Shell Scripting with Bash

Paweł Kierat

Introduction

What is Bash?

Bash = **B**ourne-**a**gain **sh**ell

- Command-line interpreter
- Powerful scripting language
- Successor of Bourne shell (sh)
- Written by Brian Fox for the GNU Project
- First release in 1989

What do I need it for?

- Administration and maintenance of UNIX machines
- Simple automation of frequent command-line tasks
- Startup/shutdown scripts for server-side apps
- Building more complex tools from basic commands
- Working effectively with files and directories
- Simple and advanced processing of text files
- Many more…

Runtime Environment

- UNIX-like (Linux, Mac, *BSD, Solaris)
  - Preinstalled
- MS Windows (options)
  - GNU/Linux in a VM (e.g. VirtualBox)
  - Windows Subsystem for Linux (Windows 10+)
  - Cygwin: https://www.cygwin.com
  - MSYS: http://www.mingw.org/wiki/msys
  - Git Bash: https://gitforwindows.org
    - contains MSYS

First script!

- Open your favourite text editor for programmers, e.g.
  - Vim, Emacs (available for all systems)
  - Kate, Gedit (GNU/Linux, *BSD, Solaris)
  - TextMate (Mac)
  - Notepad++ (Windows)
- Create a file: `myscript.sh`
- Type

```
#!/bin/bash
echo "My first Bash script!"
```

- Save and run in Bash console

```
$ bash ./myscript.sh
```

Basics

Printing on the screen:

```
echo "Enter your name:"
```

Reading data from keyboard (into a variable):

```
read NAME
```

Using variables:

```
echo "Hello, ${NAME}!"
```

Setting variables:

```
LANG=en
```

Capturing the output of a command:

```
TODAY=`date +%A`           # `` are backquotes
echo "Today is ${TODAY}."
```

Simple conditional processing:

```
if [ "${NAME}" == "" ]
then
    echo "Name cannot be empty!"
fi
```

Simple `case' instruction:

```
case "${NAME}" in
    "Alice")
        echo "You look  gorgeus today!"
        ;;
    "Bob")
        echo "Nice suite!"
        ;;
    *)
        echo "Pleased to meet you!"
esac
```

Simple `while' loop:

```
while [ "${NAME}" == "" ]
do
   echo "Enter your name:"
   read NAME
done
```

Simple `for-each' loop:

```
echo "Seasons:"
for SEASON in "Spring" "Summer" "Fall" "Winter"
do
    echo "${SEASON}"
done
```

Simple function:

```
say_bye() {
    echo "Bye, ${1}!"
}
say_bye ${NAME}
```

Commands

General Syntax

```
command_name [options] [args...]
```

- `command_name` - name of the executable file or built-in command
- `args` - arguments of the command
- `options` - special arguments for custom configuration

Commands - useful examples

Command	Description
env	print the environment variables
cp, mv, rm	copy, move/rename, delete files
mkdir, rmdir	create, remove a directory
chmod, chown	change various permissions
touch	change file timestamps
truncate	shrink or extend file size
pwd	check current directory path
ls	list files in the directory
find	find files in a directory tree
grep	search in file by regular expr.
head, tail	write first/last N lines of input
cut, sed, awk	process the file line by line
tar, gzip, gunzip	pack, compress, decompress files
ps, jobs	list processes, jobs
kill	send a signal to a process
sleep	pause execution for a fixed time
which	get the full path of a command
xargs	pass stdin as args to a command
date	print/set the date and time
time	measure execution time
man	shows manual for a command

Arguments

- Space-separated list of words

```
cat file1.txt file2.txt file3.txt
```

- Spaces within quotes (" ", ' ', \` \`) are not separators

```
grep 'John Doe' data.csv
```

- Escaped spaces (` `\ ") are not separators

```
rm /tmp/file\ with\ spaces\ in\ name.txt
```

- Max. number of arguments depends on the OS

Options

- Alter the default behaviour of a command
- Traditional (POSIX) options
  - one-letter long, preceded by a single dash (`-`)
  - can have associated value, e.g. `-o file.txt`
  - can be combined: `-abc = -a -b -c`
- Long (X11-style) options
  - keyword options preceded by single dash
  - can have associated value, e.g. `-depth 1`
- Long (GNU) options
  - keyword options preceded by double dash (` `--')
  - value assigned using `=, e.g. --output=file.txt`

