

Introduction to Bash scripting language

Day 2

Alessandro Sciarra

Z02 – Software Development Center

Organised by the CSC Frankfurt

27.10.2020

Topics of the day

- | | | | |
|---|-----------------------------------|----|---------------------------------------|
| 1 | Conditional blocks | 6 | Arrays and associative arrays |
| 2 | Loops | 7 | Input and output |
| 3 | Choices | 8 | Shell options |
| 4 | Colours and cursor movements | 9 | The GNU core utilities and util-linux |
| 5 | References and indirect expansion | 10 | Ready to script! |

You already know the basics

Yesterday you learnt already almost everything about the core of Bash. Today we will complete ~90% of the picture mostly exploring keywords.

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; then
    # 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 */
2 TeX
3 $ if [ -d 'TeX' ]; then echo 'YES'; else echo 'NO'; fi
4 YES
5 $ if [ -e 'TeX' ]; then echo 'YES'; else echo 'NO'; fi
6 YES
7 $ if [ -s 'TeX' ]; then echo 'YES'; else echo 'NO'; fi
8 YES
9 $ if [ -f 'TeX' ]; then echo 'YES'; else echo 'NO'; fi
10 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"
2 $ bVar="Clark Kent"
3 $ [ ${aVar} = ${aVar} ]
4 bash: [: too many arguments
5 $ if [ "${aVar}" = "${aVar}" ]; then echo 'YES'; else echo 'NO'; fi
6 YES
7 $ if [ "${aVar}" \> "${bVar}" ]; then echo 'YES'; else echo 'NO'; fi
8 YES
9 $ if [ 319 \< 7 ]; then echo 'YES'; else echo 'NO'; fi
10 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
2 NO
3 $ if [ 7 -ne 7 ]; then echo 'YES'; else echo 'NO'; fi
4 NO
5 $ if [ 7 -eq 7 ]; then echo 'YES'; else echo 'NO'; fi
6 YES
7 $ if [ 7 -gt 7 ]; then echo 'YES'; else echo 'NO'; fi
8 NO
9 $ if [ 7 -ge 7 ]; then echo 'YES'; else echo 'NO'; fi
10 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 either on l.h.s. or on r.h.s. 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'
# Pattern matching VS string comparison
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
[[ $PATH ]] && echo 'PATH variable set and non empty!'
# Mostly they are used in script
rm file || echo 'Could not delete file!'
mkdir TeX && cd TeX
source AuxiliaryOps.bash || exit 1
# Avoid complicated one-liners which might easily be wrong!
grep -q goodword "$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
[[ $PATH ]] && echo 'PATH variable set and non empty!'
# Mostly they are used in script
rm file || echo 'Could not delete file!'
mkdir TeX && cd TeX
source AuxiliaryOps.bash || exit 1
# Avoid complicated one-liners which might easily be wrong!
grep -q goodword "$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
 -  [POSIX standard](#)
 -  [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!**

Loops



Ice cave in the Breiðamerkurjökull

Conditional loops

Often we need to repeat things: Copy and paste is not the solution!

`while` Repeat as long as a command is executed successfully (exit code is 0)

`until` Repeat as long as a command is executed unsuccessfully (exit code is not 0)

`for` It comes in two versions

- to iterate over a list
- to iterate over an integer index as in C

Which one should be used?

There will nearly always be multiple approaches to solving a problem. The test of your skill soon won't be about solving a problem as much as about how best to solve it.

While instead of until

The `until` loop is barely ever used, if only because it is pretty much exactly the same as `while` !



The while and until keywords

```
while COMMAND; do
    # Body of the loop: entered if COMMAND's exit code = 0
done

until COMMAND; do
    # Body of the loop: entered if COMMAND's exit code ≠ 0
done
```

- Testing command like [or preferably [[are often used
- Infinite loops can be achieved using the builtins true, false and :
- Use the continue builtin to skip ahead to the next iteration of a loop without executing the rest of the body
- use the break builtin to jump out of the loop and continue with the script after it
- Both continue and break accept an optional integer to act on nested loops



The while and until keywords

```
1 $ while true; do    # while :; do
2 >   echo "Infinite loop"
3 >   sleep 3
4 > done
5 Infinite loop
6 Infinite loop
7 ^C      # Press CTRL-C
```

```
1 # An example of countdown...
2 $ deadline=$(date -d "8 seconds" +'%s'); \
3 > now=$(date +'%s'); \
4 > while (( deadline - now > 0 )); do
5 >   echo "$((deadline - now)) seconds to BOOM!"
6 >   sleep 3
7 >   now=$(date +'%s')
8 > done; echo 'BOOOOOM!!'
9 8 seconds to BOOM!
10 5 seconds to BOOM!
11 2 seconds to BOOM!
12 BOOOOOM!!
```



The `for` keywords

```
for VARIABLE in WORDS; do
    # Body of the loop: VARIABLE set to WORD
done

for (( EXPR1; EXPR2; EXPR3 )) # Expressions can be empty
    # Body of the loop
done
```

- In the second form,
 - 1 it starts by evaluating the first arithmetic expression;
 - 2 it repeats as long as the second arithmetic expression is successful;
 - 3 at the end of each loop evaluates the third arithmetic expression.
- Bash takes the characters between `in` and the end of the line and
 - it splits them up into words;
 - this splitting is done on spaces and tabs, just like argument splitting;
 - if there are any unquoted substitutions, **they will be word-split as well** (using `IFS`).



The for keywords

```
1 $ for ((( ; 1; )); do echo "Infinite loop"; sleep 1; done  
2 Infinite loop  
3 Infinite loop  
4 ^C      # Press CTRL-C
```

```
1 $ for index in {0,1}{0,1}; do  
2 >   echo "${index} in base 2 is $(( 2#${index})) in base 10"  
3 > done; unset index  
4 00 in base 2 is 0 in base 10  
5 01 in base 2 is 1 in base 10  
6 10 in base 2 is 2 in base 10  
7 11 in base 2 is 3 in base 10
```

```
1 # BAD code!  
  
2 $ for file in $(ls *.mp3); do      # AAAARGH!  
3 >   rm "$file"  
4 > done; unset file
```



The `for` keywords

```
1 $ ls
2 Happy birthday.mp3
3 $ for file in $(ls *.mp3); do      # AAAARGH!
4 >   rm "$file"
5 > done; unset file
6 rm: cannot remove `Happy': No such file or directory
7 rm: cannot remove `birthday.mp3': No such file or directory
```

You want to quote it, you say?

```
1 $ ls
2 Happy birthday.mp3    Hello.mp3
3 $ for file in "$(ls *.mp3)"; do      # AAAARGH!
4 >   rm "$file"
5 > done; unset file
6 rm: cannot remove `Happy birthday.mp3'    Hello.mp3': No such
     file or directory
```

So, what do we do?



The `for` keywords

Use globs!

