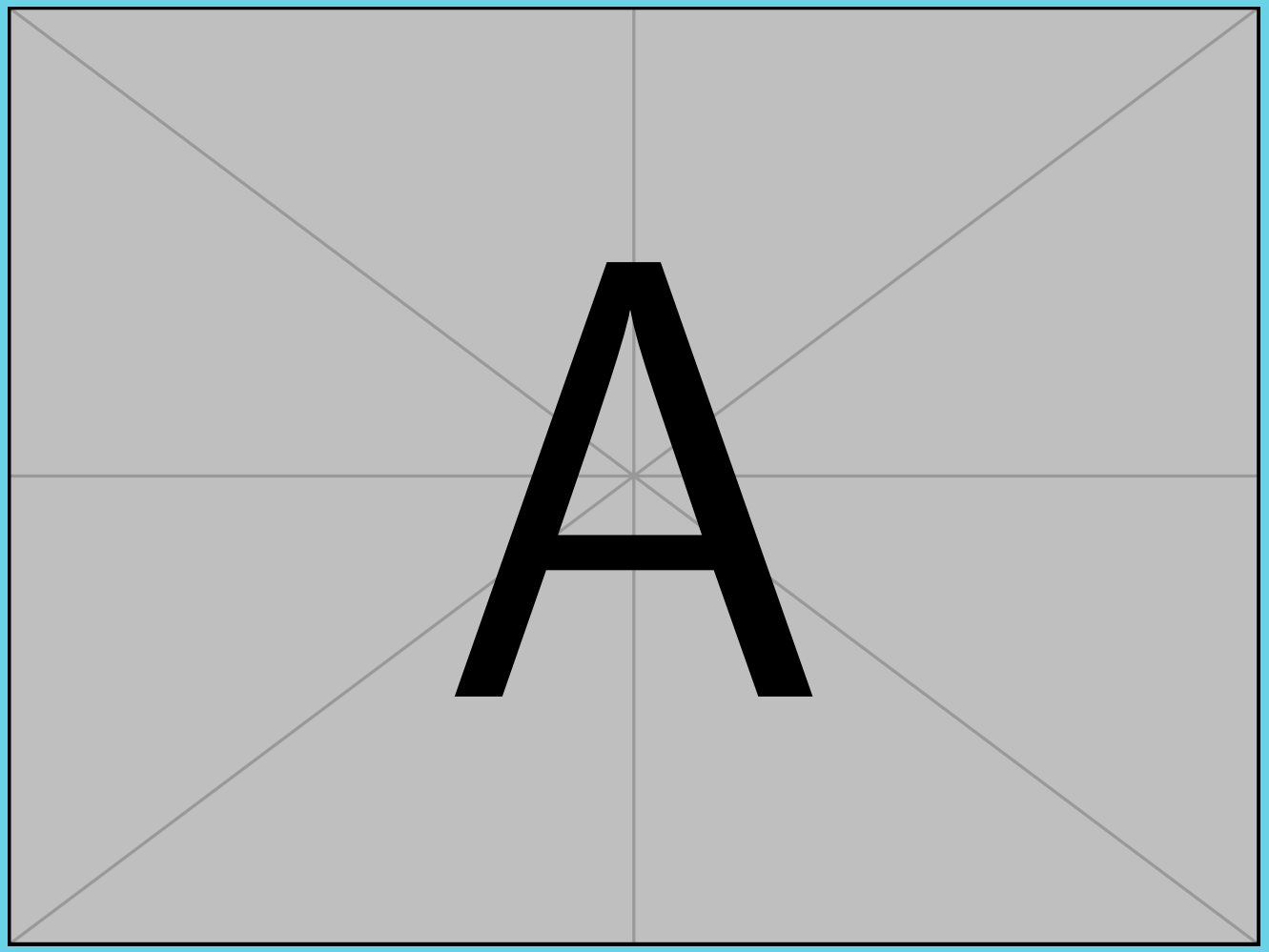
`$[ ]` Arithmetic expression - use `$(( ))` instead
Simply do not use it, it dates back to 1990s!!!

Legacy code and portability

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

At the end of the lecture, you might come back to this page and read about obsolete features!