Exit status

- Every command returns an exit status code
- Info for the parent, how the command ended
  - 0 - execution ended successfuly
  - other - execution ended unsuccessfully
- Unsuccessful ending may mean
  - an error (e.g. missing argument)

  - abnormal termination (e.g. process was killed)
  - negative result (e.g. pattern not found in file)
- Exact number may give more details
  - e.g. code == 2 - invalid usage (user's mistake)
- Exit code can be returned via
  - `exit` - for the whole script
  - `return` - exit status of a function

```
exit 0
```

- Exit code of last command can be checked using special parameter: `$?`

```
ls -yz      # Error: unrecognized options
echo $?
```

- `true` and `false` are special commands
  - always returning 0 and 1 respectively

Examples

- `[ ··· ]` in `if` statement is not special syntax
  - it's just a command

```
$ [ 1 == 2 ]
$ echo $?
1
```

- `if` and `while` can use any command as a condition

```
while read NAME
do
    echo "Hello, ${NAME}!"
done
```

Parameters

Parameters - entities that store values

- Variables
  - parameters denoted by name
- Positional parameters
  - denoted by numbers
- Special parameters
  - denoted by special symbols

Variables

- *Local variables* - limited to the scope of function
- *Process variables* - available in the current process
- *Shell variables* - provided by the shell
- *Environment variables* - inherited from the parent process
- Setting the value: `name=value`
  - Note: no spaces around `=' (!)
- Accessing the value: `$name or ${name}`
  - Both are valid, the latter is recommended
- Changes visible only in the scope of process

Positional arguments

- `$0` - name of the script used during the call
- `$1, $2, $3, ⋯` - actual arguments

```
echo "My first Bash script with parameters:" $1 $2 $3
```

- `shift` moves all args to the left (except `$0`)
  - `$1` is dropped, `$2` becomes `$1`, `$3` becomes `$2` etc.

```
echo "Argument 1: $1"
shift
echo "Argument 2: $1"
```

Special parameters

- $* - all arguments starting from 1, each as a separate word
  - "$*" resolves to: "$1 $2 $3…"
- $@ - all arguments starting from 1, each as a separate word
  - "$@" resolves to: "$1" "$2" "$3" …
- $# - total number of arguments (excluding $0)
- $_ - last argument of the last command
- $? - exit code of the last command

Quoting

- String within double quotes (" ") can contain variables, expressions etc. which are replaced with actual values

```
echo "The name of this script is: ${0}"
The name of this script is: myscript.sh
```

- String within single quotes (' ') is treated literally (no evaluation)

```
echo 'The name of this script is: ${0}'
The name of this script is: ${0}
```

- String within backquotes (` `) is treated as a command and replaced with its output
- All types of quotes can be multi-line

Escaping

Backslash ('\') before the character blocks special meaning

- applies only to one, immediately following character

```
echo "Regular:" ${0}
echo "Escaped:" \${0}
```

- Even line-feed can be escaped
  - used to break long commands into several lines

```
echo "This is a very long command" \
     "broken into lines."
```

Running commands

- One command per line

```
cat file1.txt
```

- Run one command after another

```
cat file1.txt ; cat file2.txt
```

- Run the second only if the first succeeded

```
cat file1.txt && cat file2.txt
```

- Run the second only if the first failed

```
cat file1.txt || cat file2.txt
```

- pass the output to another command (pipeline)

```
grep 'John' data.csv | wc -l
```

- write the output to a file (overwrite)

```
ls -la > listing.txt
```

- write the output to a file (append)

```
ls -la >> listing.txt
```

- Run the first one in background

```
cat file1.txt & cat file2.txt
```

Exercise

Write a script that will:

- ask the user for employee data
  - first/last name
  - position
  - salary
- write the data as one row in a CSV file
- ask the user, whether to enter another employee

Example:

```
$ ./employees.sh
First Name: James T.
Last Name: Kirk
Position: Captain
Salary: 15000
Continue (Y/n)? Y
...
```

employees.csv:

```
James T.,Kirk,Captain,15000
Jean-Luc,Picard,Captain,17000
```

Input/Output

Basic concepts

- Streams
  - `stdin` - Standard Input stream
    - read-only
    - by default, attached to the keyboard
  - `stdout` - Standard Output stream
    - write-only
    - by default, attached to the screen
  - `stderr` - Standard Error stream
    - write-only
    - by default, attached to the screen
    - separate from `stdout`

Reading

- Read a line from `stdin` into a variable

```
read LINE        # Note: no '$' here!
echo ${LINE}
```