Bash **does** know that it is dealing with filenames, and it **does** know what the filenames are, and as such it can split them up nicely!

```
1 $ ls
2 Happy birthday.mp3    Hello.mp3
3 $ for file in *.mp3; do    # GOOD code
4 >   rm "$file"
5 > done; unset file
```

Do not be tempted

You might argue that, if there are no spaces in filenames, then you would have no troubles. But would you throw in the air a sharp knife trying to catch it afterwards, just because if you do not touch the blade it would be safe?



Choices



The Skaftafell natural park: Svartifoss

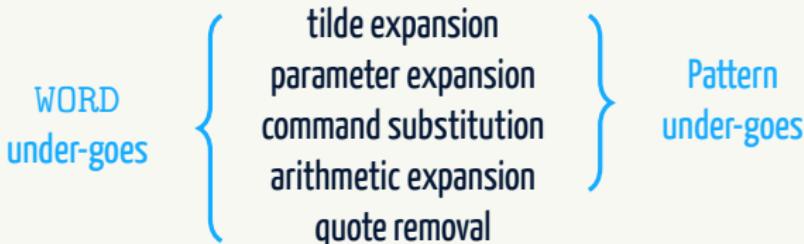
The case keyword

Completely equivalent to a `if-elif-else-fi` construct, but often handy:

```
case WORD in  
  [ [() pattern [| pattern]...] command-list ;; ]...  
esac
```

- The command-list corresponding to the first pattern that matches word is executed
- The match is performed according [Pattern Matching](#) rules
- The | is used to separate multiple patterns
- The) operator terminates a pattern list
- A list of patterns and an associated command-list is known as a **clause**

Before matching is attempted:



The case keyword

Completely equivalent to a **if-elif-else-fi** construct, but often handy:

```
case WORD in
  [ [() pattern [| pattern]...) command-list ;;]...
esac
```

```
1 case ${LANG} in
2   en* )
3     echo 'Hello!' ;;
4   fr* )
5     echo 'Salut!' ;;
6   de* )
7     echo 'Hallo!' ;;
8   it* )
9     echo 'Ciao!' ;;
10  es* )
11    echo 'Hola!' ;;
12  C | POSIX )
13    echo 'Hello World!' ;;
14  *)
15    echo 'I do not speak your language.' ;;
16 esac
```

Using ;& or ; & instead of ; ; allows to execute the command list or to test the patterns of the next clause, respectively.

The `select` keyword

A convenient statement for generating a menu of choices that the user can choose from:

```
select NAME in WORDS; do
    # Body with commands -> break needed to terminate
done
```

- The list of words following `in` is expanded, generating a list of items
- The set of expanded words is printed on the **standard error**, each preceded by a number
- If the `in WORDS` is omitted, the positional parameters are printed, as if `in "$@"`
- The `PS3` prompt is then displayed and a line is read from the standard input:
 - If the line consists of a number corresponding to one of the displayed words, then the value of `NAME` is set to that word
 - If the line is empty, the words and prompt are displayed again
 - Any other value read causes `NAME` to be set to null
 - The line read is saved in the variable `REPLY`
- It is good practice not to `break` if the user typed something meaningless!



The select keyword

A convenient statement for generating a menu of choices that the user can choose from:

```
select NAME in WORDS; do
    # Body with commands -> break needed to terminate
done
```

```
1 $ select file in *; do
2 >     echo "You picked \"${file}\" (${REPLY})"
3 >     break
4 > done
5 1) Day_1.tex
6 2) Day_2.tex
7 3) Day_3.tex
8 #? 2
9 You picked "Day_2.tex" (2)
10 $ echo "After select: \"${file}\" (${REPLY})"
11 After select: "Day_2.tex" (2)
12 $ unset file
```



The select keyword

A convenient statement for generating a menu of choices that the user can choose from:

```
select NAME in WORDS; do
    # Body with commands -> break needed to terminate
done
```

```
1 $ select file in *; do
2 >     echo "You picked \"${file}\" (${REPLY})"
3 >     break;
4 > done
5 1) Day_1.tex
6 2) Day_2.tex
7 3) Day_3.tex
8 #? dlfagfdsu
9 You picked "" (dlfagfdsu)
10 $ echo "After select: \"${file}\" (${REPLY})"
11 After select: "" (dlfagfdsu)
12 $ unset file
```

It is always important to validate the answer!



The select keyword

A convenient statement for generating a menu of choices that the user can choose from:

```
select NAME in WORDS; do
    # Body with commands -> break needed to terminate
done
```

```
1 $ select file in *; do
2 >     if [[ ${file} != '' ]]; then # GOOD PRACTICE
3 >         echo "You picked \"${file}\" (${REPLY})"
4 >         break;
5 >     fi
6 > done; unset file
7 1) Day_1.tex
8 2) Day_2.tex
9 3) Day_3.tex
10 #? dlfagfdsu
11 #? dgasdf
12 #? 4
13 #? 231345134
14 #? 2
15 You picked "Day_2.tex" (2)
```



Colours and cursor movements



Icebergs on the beach of the Jökulsárlón

Colours and formatting in the terminal

🔗 A good reference

- Most terminals are able to display colours and formatted texts thanks to [escape sequences](#)
- Your script can benefit from using colours (e.g. `errors`, `warnings`, `info`)
- You might run into compatibility problems (not really) [🔗 Compatibility list](#)
- The `echo` command has a `-e` option to enable the parsing of the escape sequences
- The `printf` command parses escape sequences

[Escape sequences](#) —

`<Esc> [FormatCode]`

- The `<Esc>` character can be obtained with any of the following syntaxes:
 - `\e`
 - `\033`
 - `\x1B`
- The `FormatCode` is an integer and different ones can be combined using semi-colons

Colours and format codes

<Esc> [FormatCode_m

- 0 Reset, all attributes off
 - 1 Bold (or increased intensity)
 - 2 Faint (decreased intensity)
 - 3 Italic { not widely supported, sometimes treated as inverse }
 - 4 Underline
 - 5 or 6 Slow/Rapid Blink
 - 7 Swap foreground and background colours
 - 8 Hidden (useful for passwords)
 - 2{1..8} Reset second digit format (e.g. 24 to stop underlining)*
 - 30 to 37 Set foreground colour
 - 40 to 47 Set background colour
 - 38;5;{0..255} Set foreground colour
 - 48;5;{0..255} Set background colour
 - 90 to 97 Set bright foreground colour
 - 100 to 107 Set bright background colour
- }
- 8 colours
- }
- 256 colours
- }
- 16 colours

* GNOME Terminal 3.28 (VTE 0.52), debuting in Ubuntu 18.04 LTS, adds support for a few more styles. Use code 22 to unbold since 21 is double underline!