Variables and special parameters

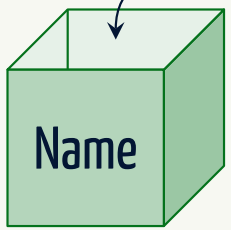


Seljalandsfoss: A 360° cascade

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)

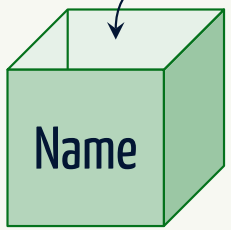
A word consisting only of

- 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

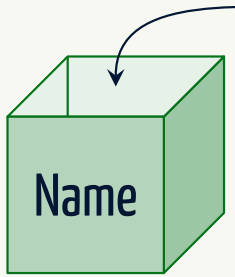
```
$ variableName=variableContent
```

- 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

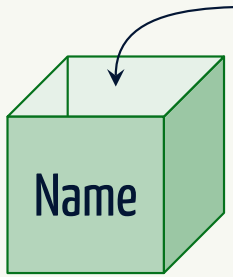
```
$ prefix='Day_'
$ day='Monday'
$ echo "$prefix1.pdf are the slides for ${day}"
.pdf are the slides for Monday
$ echo "${prefix}1.pdf are the slides for ${day}"
Day_1.pdf are the slides for Monday
```

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!

There is a way to activate a different behaviour of Bash when performing parameter expansion on unset variables: `set -u` → More on this next days.

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

```
$ aVar=5; aVar+=2; echo "${aVar}"; unset aVar
52
$ aVar=5; let aVar+=2; echo "${aVar}"; unset aVar
7
$ declare -i aVar=5; aVar+=2; echo "${aVar}"; unset aVar
7
$ aVar=5+2; echo "${aVar}"; unset aVar
5+2
$ declare -i aVar=5+2; echo "${aVar}"; unset aVar
7
$ declare -i aVar=5; aVar+=aVar; echo "${aVar}"; unset aVar
10
$ declare -i aVar=5; aVar+='foo'; echo "${aVar}"; unset aVar
5
# "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!

```
$ today=Monday
$ echo "Today is ${today}"
Today is Monday
# Bash takes the content of the variable today
# and replaces ${today} by Monday. It seems equivalent to:
$ echo Today is Monday
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!

```
$ today=Monday
$ echo "Today is ${today}"
Today is Monday
# Bash takes the content of the variable today
# and replaces ${today} by Monday. It seems equivalent to:
$ echo Today is Monday
Today is Monday
```

Everything seems to work as expected... but:

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

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

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

```
# Please, do not try to put quotes in variables!
$ song="\"My song.mp3\""
$ rm ${song}
rm: "My: No such file or directory
rm: song.mp3": No such file or directory
# Here the quotes contained in the variable song
# are literal characters and they are not interpreted
# 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!

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

```
# 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 v5.2 section 5

```
# 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_WORDBREAKS     HISTTIMEFORMAT    PPID
BASH_ARGC               COMP_WORDS          HOSTFILE          PROMPT_COMMAND
BASH_ARGV               COMPREPLY           HOSTNAME          PROMPT_DIRTRIM
BASH_ARGV0              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           SHELLOPTS
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
```


Use `compgen -v` to get a complete list of all variables available in your session.

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 v5.2 section 3.4.2

- `0` 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 each positional parameter to a separate word
  Double quoted, `"$*"`, expands to `"$1c$2c...c$N"`
  where `c` is the first character of the `IFS` variable
- `@` Expands each positional parameter to a separate word
  Double quoted, `"$@"`, expands to `"$1" "$2" ... "$N"`
- `#` 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

Unquoted `$*` and `$@` expansions in many context are then subject to further word splitting and filename expansion.