- Read a line and split it by comma (',')

```
IFS=',' read FIELD_1 FIELD_2
echo "Field 1: ${FIELD_1}, Field 2: ${FIELD_2}"
```

- Process the whole input, line by line

```
while read LINE ; do echo ${LINE} ; done
```

- Read whole file into a variable

```
FILE=$(<input.txt)
```

- Read the output of a command into a variable

```
COUNT=$(ls -1 *.txt | wc -l)
```

Writing

- Write a line to standard output

```
echo "Some text"
```

- Write a text to `stdout` without trailing end-of-line

```
echo -n "Some text..."
```

- Use special ASCII characters

```
echo -e "Some text\rXXXX..."
```

- Print formatted text

```
printf "Price: %10.2f" 1234,567
```

- Send the content of the files to `stdout` (in order)

```
cat file1.txt file2.txt file3.txt
```

Pipelines

Pipeline is a sequence of commands with output of one command passed as an input to the next command

```
grep 'John Doe' data.csv | wc -l
```

- All commands in the pipeline are started simultaneously (as subprocesses)
- Subsequent commands block on reading from their input stream

Useful commands

- Limit the input to first 10 lines

```
... | head -n 10 | ...
```

- Limit the input to last 10 lines

```
... | tail -n 10 | ...
```

- Find lines that match a pattern (regular expr.)

```
... | grep 'John' | ...
```

- Extract column(s) from a delimited file

```
... | cut -d ',' -f 1 | ...
```

- Replace all occurences of a string with another one

```
... | sed -e 's/John/Jack/' | ...
```

- Sort the lines and filter out duplicates

```
... | sort -u | ...
```

- Count the characters (suppress normal output)

```
echo -n "This is a very long text." | wc -c
```

- Copy the input to a file (without breaking the pipeline)

```
grep 'John' data.csv | tee copy.csv | ...
```

Streams

- A way of passing data from and to the command process
- Accessing streams is similar to accessing regular files
- sequential processing, no random access

Stream	Descriptor	Device
Standard Input	0	/dev/stdin
Standard Output	1	/dev/stdout
Standard Error	2	/dev/stderr

Redirection

- Redirect `stdout` to a file - overwrite

```
# COMMAND > FILE
ls -la > listing.txt
```

- Redirect `stdout` to a file - append

```
# COMMAND >> FILE
ls -la >> listing.txt
```

- Redirect `stderr` to a file - overwrite

```
# COMMAND 2> FILE
ls -yz 2> errors.log
```

- Redirect `stderr` to a file - append

```
# COMMAND 2>> FILE
ls -yz 2>> errors.log
```

- Read a file and pass to `stdin` of the command

```
# COMMAND < FILE
cat < listing.txt
```

- Read lines below the command down to the delimiter and pass to `stdin` of the command

```
# COMMAND << DELIMITER
cat << EOF
Some text...
EOF
```

- pass a single string to `stdin` of the command

```
# COMMAND <<< "TEXT"
cat <<< "Some text..."
```

- Multiple redirections for one command

```
cat <input.txt 2>error.txt >output.txt
```

- Join standard error with standard output

```
# COMMAND 2>&1
ls -la 2>&1
```

- Join `stderr` with `stdout` and pass them both to `stdin` of next command (pipeline)

```
# COMMAND_1 |& COMMAND_2 ("|&" is the same as "2>&1 |")
ls -la |& tee listing.txt
```

Exercise

Given a CSV file called `employees.csv':

```
EMP_ID,FULL_NAME,AGE,PHONE
1,Jack Sparrow,34,555-1111
2,Hektor Barbossa,21,555-2222
3,Joshamee Gibbs,52,555-3333
4,Will Turner,43,555-4444
5,Elizabeth Swann,18,555-5555
```

And a CSV file called `salaries.csv':

```
EMP_ID,SALARY
5,1234
3,2341
2,3412
4,4123
1,4321
```

Print the name of the highest-paid employee:

```
$ ./highest_paid.sh employees.csv salaries.csv
Jack Sparrow
```

Requirements:

- Names of CSV files are passed as arguments
- Script must not rely on the exact contents of the files given above (no hardcoded values)

Recommendations:

- Try to think about corner cases
- Try different approaches
- Check out various options in the manuals
  - UNIX: man <command>
  - Google: man <command>