Colours and format codes

```
1 $ echo "Default \e[31mRed\e[0m"
Default \e[31mRed\e[0m
3 $ echo -e "Default \e[31mRed\e[0m"
Default Red
5 $ echo -e "Default \e[91mLight Red\e[0m"
Default Light Red
7 $ echo -e "Default \e[46mCyan\e[0m"
Default Cyan
9 $ echo -e "Default \e[106mLight Cyan\e[0m"
Default Light Cyan
11 $ echo -e "Default \e[1mBold\e[0m"
Default Bold
13 $ echo -e "Default \e[4mUnderlined\e[0m"
Default Underlined
15 $ echo -e "Default \e[4m\e[91mUnderlined\e[0m"
Default Underlined
17 $ echo -e "Default \e[4;91mUnderlined\e[0m"
Default Underlined
19 $ printf "Default \e[4;91mUnderlined\e[24m still red\e[0m Default"
Default Underlined still red Default
21 $ echo -e "\e[40;38;5;82m Hello \e[30;48;5;82m World \e[0m"
Hello World
```

Colours and format codes

```
1 $ for fgbg in 38; do # 38 48 to get also background
2 >     for color in {0..255}; do
3 >         printf "\e[${fgbg};5;%sm %3s \e[0m" ${color} ${color}
4 >         if [[ $(((color + 1) % 16)) -eq 0 ]]; then
5 >             echo
6 >         fi
7 >     done
8 >     echo
9 > done
```

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142
144	145	146	147	148	149	150	151	152	153	154	155	156	157	158
160	161	162	163	164	165	166	167	168	169	170	171	172	173	174
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190
192	193	194	195	196	197	198	199	200	201	202	203	204	205	206
208	209	210	211	212	213	214	215	216	217	218	219	220	221	222
224	225	226	227	228	229	230	231	232	233	234	235	236	237	238
240	241	242	243	244	245	246	247	248	249	250	251	252	253	254

Defining variables for colour codes

```
# Reset
Default='\033[0m'

# Regular Colors
Black='\033[0;30m'
Red='\033[0;31m'
Green='\033[0;32m'
Yellow='\033[0;33m'
Blue='\033[0;34m'
Magenta='\033[0;35m'
Cyan='\033[0;36m'
White='\033[0;37m'

# Bold
Bold='\033[1m'
BBlack='\033[1;30m'
BRed='\033[1;31m'
BGreen='\033[1;32m'
BYellow='\033[1;33m'
BBlue='\033[1;34m'
BMagenta='\033[1;35m'
BCyan='\033[1;36m'
BWhite='\033[1;37m'

# Or you can wait to
# learn about functions
# and create your way!

# ...and so on and so forth!
```



Cursor movements

In the same spirit of colours, the terminal cursor position can be moved:

`<Esc> [<L>;<C>H` Puts the cursor at line L and column C

`<Esc> [<L>;<C>f` Puts the cursor at line L and column C

`<Esc> [<N>A` Move the cursor up N lines

`<Esc> [<N>B` Move the cursor down N lines

`<Esc> [<N>C` Move the cursor forward N columns

`<Esc> [<N>D` Move the cursor backward N columns

`<Esc> [2J` Clear the screen, move to (0,0)

`<Esc> [K` Erase to end of line

`<Esc> [s` Save cursor position

`<Esc> [u` Restore cursor position

Cursor movements

Use your imagination to take advantage of this functionality:

```
$ while true; do
>     printf "      $(date) \n\e[1A"
>     sleep 1
> done
^C    Thu 11 Jul 18:07:26 CEST 2019 # Stop it with CTRL-C

# Here a crazy progress bar:
$ while true; do
>     printf '\e[s'
>     printf '%0.s=' $(seq 1 $(bc -l <<< "$RANDOM/32767*100"))
>     sleep 0.2
>     printf '\e[u\e[K' # Use \r instead of saving cursor
> done
=====^C # Stop it with CTRL-C

$ printf 'You will never see my ${password}... MUAHAHA!\r\e[K'
$
```

References and indirect expansion



The geyser Strokkur: 25-35m high every ~10 minutes

The `nameref` attribute

Introduced in 2014, Bash v4.3.0

Reference: `declare -n reference`

The variable is a reference to another variable

- It allows variables to be manipulated indirectly
- Whenever the `reference` variable
 - is referenced, assigned to, unset,
 - or has its attributes modified (other than using or changing the `nameref` attribute itself), the operation is performed on the variable specified by the `reference`'s value!

```
1 $ aVar='Hello'; echo "${aVar}"
Hello
3 $ declare -n bVar='aVar'; echo "${bVar}"
Hello
5 $ bVar='Goodbye'; echo "${aVar}"
Goodbye
7 $ declare +n bVar; echo "${bVar}"
aVar
9 $ unset aVar bVar
```

The `nameref` attribute

Introduced in 2014, Bash v4.3.0

Think before using indirection

Putting variable names or any other bash syntax inside parameters is frequently done incorrectly and in inappropriate situations to solve problems that have better solutions. **It violates the separation between code and data**, and as such puts you on a slippery slope toward bugs and security issues. Indirection can make **your code less transparent and harder to follow**.

Normally, in bash scripting, you won't need indirect references at all. Generally, people look at this for a solution when they don't understand or know about Bash Arrays or haven't fully considered other Bash features such as functions.

Greg's Wiki

Sometimes you might need it

A `nameref` is commonly used within shell functions to refer to a variable whose name is passed as an argument to the function.



Indirect expansion

An alternative to use the `nameref` attribute

- It is about using the content of a variable as name of a different variable
- It is a particular case of parameter expansion: `$(!parameter)`
 - Bash uses the value formed by expanding `parameter` as the new parameter
 - this is then expanded and that value is used in the rest of the expansion

```
1 $ aVar='Hello'; bVar='aVar'
2 $ echo "bVar contains \"${bVar}\\" which contains \"${!bVar}\\""
3 bVar contains "aVar" which contains "Hello"
4 $ unset aVar bVar
```



Indirect expansion

An alternative to use the `nameref` attribute

- It is about using the content of a variable as name of a different variable
- It is a particular case of parameter expansion: `${!parameter}`
 - Bash uses the value formed by expanding `parameter` as the new parameter
 - this is then expanded and that value is used in the rest of the expansion

```
1 $ aVar='Hello'; bVar='aVar'
2 $ echo "bVar contains \"${bVar}\\" which contains \"${!bVar}\\""
3 bVar contains "aVar" which contains "Hello"
4 $ unset aVar bVar
```

- If a `*` or a `@` is put at the end of parameter, the behaviour changes!

`${!prefix@}` `${!prefix*}` } Expands to the names of variables whose
`"${!prefix@}"` } names begin with `prefix` as single word

`"${!prefix*}"` → As above, but words are separated by the first character of the IFS



Indirect expansion

An alternative to use the `nameref` attribute

- It is about using the content of a variable as name of a different variable
- It is a particular case of parameter expansion: `${!parameter}`
 - Bash uses the value formed by expanding `parameter` as the new parameter
 - this is then expanded and that value is used in the rest of the expansion

```
1 $ aVar='Hello'; bVar='aVar'
2 $ echo "bVar contains \"${bVar}\\" which contains \"${!bVar}\\""
3 bVar contains "aVar" which contains "Hello"
4 $ unset aVar bVar
```

- If a * or a @ is put at the end of parameter, the behaviour changes!