The Special Parameters

Bash manual v5.2 section 3.4.2

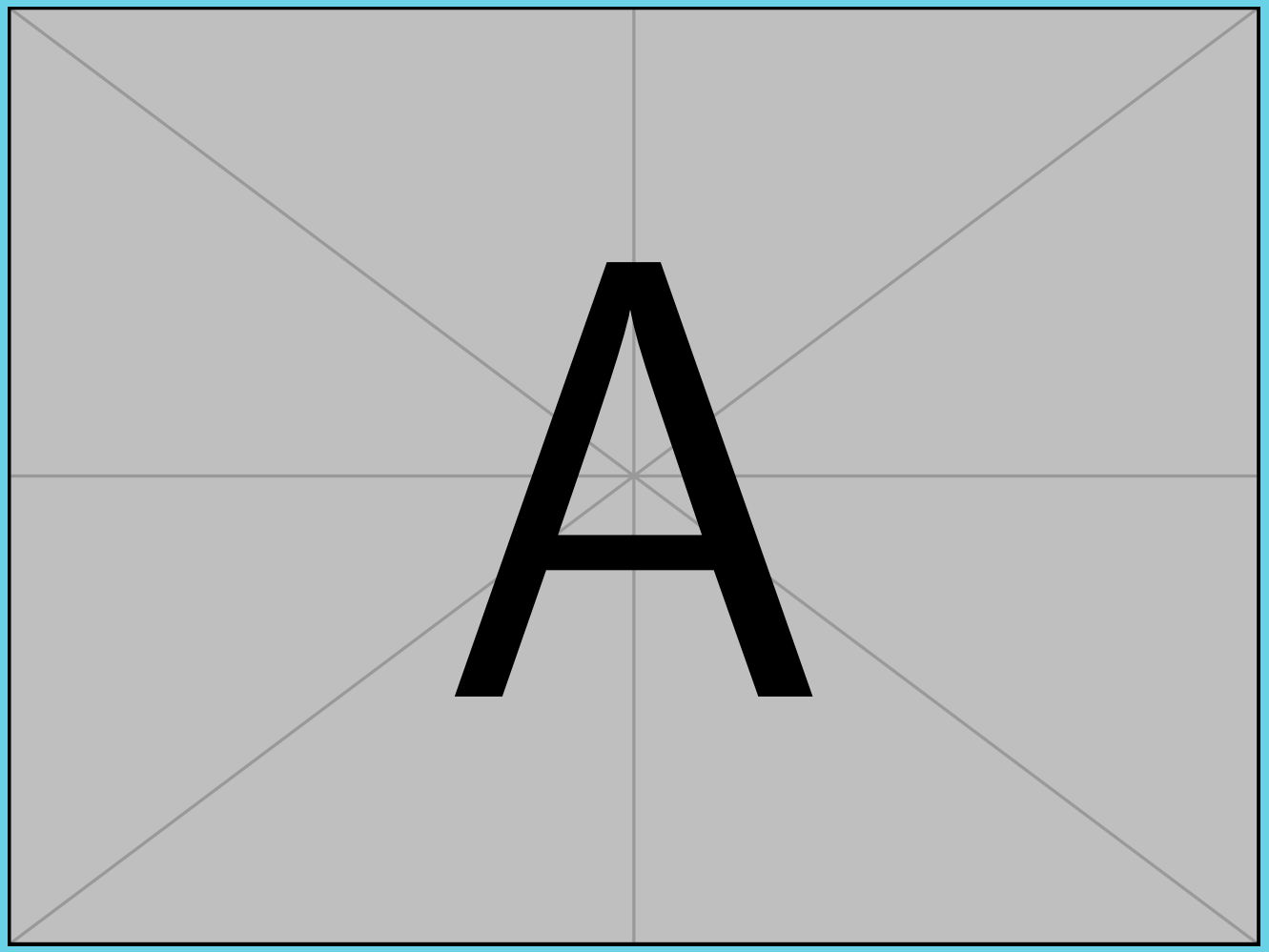
- `0` Contains the name, or the path, of the script. Do not rely on it or read the manual carefully!
- `1` `2` … Positional Parameters contain the arguments passed to the current script or function
- `*` Expands each positional parameter to a separate word
  Double quoted, `"$*"`, expands to `"$1c$2c...c$N"`
  where `c` is the first character of the `IFS` variable
- `@` Expands each positional parameter to a separate word
  Double quoted, `"$@"`, expands to `"$1" "$2" ... "$N"`
- `#` Expands to the number of positional parameters that are currently set

Exercise Sheet 1

- `-` 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

Unquoted $* and $@ expansions in many context are then subject to further word splitting and filename expansion.

Shell expansion



Danger: Sneaker waves!

Several kind of expansions

Bash manual v5.2 section 3.5

A line is split into tokens using unquoted metacharacters:

␣ \t \n | & ; ( ) < >

Expansion is performed on the command line after it has been split into tokens:

- Brace expansion ← 1
- Parameter and variable expansion
- Tilde expansion
- Arithmetic expansion
- Process substitution { Done only from v5.1 when Bash in POSIX Mode (item 30) }
- Command substitution

  2 At the same time!
- Word splitting ← 3
- Filename expansion ← 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. `[prefix]{comma separated list}[postfix]`
  2. `[prefix]{X..Y[..increment]}[postfix]`
- `[prefix]` and `[postfix]` are optional and 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 `[prefix]` and `[postfix]` 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. [prefix]{comma separated list}[postfix]
  2. [prefix]{X..Y[..increment]}[postfix]

```
$ echo {a,b,c}
a b c
$ echo {a,b,c}.tex
a.tex b.tex c.tex
$ echo image.{jpg,png,pdf}
image.jpg image.png image.pdf
$ echo {1..8}
1 2 3 4 5 6 7 8
$ echo {8..1}
8 7 6 5 4 3 2 1
$ echo {1..8..3}
1 4 7
$ echo {a..e}
a b c d e
$ echo {e..a..2}
e c a
$ echo file_{01..3}.pdf
file_01.pdf file_02.pdf file_03.pdf #Note leading zeros!
```