`${!prefix@}`
`"${!prefix@}"` }
 `${!prefix*}`

Expands to the names of variables whose
names begin with `prefix` as single word

`"${!prefix*}"` → As above, but words are separated by the first character of the IFS

You will hardly need this!

Arrays and associative arrays



A baby moulin on the Breiðamerkurjökull

Motivation

- Strings are without any doubt the most used parameter type
- They are also the most misused parameter type, though!
- Strings hold just one element!
- Capturing a list in a string is very often plain wrong, even if it might work...
- A parameter contains just one string of characters, no matter the meaning of them
- If you put multiple filenames in a string, maybe to iterate over them at a later point, which delimiter would you use?
- Is there a delimiter that is not accepted to be part of a filename? Mmmh...

```
# This does NOT work in the general case
$ files=$(ls ~/*.jpg); cp ${files} /backups/      # BAD code!
```

Array

An array is a numbered list of strings: It maps integers to strings.

Creating an array

array=(...) The most used syntax: space separated list in parenthesis

```
array=(one two "three and four") # Equivalent to:  
array=([0]=one [1]=two [2]="three and four")  
# Sparse array perfectly legal!  
array=([0]=First [11]=Middle [23]=Last)
```

array[n]= Setting single elements works for undeclared variable, too

```
array[0]=First  
array[11]=Middle  
array[23]=Last
```

declare -a array Rarely used but possible

```
declare -a array # Because of declare,  
array=One          # equivalent to array[0]=One, AVOID!  
array[1]=Two
```

array+=(...) Concatenate r.h.s. array to l.h.s. array

Storing filenames into an array

Take home message

If you want to fill an array with filenames, then you'll probably want to use **glob**s in there!

```
$ files=(~/"My Photos"/*.jpg)
```

- Here we quoted the "My Photos" part because it contains a space
- If we hadn't quoted it, Bash would have split it up into

```
files=(~/My' 'Photos/*.jpg)
```

which is obviously not what we want!

- We quoted **only** the part that contained the space: We cannot quote the ~ or the *
- If we do, they'll become literal and Bash won't treat them as special characters anymore!



Storing filenames into an array

Take home message

If you want to fill an array with filenames, then you'll probably want to use **glob**s in there!

```
# Please, really, use glob instead of commands!

$ files=$(ls)          # BAD, BAD, BAD!
$ files=(${!ls})        # STILL BAD!

# So, how should you do?

$ files=(*)            # GOOD!
```

Globs know about files

Using glob patterns is possible to create an array with filenames in different entries!



Accessing array's content (I)

Definition, single elements and length

`declare -p array` It prints the definition of the variable {not so useful in scripts}

```
$ declare -p array
declare -a array='([0]="one" [1]="two and four")'
```

`${array[n]}` It retrieves the n-th element {no error if entry missing in array}

```
$ echo "${array[1]}"; echo "_${array[3156]}_"
two and four
--
$ echo "${array[0+1]}" # [...] is an arithmetic context
two and four
```

`${#array[@]}` It retrieves the length of the array {same as `${#array[*]}` }

```
$ echo "${#array[@]}"; array[7]=Hello; echo "${#array[@]}";
2 # Quoting the expansion does not change anything!
3
$ echo "${#array[7]}"; echo "${#array[-1]}";
5
5 # What does this mean?
```



Accessing array's content (II)

All elements at once

"\${array[@]}"
Bash replaces this syntax with each element properly quoted

```
$ printf '%s\n' "${array[@]}"
one
two and four
Hello
```

"\${array[*]}"
expands to a IFS-first-character-separated list of the array entries

```
$ IFS=':'; printf '%s\n' "${array[*]}"; unset IFS
one:two and four:Hello
```

Unquoted \${array[@]} and \${array[*]} let word splitting kick in!

```
$ printf '%s\n' ${array[@]} # The same with ${array[*]}
one
two
and
four
Hello
```



Accessing array's content (III)

All indices

"\${!array[@]}" Bash replaces this syntax with the indices of the entries that are set

```
$ printf '%d\n' "${!array[@]}"
0
1
7
```

"\${!array[*]}" expands to a IFS-first-character-separated list of the array indices

```
$ IFS=:; printf '%s\n' "${!array[*]}"; unset IFS
0:1:7
```

In any case, prefer quoted versions!

Unquoted \${!array[@]} and \${!array[*]} let word splitting kick in.
This, for normal arrays, is harmless, since indices are just integers.



Iterating over array elements

- Depending on the needs you can iterate over the elements or the indices in the array
- When iterating over the **elements**, 99% of the time you will need "\${array[@]}"
- When iterating over the **indices**, 99% of the time you will need "\${!array[@]}"
- To implicitly iterate over arrays is possible using different commands (e.g. `printf`, `cp`)

```
1 $ array=(one "two and four")
2 $ for entry in "${array[@]}"; do
3 >     echo "$entry"
4 > done
5 one
6 two and four
7 $ for index in "${!array[@]}"; do
8 >     echo "array[$index]=${array[index]}"
9 > done
10 array[0]=one
11 array[1]=two and four
12 $ unset entry index
13 # Coming back to motivation: GOOD code!
14 $ files=(~/*.jpg); cp "${files[@]}" /backups/
```



Deleting arrays or array elements

- To delete an element of an array, pass it to the `unset` builtin

```
1 $ array=({a..e}); unset 'array[2]' 'array[3]'  
2 $ for index in "${!array[@]}"; do  
3 >     echo "array[$index]=${array[index]}"  
4 > done; unset 'index'  
5 array[0]=a  
6 array[1]=b  
7 array[4]=e
```

- If you pass the name of an array to `unset`, the whole array will be unset!

```
1 $ array=({a..c})  
2 $ for index in "${!array[@]}"; do  
3 >     echo "array[$index]=${array[index]}"  
4 > done; unset 'index'  
5 array[0]=a  
6 array[1]=b  
7 array[2]=c  
8 $ unset array  
9 $ echo "_${array[@]}_ -> ${#array[@]} entries"  
10 -- -> 0 entries
```



Deleting arrays or array elements

Do not forget to quote your variable to avoid file pattern matching!

```
1 $ a=(file{1..3})
2 $ ls
3 a0  Day_1.tex  Day_1.pdf
4 $ a0='Hello'
5 $ printf '%s\n' "${a[@]}"
6 file1
7 file2
8 file3
9 $ echo "${a0}"
10 Hello
11 $ unset a[0] # <--- BAD, BAD, BAD CODE!
12 $ printf '%s\n' "${a[@]}"
13 file1 # Wait! I unset the first element, didn't I?
14 file2
15 file3
16 $ echo "_${a0}_"
17 --                                # Is it clear what happened?
```



Sparse arrays

Arrays might be sparse!



- Don't assume that your indices are sequential
- If the index values matter, always iterate over the indices { instead of making assumptions about them... }
- If you loop over the values instead, don't assume anything about indices
- In particular, don't assume that just because you're currently in the first iteration of your loop, that you must be on index 0...