Brace expansion

- It comes in two forms:
  1. [prefix]{comma separated list}[postfix]
  2. [prefix]{X..Y[..increment]}[postfix]

```
$ echo {A..C}{0..2}
A0 A1 A2 B0 B1 B2 C0 C1 C2
$ echo {A..C} {0..2} # Here two independent expansions!
A B C 0 1 2
$ echo {in,out}{go,com}ing
ingoing incoming outgoing outcoming
$ echo {{A,E,I,O,U},{0..9}}
A E I O U 0 1 2 3 4 5 6 7 8 9
$ echo {a..z..x}
{a..z..x}  # No brace expansion!
$ aVar=1; echo {$aVar..5}; unset aVar
{1..5}     # No brace expansion!
$ echo {b,1..5}
b 1..5     # Not surprising, right?
$ echo {b,{1..5}}
b 1 2 3 4 5
$ 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
- There are 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

Everything can be found in the Bash manual v5.2 section 3.5.3

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

```
$ aVar="Hello"
# The variable aVar is set and not empty/null
$ echo "_${aVar}_"
_Hello_
$ echo "_${aVar-Goodbye}_   _${aVar:-Goodbye}_"  # Use default Value
_Hello_   _Hello_
$ echo "_${aVar=Goodbye}_   _${aVar:=Goodbye}_"  # Assign default Value
_Hello_   _Hello_
$ echo "_${aVar?Goodbye}_   _${aVar:?Goodbye}_"  # Exit with message
_Hello_   _Hello_
$ echo "_${aVar+Goodbye}_   _${aVar:+Goodbye}_"  # Use alternative value
_Goodbye_   _Goodbye_
$ aVar=""
# The variable aVar is now set BUT empty/null
$ echo "_${aVar}_"
__
$ echo "_${aVar-Goodbye}_   _${aVar:-Goodbye}_"
__   _Goodbye_
$ echo "_${aVar=Goodbye}_   _${aVar:=Goodbye}_"; aVar=""
__   _Goodbye_
$ echo "_${aVar?Goodbye}_   _${aVar:?Goodbye}_"
bash: aVar: Goodbye # Shell STOPS at second expansion and prints message!
$ echo "_${aVar?Goodbye}_"
__
$ echo "_${aVar+Goodbye}_   _${aVar:+Goodbye}_"
_Goodbye_   __
```


Parameter expansion: Checking variables state

```
$ unset aVar
# The variable aVar is now UNSET
$ echo "_${aVar}_"
__
$ echo "_${aVar-Goodbye}_   _${aVar:-Goodbye}_"
_Goodbye_   _Goodbye_
$ echo "_${aVar=Goodbye}_"; unset aVar
_Goodbye_
$ echo "_${aVar:=Goodbye}_"; unset aVar
_Goodbye_
$ echo "_${aVar?Goodbye}_"
bash: aVar: Goodbye # Shell STOPS at expansion and prints message!
$ echo "_${aVar:?Goodbye}_"
bash: aVar: Goodbye # Shell STOPS at expansion and prints message!
$ echo "_${aVar+Goodbye}_   _${aVar:+Goodbye}_"
__   __
```

Ok, but is all this useful? When?

Ohhh yes! These kinds of expansion are useful e.g. when it comes to checking 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 }

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

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


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

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

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

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

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


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

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**

{ We will discuss shell options in detail at a later point }

Introduced in Bash v5.2

If the `patsub_replacement` shell option is enabled using `shopt`, any unquoted instances of `&` in `string` are replaced with the matching portion of pattern. This is intended to duplicate a common sed idiom.

However, this is a breaking change: Be aware!

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

Parameter expansion: String manipulation (III)

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


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

Parameter expansion: String manipulation (III)

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

```
$ bVar=' & ';
# Assume to have Bash v5.2 or later
$ echo "${BASH_VERSION} -> $(shopt patsub_replacement)"
5.2.26(1)-release -> patsub_replacement on
$ echo "${aVar//aa/ & }   ${aVar/aa/${bVar}}"
123 aa 000 aa 321   123 aa 000aa321
$ echo "${aVar//aa/ \& }"
123 & 000 & 321
$ echo ${aVar//aa/" & "} ' ' ${aVar/aa/"${bVar}"}
123 & 000 & 321   123 & 000aa321
# Nested quotes are correctly handled by the shell
$ echo "${aVar//aa/" & "}   ${aVar/aa/"${bVar}"}"
123 & 000 & 321   123 & 000aa321
```


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

Parameter expansion: String manipulation (IV)

Since Bash v4.0 (2009)

First uppercase: `${parameter^pattern}`

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

All uppercase: `${parameter^^pattern}`

As the previous, but all characters are capitalised

First lowercase: `${parameter,pattern}`

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

All lowercase: `${parameter,,pattern}`

As the previous, but all characters are transformed to lowercase

Pattern matching

The pattern is expanded to produce a pattern just as in filename expansion. Each character in the expanded value of parameter is tested against pattern, and, if it matches the pattern, its case is converted. The pattern should not attempt to match more than one character. If pattern is omitted, it is treated like a `?`, which matches every character.

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

Parameter expansion: String manipulation (IV)

Since Bash v4.0 (2009)

First uppercase: `${parameter^pattern}`

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

All uppercase: `${parameter^^pattern}`

As the previous, but all characters are capitalised

First lowercase: `${parameter,pattern}`

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

All lowercase: `${parameter,,pattern}`

As the previous, but all characters are transformed to lowercase

```
$ aVar='hELLO, World!'
# Let us demonstrate a bit
$ echo "${aVar^}  -  ${aVar^^}"
HELLO, World!  -  HELLO, WORLD!
$ echo "${aVar,}  -  ${aVar,,}  -  ${aVar,,[LW]}"
hELLO, World!  -  hello, world!  -  hEllO, world!
```


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

Parameter expansion: String manipulation (V) and reflection (I)

Since Bash v4.4 (2016)

Syntax: `${parameter@operator}`
The expansion is either a transformation of the value of `parameter` or information about `parameter` itself, depending on the value of `operator`.

Each operator is a single letter:

- **a** The expansion is a string consisting of flag values representing `parameter`'s attributes.
- **A** The expansion is a string in the form of an assignment statement or `declare` command that, if evaluated, will recreate parameter with its attributes and value.
- **E** The expansion is a string that is the value of parameter with backslash escape sequences expanded as with the `$'...'` quoting mechanism.
- **P** The result of expanding the value of parameter as if it were a prompt string.
- **Q** The value of parameter quoted in a format that can be reused as input.

Parameter expansion: String manipulation (V) and reflection (I)

Since Bash v4.4 (2016)

Syntax: `${parameter@operator}`

The expansion is either a transformation of the value of `parameter` or information about `parameter` itself, depending on the value of `operator`.

```
$ declare -i aVar=5
# Retrieve information about the variable
$ echo "${aVar@a}"
i
$ echo "${aVar@A}"; unset aVar
declare -i aVar='5'
$ aVar='Hello\tworld!'; echo "${aVar@E}"
Hello   world!
$ aVar='\d \t'; echo "${aVar@P}"
Wed Sep 13 16:04:11
$ echo "${IFS@Q}"
$' \t\n'
```

Parameter expansion: String manipulation (VI) and reflection (II)

Since Bash v5.1 (2020)

Further operators introduced more recently:

- **U** The expansion is a string that is the value of `parameter` with lowercase alphabetic characters converted to uppercase.
- **u** The expansion is a string that is the value of `parameter` with the first character converted to uppercase, if it is alphabetic.
- **L** The expansion is a string that is the value of `parameter` with uppercase alphabetic characters converted to lowercase.
- **K** Produces a possibly-quoted version of the value of `parameter`, except that it prints the values of indexed and associative arrays as a sequence of quoted key-value pairs
- **k** Like the `K` transformation, but expands the keys and values of indexed and associative arrays to separate words after word splitting. { Since Bash v5.2 }

We will see examples of `@k` and `@K` expansions when discussing arrays

Parameter expansion: String manipulation (VI) and reflection (II)

Since Bash v5.1 (2020)

Further operators introduced more recently:

- **U** The expansion is a string that is the value of `parameter` with lowercase alphabetic characters converted to uppercase.
- **u** The expansion is a string that is the value of `parameter` with the first character converted to uppercase, if it is alphabetic.
- **L** The expansion is a string that is the value of `parameter` with uppercase alphabetic characters converted to lowercase.