If you need a non-sparse array, just make it such!

```
$ array=("${array[@]}")
```

Associative arrays

Since Bash v4.0 (2009)

- An associative array is a map of strings to strings
- It has to be explicitly declared as such via `declare -A`
- Since keys are strings, they might contain spaces!
⇒ Iterating over keys requires quotes: `"${!array[@]}"`

```
1 $ declare -A dict
2 $ dict[Apartment]="Die Wohnung"
3 $ declare -p dict
4 declare -A dict='([Apartment]="Die Wohnung" )'
5 $ dict[Water]="Das Wasser"
6 $ for key in "${!dict[@]}"; do
7 >     printf 'EN: %20s -> DE: %s\n' "${key}" "${dict[$key]}"
8 > done
9 EN:           Water -> DE: Das Wasser
10 EN:           Apartment -> DE: Die Wohnung
11 $ echo "${#dict[@]}"; unset dict
12 2
13 $ echo "${#dict[@]}"
14 0
```

Associative arrays

Since Bash v4.0 (2009)

- An associative array is a map of strings to strings
- It has to be explicitly declared as such via `declare -A`
- Since keys are strings, they might contain spaces!
⇒ Iterating over keys requires quotes: `"${!array[@]}"`

```
15 $ declare -A dict
16 $ dict[Die Wohnung]="Apartment"
17 $ dict[Das Wasser]="Water"
18 $ for key in ${!dict[@]}; do      # AAAAAARGH!
19 >     printf 'DE: %20s -> EN: _%s_\n' "${key}" "${dict[$key]}"
20 > done
21 DE:           Die -> EN: --
22 DE:           Wohnung -> EN: --
23 DE:           Das -> EN: --
24 DE:           Wasser -> EN: --
25 $ echo "${#dict[@]}"; unset -v 'dict[Das Wasser]'
26 2
27 $ echo "${#dict[@]}"; unset -v 'dict'
28 1
```



Associative arrays: Remarks

Since Bash v4.0 (2009)

- The order of both keys and elements of an associative array is unpredictable
⇒ They are not well suited to store lists that need to be processed in order
- If you use a parameter as key of an associative array, you must use the \$ sign
⇒ Indeed, it makes sense: The name of the parameter might simply be a valid key

```
1 $ declare -A array
2 $ index=1; for entry in one two three four five six; do
3 >     array[$entry]=${index}
4 > done
5 $ echo "${array[@]}"
6 4 1 5 6 2 3
7 $ echo "${!array[@]}"
8 four one five six two three
9 $ unset array entry index
```



Associative arrays: Remarks

Since Bash v4.0 (2009)

- The order of both keys and elements of an associative array is unpredictable
⇒ They are not well suited to store lists that need to be processed in order
- If you use a parameter as key of an associative array, you must use the \$ sign
⇒ Indeed, it makes sense: The name of the parameter might simply be a valid key

```
10 $ indexed=("one" "two")    index=0   key="foo"
11 $ declare -A associative=([foo]=bar [alpha]=omega)
12 $ echo "${indexed[$index]}"
13 one
14 $ echo "${indexed[index]}"
15 one
16 $ echo "${indexed[index + 1]}"
17 two
18 $ echo "${associative[$key]}"
19 bar
20 $ echo "_${associative[key]}_"
21 --
22 $ echo "_${associative[key + 1]}_"
23 --
24 $ unset indexed associative index key
```



The power of parameter expansion on arrays

- Arrays are very flexible, because they are well integrated with the other shell expansions
- Any parameter expansion can be carried out on individual array elements
- Parameter expansions can equally well apply to an entire array!
- To use parameter-expansion manipulations on all array entries, use `[@]` and `[*]`
- It is critical that these special expansions are properly quoted

```
1 $ array=(alpha beta gamma)
2 $ echo "${array[-1]:2:2}"      # Substring of last entry
3 mm
4 $ echo "${array[@]#a}"          # Chop 'a' from the beginning
5 lpha beta gamma
6 $ echo "${array[@]%a}"          # Chop 'a' from the end
7 alph bet gamm
8 $ echo "${array[@]//a/_}"        # Substitute all 'a' by '_'
9 _lph_ bet_ g_mm_
10 $ echo "${array[@]/#a/_}"       # Substitute 'a' by '_' at start
11 _lpha beta gamma
12 $ echo "${array[@]%/a/_}"       # Substitute 'a' by '_' at end
13 alph_ bet_ gamm_
```

The power of parameter expansion on arrays

- Arrays are very flexible, because they are well integrated with the other shell expansions
- Any parameter expansion can be carried out on individual array elements
- Parameter expansions can equally well apply to an entire array!
- To use parameter-expansion manipulations on all array entries, use `[@]` and `[*]`
- It is critical that these special expansions are properly quoted

```
14 $ echo "${array[@]#/prefix_}" # Add prefix to all entries
15 prefix_alpha prefix_beta prefix_gamma
16 $ echo "${array[@]%/[_suffix]}" # Add suffix to all entries
17 alpha_suffix beta_suffix gamma_suffix
18 $ echo "${array[@]^^}"           # Capitalise all characters
19 ALPHA BETA GAMMA
20 $ echo "${array[3]}?Third entry is unset}"
21 bash: array[3]: Third entry is unset
22 $ echo "${array[3]=delta}"
23 delta
24 $ echo "${array[@]^}"
25 Alpha Beta Gamma Delta
26 $ unset array
```

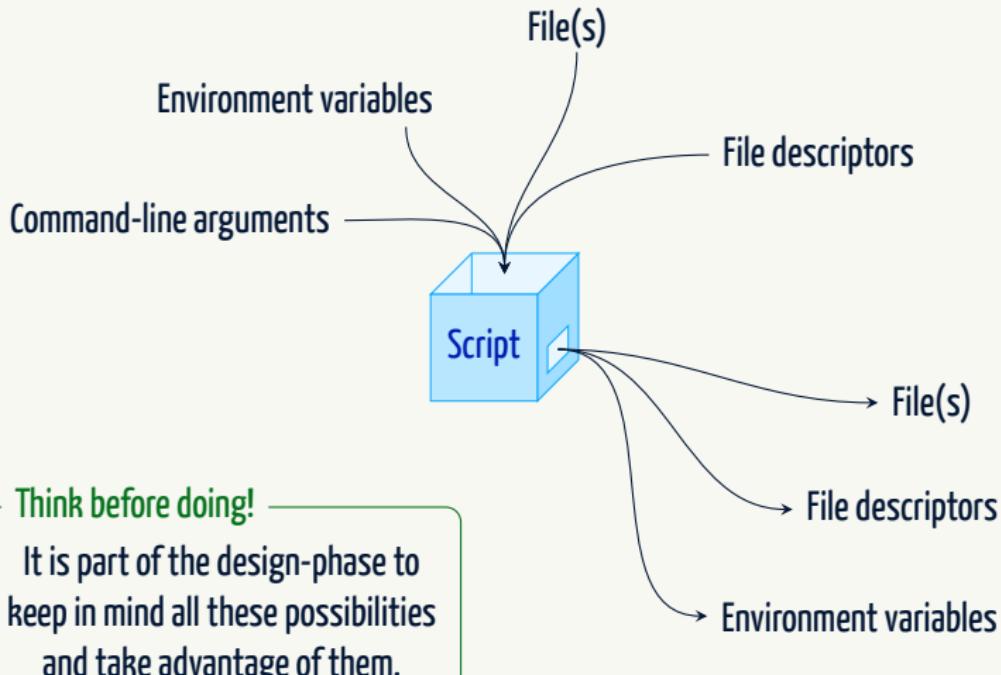


Input and output

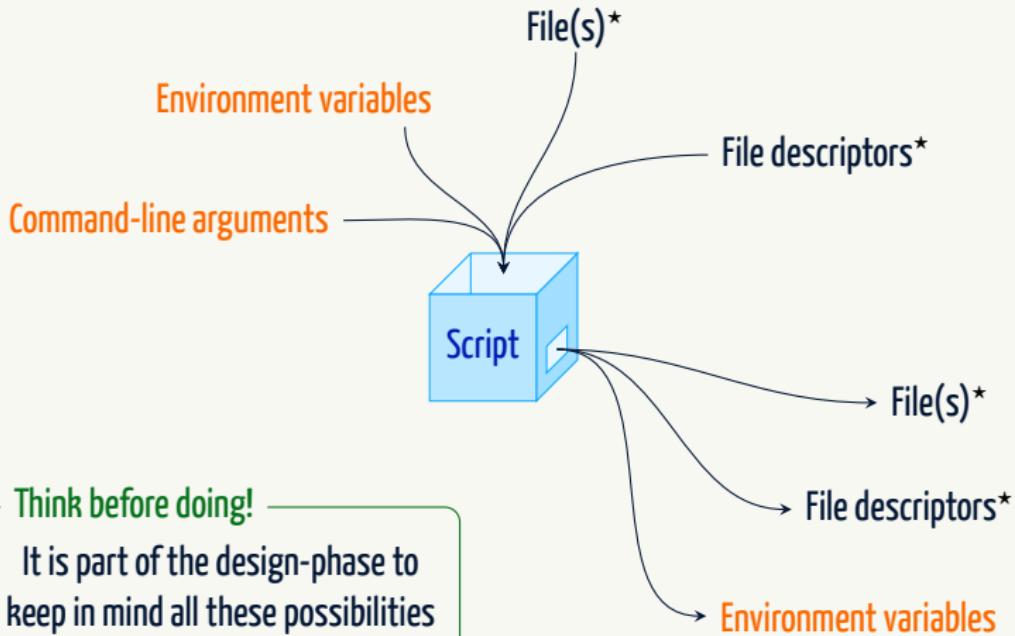


The Krisuvikurberg cliffs at the sunset

The Bash script flow



The Bash script flow



Command-line parameters

- They are accessible via `$1`, `$2`, etc.
- After the 9th one, you need curly braces: `${10}`, `${11}`, etc.
- It is possible to refer to all of them via `$@` and `$*`
- When you refer to all of them, especially to pass them over, use `"$@"`
 - the double quotes are crucial to preserve the parameters without splitting them!
- The `shift` built-in is remarkably handy when parsing command-line parameters
 - it destroys `$1` and it maps `$2` into `$1`, `$3` into `$2` and so on

```
1 $ set -- Hello my "nice world"
2 $ printf '%s\n' "$@"
3 Hello
4 my
5 nice world
6 $ printf '%s\n' $@    # <-- Probably, AAAAARGH!
7 Hello
8 my
9 nice
10 world
```



Command-line parameters

- They are accessible via `$1`, `$2`, etc.
- After the 9th one, you need curly braces: `${10}`, `${11}`, etc.
- It is possible to refer to all of them via `$@` and `$*`
- When you refer to all of them, especially to pass them over, use `"$@"`
→ the double quotes are **crucial** to preserve the parameters without splitting them!
- The `shift` built-in is remarkably handy when parsing command-line parameters
→ it destroys `$1` and it maps `$2` into `$1`, `$3` into `$2` and so on

```
11 $ echo "$1"; shift
12 Hello
13 $ echo "$1"; shift
14 my
15 $ echo "$1"; shift
16 nice world
17 $ echo _"$1"_  

18 --
19 $ set -- Hello my "nice world"; shift 2; echo "$1"
20 nice world
```

Cool!

Environment variables (I)

- Every program inherits certain information from its parent process {resources, privileges and restrictions}
- One of those resources is a set of variables called Environment Variables
- Traditionally, environment variables have names that are all capital letters, such as PATH
- When you run a command in Bash, you have the option of specifying a temporary environment change which only takes effect for the duration of that command
- This is done by putting VAR=value in front of the command

```
1 $ ls /tpm
2 ls: cannot access '/tpm': No such file or directory
3 $ LANGUAGE=de_DE.utf-8 ls /tpm # ↗Read more about LANGUAGE
4 ls: Zugriff auf '/tpm' nicht möglich: Datei oder
   Verzeichnis nicht gefunden
5 $ VERBOSE=1 make
```

Good practice

Don't use all-capital variable names in your scripts, unless they are environment variables. Use lower-case or mixed-case variable names, to avoid accidents.

Environment variables (II)

- In a script, you can use environment variables just like any other variable

```
if [[ ${EDITOR} ]]; then
    ${EDITOR}
else
    emacs
fi
```

- To change the environment for your child processes to inherit, use the `export` builtin

```
export PATH=${HOME}/.local/bin:$PATH
```

Remember!

The tricky part here is that your environment changes are only inherited by your descendants. You can't change the environment of a program that is already running, or of a program that you don't run.

...mmmh, and if I needed it?



Environment variables (III)

- Sourcing a script, will execute it in the current environment/shell

```
$ cat script.bash
#!/bin/sh
cd /tmp
$ pwd; ./script.bash; pwd
/home/sciarra/Documents
/home/sciarra/Documents
$ pwd; source script.bash; pwd # 'source' as '.'
/home/sciarra/Documents
/tmp
```

Indeed, this is what you do e.g. when you add code in the `$(HOME)/.bashrc` file

Splitting a large script in several files

Although a script should not be huge, it is important sometimes to split it into pieces for handier development. This can be done using the `source` builtin, delegating to the main (executable) script to source all secondary files. We will come back to this point when we introduce functions.

Shell options



The Þingvellir national park seen from above

Modifying the shell behaviour

- There are some shell features that can be activated, if needed
- There are two builtins that are responsible for that:

`set` associated to the `SHELLOPTS` environment variable

`shopt` associated to the `BASHOPTS` environment variable

Why two?

Historically, the `set` command was used to turn options on and off. As the number of options grew, `set` became more difficult to use because options are represented by single letter codes. As a result, Bash provides the `shopt` (shell option) command to turn options on and off by name instead of a letter. You can set certain options only by letter. Others are available only under the `shopt` command.