```
$ aVar="hello WoRLd!"
# Let us demonstrate
$ echo "${aVar@U}"    # Equivalent to ${aVar^^}
HELLO WORLD!
$ echo "${aVar@u}"    # Equivalent to ${aVar^}
Hello WoRLd!
$ echo "${aVar@L}"    # Equivalent to ${aVar,,}
hello world!
```

Parameter expansion: 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!

Tilde expansion in its basic version

1. If a word begins with an unquoted tilde character ~
   ⇒ all of the characters up to the first unquoted slash are considered a **tilde-prefix**.
   { all characters are considered, if there is no unquoted slash }
2. If none of the characters in the tilde-prefix are quoted
   ⇒ the characters in the tilde-prefix following the tilde are treated as a possible **login name**.
3. If this login name is the null string
   ⇒ the tilde is replaced with the value of the `HOME` shell variable,
   or with the home directory of the user executing the shell, if `HOME` is unset.

   Otherwise
   ⇒ the **tilde-prefix** is replaced with the home directory associated with the specified **login name**.
4. If the login name is invalid the word is left unchanged.

Each variable assignment is also checked for unquoted tilde-prefixes immediately following a `':'` or the first `'='`. In these cases, tilde expansion is also performed.

Tilde expansion in its basic version

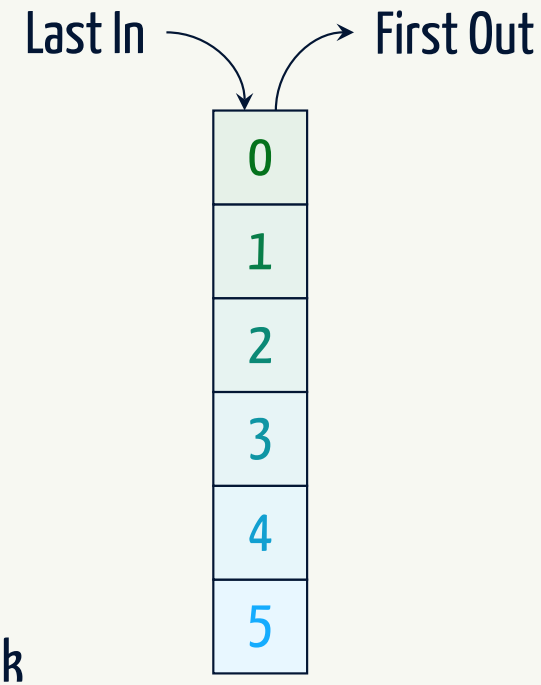
1. If a word begins with an unquoted tilde character ~
   ⇒ all of the characters up to the first unquoted slash are considered a **tilde-prefix**.
   { all characters are considered, if there is no unquoted slash }
2. If none of the characters in the tilde-prefix are quoted
   ⇒ the characters in the tilde-prefix following the tilde are treated as a possible **login name**.
3. If this login name is the null string
   ⇒ the tilde is replaced with the value of the HOME shell variable,
   or with the home directory of the user executing the shell, if HOME is unset.
   Otherwise
   ⇒ the **tilde-prefix** is replaced with the home directory associated with the specified **login name**.
4. If the login name is invalid the word is left unchanged.

```
$ echo ~  ~palao
/home/sciarra /home/palao
$ echo ~"sciarra" ~sciar"ra" ~sciarra""
~sciarra ~sciarra ~sciarra
$ echo ~NotExistentUser # Invalid login name
~NotExistentUser
```


Each variable assignment is also checked for unquoted tilde-prefixes immediately following a ':' or the first '='. In these cases, tilde expansion is also performed.

The Directory Stack

Bash manual v5.2 section 6.8



Intuitive definition of stack

An abstract data type or data structure based on the principle of last in first out.

- The directory stack is a list of recently-visited directories
- The `dirs` builtin **displays** the contents of the directory stack
- The current directory is always the "top" of the directory stack
- The `pushd` builtin **adds** directories to the stack as it changes the current directory
- The `popd` builtin **removes** specified directories from the stack and changes the current directory to the new top directory
- Both `pushd` and `popd` accept a `-n` option to avoid changing directory

The contents of the directory stack are also visible as the value of the `DIRSTACK` shell array variable.

The Directory Stack: An example usage

```
$ cd ~; pwd; dirs -v
/home/sciarra
 0  ~
$ cd ~/Documents/Seminars/BashCourse_2020; dirs -v
 0  ~/Documents/Seminars/BashCourse_2020
$ pushd /myBackupDisk/University/Seminars/BashCourse_2020
# Stack automatically displayed on one line
$ dirs -v
 0  /myBackupDisk/University/Seminars/BashCourse_2020
 1  ~/Documents/Seminars/BashCourse_2020
$ pushd ~; pwd
# Stack automatically displayed on one line
/home/sciarra
$ dirs -v
 0  ~
 1  /myBackupDisk/University/Seminars/BashCourse_2020
 2  ~/Documents/Seminars/BashCourse_2020
$ cd Documents; dirs -v
 0  ~/Documents
 1  /myBackupDisk/University/Seminars/BashCourse_2020
 2  ~/Documents/Seminars/BashCourse_2020
$ popd; pwd  # Stack automatically displayed on one line
/myBackupDisk/University/Seminars/BashCourse_2020
```

Tilde expansion and its advanced usage

If the tilde-prefix matches the following forms, a special expansion is done:

- `~+` `${PWD}` is substituted.
- `~-` The value of the shell variable `OLDPWD`, if it is set, is substituted.
  → `${OLDPWD-'~-'}`
- `~N` The string that would be displayed by `dirs +N` is substituted.
- `~+N` The string that would be displayed by `dirs +N` is substituted.
- `~-N` The string that would be displayed by `dirs -N` is substituted.

If the tilde expansion fails, the word is left unchanged!

Check out `help dirs` to read more about the options meaning

Bash also performs tilde expansion on words satisfying the conditions of variable assignments (e.g. `alias`, `declare`, `export`, `readonly`, `local`).

Tilde expansion and its advanced usage

If the tilde-prefix matches the following forms, a special expansion is done:

- `~+` `${PWD}` is substituted.
- `~-` The value of the shell variable `OLDPWD`, if it is set, is substituted.
  → `${OLDPWD-'~-'}`
- `~N` The string that would be displayed by `dirs +N` is substituted.
- `~+N` The string that would be displayed by `dirs +N` is substituted.
- `~-N` The string that would be displayed by `dirs -N` is substituted.

If the tilde expansion fails, the word is left unchanged!

Check out `help dirs` to read more about the options meaning

You can then take advantage of the Directory Stack!

Bash also performs tilde expansion on words satisfying the conditions of variable assignments (e.g. `alias`, `declare`, `export`, `readonly`, `local`).

Tilde expansion and its advanced usage