This makes finding and setting a particular option a confusing task.

Linux Shell Scripting with Bash

- The `set` builtin allows you to also set positional parameters → it can be very handy!



Modifying the shell behaviour

- There are some shell features that can be activated, if needed
- There are two builtins that are responsible for that:

`set` associated to the `SHELLOPTS` environment variable

`shopt` associated to the `BASHOPTS` environment variable

Gilles says... —

As far as I know, the `set -o` options are the ones that are inherited from other Bourne-style shells (mostly `ksh`), and the `shopt` options are the ones that are specific to bash. *There's no logic that I know of.*

Well, there are `set -o` options like `posix`, `physical`, `interactive-comments` that are not in `ksh`, and `shopt` ones that are in other shells including `ksh` for some like `login_shell` or `nullglob`. Like you say, there's no logic. It was probably the idea at the start (that `SHELLOPTS` would be the standard ones, and `BASHOPTS` the bash specific ones), but that got lost along the way, and now it just ends up being annoying and a UI design fiasco.

...Stéphane comments



The set builtin

🔗 Bash manual v5.0 section 4.3.1

- Without arguments, `set` displays the names and values of all shell variables/functions.
- When options are supplied, they set or unset shell attributes.
- Using + rather than – causes these options to be turned off.
- The `set` options can also be used upon invocation of the shell.
- The current set of options may be found in `$-`.
- The remaining N arguments are positional parameters and are assigned, in order, to `$1`, `$2`, ..., `$N`. The special parameter `#` is set to N.
- If you want to set positional arguments that start by a dash, you need to use `--`.



The set builtin

🔗 Bash manual v5.0 section 4.3.1

- Without arguments, `set` displays the names and values of all shell variables/functions.
- When options are supplied, they set or unset shell attributes.
- Using `+` rather than `-` causes these options to be turned off.
- The `set` options can also be used upon invocation of the shell.
- The current set of options may be found in `$-`.
- The remaining `N` arguments are positional parameters and are assigned, in order, to `$1`, `$2`, ..., `$N`. The special parameter `#` is set to `N`.
- If you want to set positional arguments that start by a dash, you need to use `--`.

```
1 $ echo $-; set -u; echo $-; set +u; echo $-
2 himBHs
3 himuBHs
4 himBHs
5 $ echo $#; set -- -s --long; echo $#; echo $@
6 0
7 2
8 -s --long
```



Some among the many available options (I)

- o **noglob** It is the equivalent to **-f**.
Disable filename expansion (globbing).
- o **noclobber** It is the equivalent to **-C**.
Prevent output redirection from overwriting existing files.
- o **noexec** It is the equivalent to **-n**.
Read commands but do not execute them. { This may be used to check a script for syntax errors. }
- o **nounset** It is the equivalent to **-u**.
Treat unset variables and parameters other than the special parameters @ or * as an error when performing parameter expansion. An error message will be written to the standard error, and a non-interactive shell will exit.
- o **pipefail** If set, the return value of a pipeline is the value of the last (rightmost) command to exit with a non-zero status, or zero if all commands in the pipeline exit successfully. This option is disabled by default.

Some among the many available options (I)

-o `errexit` It is the equivalent to `-e`.

Roughly speaking, this option makes the shell exit immediately when a command returns a non-zero status. This is not always true, though.

-o `errtrace` It is the equivalent to `-E`.

-o `functrace` It is the equivalent to `-T`.

An attempt to simplify error handling

We will come back to these options and discuss them in detail. Although setting `-e` enables a nice feature, it also switches on lots of drawbacks that often need special handling.

Some among the many available options (I)

-o `errexit` It is the equivalent to `-e`.

Roughly speaking, this option makes the shell exit immediately when a command returns a non-zero status. This is not always true, though.

-o `errtrace` It is the equivalent to `-E`.

-o `functrace` It is the equivalent to `-T`.

An attempt to simplify error handling

We will come back to these options and discuss them in detail. Although setting `-e` enables a nice feature, it also switches on lots of drawbacks that often need special handling.

To me it actually seems like...



...a nice try!

The shopt builtin

🔗 Bash manual v5.0 section 4.3.2

```
shopt [-pqsu] [-o] [optname ...]
```

-s Enable (set) each optname.

-u Disable (unset) each optname.

-q No output; the return status indicates whether the optname is set or unset.

{ If multiple optname arguments are given with -q, the return status is zero if all optnames are enabled; non-zero otherwise. }

-p A list of all settable options is displayed, with an indication of whether or not each is set; if optnames are supplied, the output is restricted to those options.

{ Without options the behaviour of shopt is the same as with the -p option }

-o Restricts optname to be one of values of the -o option of the set builtin.

```
1 $ shopt extglob nullglob
2 extglob          on
3 nullglob         off
4 $ shopt -p extglob
5 shopt -s extglob
```



Some among the many available options (II)

Here focusing on those about globbing

dotglob If set, Bash includes filenames beginning with a `.` in the results of filename expansion.

{ The filenames `.` and `..` must always be matched explicitly, even if `dotglob` is set }

extglob If set, the extended pattern matching features (described yesterday) are enabled.

failglob If set, patterns which fail to match filenames during filename expansion result in an expansion error.

globstar If set, the pattern `**` used in a filename expansion context will match all files and zero or more directories and subdirectories. If the pattern is followed by a `/`, only directories and subdirectories match. { This option exists since bash v4.0. }

nullglob If set, Bash allows filename patterns which match no files to expand to a null string, rather than themselves.



Some among the many available options (II)

Here focusing on those about globbing

```
1 $ ls -a
2 . .. .hiddenFile  dir1  file1.c  file2.dat  file3.txt
3 # Example of dotglob
4 $ shopt -s dotglob; echo *
5 .hiddenFile dir1 file1.c file2.dat file3.txt
6 $ shopt -u dotglob; echo *
7 dir1 file1.c file2.dat file3.txt
8 # Example of failglob
9 $ shopt -s failglob; echo *.pdf  # echo command not executed!
10 bash: no match: *.pdf
11 $ shopt -u failglob; echo *.pdf
12 *.pdf
13 # Example of nullglob
14 $ shopt -s nullglob; echo *.pdf
15
16 $ shopt -u nullglob; echo *.pdf
17 *.pdf
18 $ shopt -s extglob  # We saw it yesterday!
19 $ echo !(*.c|*.dat)
20 dir1 file3.txt
21 $ shopt -u extglob
```



Some among the many available options (II)

Here focusing on those about globbing

```
22 $ ls -a
23 . ... .hiddenFile  dir1  file1.c  file2.dat  file3.txt
24 $ ls -a dir1/
25 . ... dir2  file4.c
26 $ ls -a dir1/dir2/
27 . ... file5.c
28 # Example of globstar
29 $ shopt -u globstar; echo **
30 dir1 file1.c file2.dat file3.txt
31 $ shopt -s globstar; echo **
32 dir1 dir1/dir2 dir1/dir2/file5.c dir1/file4.c file1.c file2.dat file3.txt
33 $ echo */
34 dir1/ dir1/dir2/
35 $ echo **/*.c
36 dir1/dir2/file5.c dir1/file4.c file1.c
37 $ echo **.c  # ATTENTION: Equivalent to  *.c/*.c/*.*c/...
38 file1.c
```