```
$ dirs -v
 0  ~
 1  /scratch/latticeqcd/sciarra
 2  /home/latticeqcd/public
 3  /arc01/archive/latticeqcd/sciarra
 4  /arc02/archive/latticeqcd/sciarra
$ cd ~2; pwd
/home/latticeqcd/public
$ cd ~-0; pwd; cd ~4; pwd
/arc02/archive/latticeqcd/sciarra
/arc02/archive/latticeqcd/sciarra
# Of course you can use them for any operation, e.g.
$ cp -r ~3/Results ~4/Results_backup
$ ls ~5
ls: cannot access ~5: No such file or directory
```

Check out `help dirs` to read more about the options meaning

You can then take advantage of the Directory Stack!

Bash also performs tilde expansion on words satisfying the conditions of variable assignments (e.g. `alias`, `declare`, `export`, `readonly`, `local`).

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 v5.2 section 6.5

Arithmetic expansion

```
$ echo $(( 22 / 3 ))
7
$ aVar=22; echo $(( ${aVar} / 3 )); unset aVar
7
$ aVar=22; echo $(( aVar / 3 )); unset aVar
7
$ echo $(( aVar / 3 ))     # aVar is unset,  no  ${} => 0 is used:  $((0/3))
0
$ echo $(( ${aVar} / 3 ))  # aVar is unset, with ${} => '' is used: $(( /3))
bash: / 3 : syntax error: operand expected (error token is "/ 3 ")
$ echo $(( ))
0
$ aVar='Hello'; echo $(( aVar )) $(( ${aVar} )); unset aVar
0 0 # No variable 'Hello' existing
$ aVar='bVar'; bVar=5; echo $(( aVar )) $(( ${aVar} )); unset aVar bVar
5 5 # aVar is expanded to bVar, which is defined to 5
$ aVar='HelloMyLove'; echo $(( aVar )) $(( ${aVar} )); unset aVar
0 0 # No variable 'HelloMyLove' existing
$ aVar='Hello <3'; echo $(( aVar )) $(( ${aVar} )); unset aVar
1 1 # It makes sense, doesn't it?
$ echo $(( 2**51 * 4096 ))
-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!

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


Word splitting

...the evil of Bash!

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


  xxx          xx    >
$ IFS=$'\n'; args ${aVar}; unset IFS
2 args: <   > <  xxx         xx    >
$ IFS=$'\n\t'; args ${aVar}; unset IFS
4 args: <   > <  xxx > <   > <  xx  >
$ IFS=$'\n\t x'; args ${aVar}; unset IFS
5 args: <> <> <> <> <>
$ 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.

It is almost never desirable to put syntactical whitespace in parameters.

Other expansions

Before Word Splitting:

Command substitution: It occurs when a command is enclosed as `$(...)`
Do not use the deprecated syntax `` `...` ``
Still to early to be discussed → Bash manual v5.2 section 3.5.4