Some among the many available options (II)

Here focusing on those about globbing

```
# In a script to be sourced remember whether extglob was originally set
shopt -q extglob; extglobSet=$?
# set extglob if it wasn't originally set
[[ ${extglobSet} -ne 0 ]] && shopt -s extglob
# ...
# unset extglob if it wasn't originally set
[[ ${extglobSet} -ne 0 ]] && shopt -u extglob
```



Some among the many available options (II)

Here focusing on those about globbing

```
# In a script to be sourced remember whether extglob was originally set
shopt -q extglob; extglobSet=$?
# set extglob if it wasn't originally set
[[ ${extglobSet} -ne 0 ]] && shopt -s extglob
# ...
# unset extglob if it wasn't originally set
[[ ${extglobSet} -ne 0 ]] && shopt -u extglob
```

Some options affect the parser behaviour!

Some options changes the way certain characters are parsed. It is necessary to have a newline (not just a semicolon) between e.g. `shopt -s extglob` and any subsequent commands to use it. You cannot enable such options inside a compound command that uses them, because the entire block is parsed before the `shopt` is evaluated. Note that the typical function body (or an `if`-clause) is a compound command.

This is also why you cannot run

```
shopt -s extglob; ls !(*.txt) # WRONG as one-liner
```

(when `extglob` is previously unset), but must have a newline between the two commands.

The GNU core utilities and util-linux



Walking on the Breiðamerkurjökull

They are not part of bash but they naturally interact with it

GNU coreutils

The GNU Core Utilities are the basic file, shell and text manipulation utilities of the GNU operating system. These are the core utilities which are expected to exist on every operating system.

The util-linux package

util-linux is a standard package distributed by the Linux Kernel Organization for use as part of the Linux operating system.

They are not part of bash but they naturally interact with it

GNU coreutils

The GNU Core Utilities are the basic file, shell and text manipulation utilities of the GNU operating system. These are the core utilities which are expected to exist on every operating system.

The util-linux package

util-linux is a standard package distributed by the Linux Kernel Organization for use as part of the Linux operating system.

What should I do here?!

The [v8.32 GNU coreutils manual](#) from February 2020 has **297 pages** and there are **98 available commands**. There are more than **100 commands** in the **util-linux** package.

Don't panic. Discover, read the manuals and learn when you need it!

They are not part of bash but they naturally interact with it

GNU coreutils

The GNU Core Utilities are the basic file, shell and text manipulation utilities of the GNU operating system. These are the core utilities which are expected to exist on every operating system.

arch	cut	false	logname	od	runcon	tee	uptime
base64	date	fmt	ls	paste	seq	test	users
basename	dd	fold	md5sum	pathchk	shred	timeout	vdir
cat	df	groups	mkdir	pinky	shuf	touch	wc
chcon	dir	head	mkfifo	pr	sleep	tr	who
chgrp	dircolors	hostid	mknod	printenv	sort	true	whoami
chmod	dirname	hostname	mktemp	printf	split	truncate	yes
chown	du	id	mv	ptx	stat	tsort	
chroot	echo	install	nice	pwd	stdbuf	tty	
cksum	env	join	nl	readlink	stty	uname	
comm	expand	kill	nohup	realpath	sum	unexpand	
cp	expr	link	nproc	rm	tac	uniq	
csplit	factor	ln	numfmt	rmdir	tail	unlink	

They are not part of bash but they naturally interact with it

GNU coreutils

The GNU Core Utilities are the basic file, shell and text manipulation utilities of the GNU operating system. These are the core utilities which are expected to exist on every operating system.

The highlighted commands are those that have been touched at least ones during the lecture (some more are discovered in the exercises).

arch	cut	false	logname	od	runcon	tee	uptime
base64	date	fmt	ls	paste	seq	test	users
basename	dd	fold	md5sum	pathchk	shred	timeout	vdir
cat	df	groups	mkdir	pinky	shuf	touch	wc
chcon	dir	head	mkfifo	pr	sleep	tr	who
chgrp	dircolors	hostid	mknod	printenv	sort	true	whoami
chmod	dirname	hostname	mktemp	printf	split	truncate	yes
chown	du	id	mv	ptx	stat	tsort	
chroot	echo	install	nice	pwd	stdbuf	tty	
cksum	env	join	nl	readlink	stty	uname	
comm	expand	kill	nohup	realpath	sum	unexpand	
cp	expr	link	nproc	rm	tac	uniq	
csplit	factor	ln	numfmt	rmdir	tail	unlink	

Some GNU core utilities

🔗 Cheat-sheet

`basename` Strip directory and, optionally, a suffix from filenames.

`comm` Compare two sorted files line by line.

`dirname` Print a filename with its last non-slash component and trailing slashes removed.

`expand` Convert tabs to spaces.

`fmt` Reformat each paragraph in the file(s), writing to standard output.

`fold` Wrap each input line to fit in specified width.

`ln` Make links between files.

`nl` Write each input file to standard output, with line numbers added.

`realpath` Print the resolved absolute file name.

`tac` Write each input file to standard output in a reversed order (last line first).

`uniq` Report, omit or count repeated lines in a file

`unexpand` Convert spaces to tabs.



Some commands from the util-linux package

The project is available [on GitHub](#) and a list of the commands on [Wikipedia](#)

- This package is much more OS oriented.
- Just have a look to it if you are curious, otherwise the following will be enough!

You can do better than that!

At the end of this lecture you will have learnt to implement even more flexible versions of the following commands. To know they exist might still be handy in some occasions, though.

`cal` Display a calendar and the date of Easter.

`column` Format the input into multiple columns.

`getopt` Parse command options.

`rename` Rename the specified files by replacing the first occurrence of an expression in their name by a replacement.

Ready to script!?



Yoda meditating in the Jökulsárlón

- Is it a Bash script? Take advantage of it! { Use a proper shebang! }
- You learnt all flow constructs, keep them in mind
- Use meaningful, lower-case (or mixed-case) variable names!
- Does your script accept command line options? Implement a `-h` option!
- Does your script takes files on the command line? Implement the `--` mechanism.
- If environment variables affect the script behaviour, document it
- Do you need to set particular shell options? Make a note for maintenance!
- Do the error messages get printed to the standard error?
- Take advantage of colours, if needed
- Do not make assumptions, be paranoid
- Use (associative) arrays for collections
- Avoid unnecessary pipes, use herestrings!
- Is the script getting large? Split it! { More tomorrow with functions }
- ...

**ALWAYS CODE AS
IF THE GUY WHO
ENDS UP
MAINTAINING
YOUR CODE WILL
BE A VIOLENT
PSYCHOPATH WHO
KNOWS WHERE
YOU LIVE.**