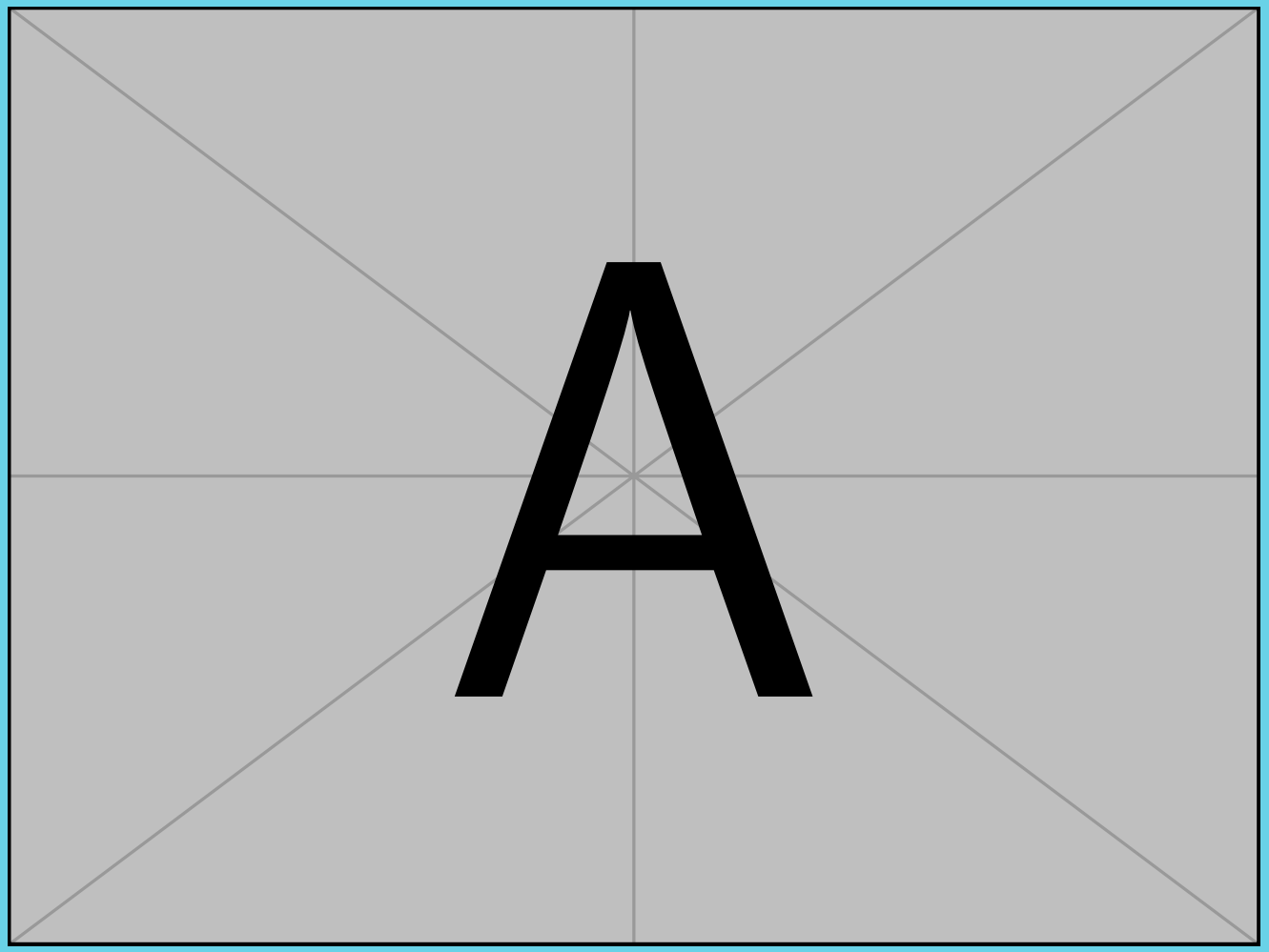
{ `$(...)` is substituted by the standard output of the command (executed in a subshell), with any trailing newlines deleted. }

Process substitution: Useful, but not discussed here → Bash manual v5.2 section 3.5.6
It takes the form `<(...)` or `>(...)`
It will discussed in the section about redirection

After Word Splitting:

Filename expansion: Discussed in the next 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 expressions { Since Bash v3.0 } ← { Cannot be used in Filename expansion! }

} Pattern matching done in Filename expansion

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 expressions { Since Bash v3.0 }

} Pattern matching done in Filename expansion

← { Cannot be used in Filename expansion! }

We will discuss this tomorrow 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

Filename expansion can be deactivated, see `set -f`.

The [...] pattern

- There are some special rules which 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 either 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:]]`.

The sorting order of characters in range expressions is determined by the current locale and the values of the `LC_COLLATE` and `LC_ALL` shell variables, if set.

Globs examples

```
$ ls
Bash.pdf   Day_1.tex   Day_2.tex   Day_3.tex   Day_extra.tex
$ echo *
Bash.pdf Day_1.tex Day_2.tex Day_3.tex Day_extra.tex
$ echo Day*
Day_1.tex Day_2.tex Day_3.tex Day_extra.tex
$ echo *.pdf
Bash.pdf
$ echo *_?.tex
Day_1.tex Day_2.tex Day_3.tex
$ echo *_[12]*
Day_1.tex Day_2.tex
$ echo *_[0-9]*
Day_1.tex Day_2.tex Day_3.tex
$ echo *_[[:digit:]]*
Day_1.tex Day_2.tex Day_3.tex
$ echo '*'  # As already said, quotes prevent globbing
*
$ echo "*_[[:digit:]]*"
*_[[:digit:]]*
```

Globs examples

```
21 $ aVar='Day3_Day2_Day1'; echo "${aVar#*[12]}"; 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!

Greg's Wiki

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 -1
   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!

Greg's Wiki

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 **sh**ell **op**tions:

```
$ 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

This option affects the parser behaviour. This aspect will be covered later in more detail.

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 **sh**ell **op**tions:

```
$ 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
```


This option affects the parser behaviour. This aspect will be covered later in more detail.