Expansion

Parameter expansion

- Simple substitution (sh-compatible)

```
${parameter}
```

- If `parameter` is unset or null, use `default` value

```
${parameter:-default}
```

- If `parameter` is set to non-null value, use `default` value

```
${parameter:+default}
```

- If `parameter` is unset or null, set it to `default` value and use it

```
${parameter:=default}
```

- If `parameter` is unset or null, display error message

```
${parameter:?error}
```

- Length of the value stored in the variable

```
${#parameter}
```

- Substring of the value of `parameter` (starting at given `offset`)

```
${parameter:offset}        # from offset till the end
${parameter:offset:length} # from offset till offset+length
```

- Remove leading part (prefix) that matches `pattern`

```
${parameter#pattern}        # shortest match
${parameter##pattern}       # longest match
```

- Remove trailing part (suffix) that matches `pattern`

```
${parameter%pattern}        # shortest match
${parameter%%pattern}       # longest match
```

- Replace matching `pattern` with `value` (anywhere)

```
${parameter/pattern/value}   # only the first match
${parameter//pattern/value}  # all matches
```

Examples

```
TEST="Hello!"
```

Expression	Result	Result when unset
${TEST}	Hello!	
${TEST:-"Hi!"}	Hello!	Hi!
${TEST:+"Hi!"}	Hi!	
${TEST:="Hi!"}	Hello!	Hi!
${TEST:?"Unset!"}	Hello!	bash: YYY: Unset!
${#TEST}	6	0

```
XXX="/tmp/test/bash_test.tar.gz"
```

Expression	Result
${XXX:2:3}	tmp
${XXX#/*/}	test/bash_test.tar.gz
${XXX##/*/}	bash_test.tar.gz
${XXX%.*}	/tmp/test/bash_test.tar
${XXX%%.*}	/tmp/test/bash_test
${XXX/test/prod}	/tmp/prod/bash_test.zip
${XXX//test/prod}	/tmp/prod/bash_prod.tar.gz
${XXX/#*./test.}	test.gz
${XXX/%.*/.zip}	/tmp/prod/bash_prod.zip

Arrays

Arrays are one-dimensional, can be indexed or associative

- Declare a variable as an array
  - optional for indexed arrays

```
declare -a NUMERALS   # Indexed array
declare -A NUMBERS    # Associative array
```

- Create an array using constructor

```
NUMERALS=("zero" "one" "two")
NUMBERS=([zero]=0 [one]=1 [two]=2)
```

- Put a value into an array

```
NUMERALS[0]="three"
NUMBERS[zero]=3
```

- Get a value from an array

```
echo ${NUMERALS[0]}   # The ${} syntax is required here
echo ${NUMBERS[zero]}
```

- Append a value (or another array) to an existing array

```
NMERICALS+=(three four five)
NUMBERS+=([three]=3 [four]=4 [five]=5)
```

- Get list of values in the array

```
echo "Values: " ${NUMERICALS[*]} # The same rules for
echo "Values: " ${NUMERICALS[@]} # quoting as in $* and $@
```

- Get list of array indices (keys)

```
echo "Keys: " ${!NUMBERS[*]}
echo "Keys: " ${!NUMBERS[@]}
```

- Get the length of an array

```
echo "Length: " ${#NUMBERS[*]}  # These two are equivalent
echo "Length: " ${#NUMBERS[@]}
```

Command substitution

- Executes a command and puts the output in place of the call
- The traditional way (sh-compatible)

```
echo "The current directory is:" `pwd`
The current directory is: /home/pawel
```

- The Bash built-in syntax (not sh-compatible)

```
echo "The current directory is: $(pwd)"
The current directory is: /home/pawel
```

- Bash syntax allows embedding command substitutions

```
echo "The current directory is: $(ls -d $(pwd))"
```

Arithmetic expressions

- The traditional (sh-compatible) way

```
TEST=`expr 2 + 2`
echo ${TEST}
4
```

- The Bash built-in syntax (not sh-compatible)

```
TEST=$((2 + 2))
echo ${TEST}
4
```

- `VAR` can be used instead of `$VAR`

```
NUMBER1=1
NUMBER2=$((NUMBER1 + 2))
echo ${NUMBER2}
3
```

Other expansions

- Word splitting
  - Result of expansion splitted using IFS chars

```
TEST1="a:b:c:d"
IFS=: TEST2=${TEST1}
echo "Before splitting: ${TEST1}"
echo "After splitting: ${TEST2}"
```

```
Before splitting: a:b:c:d
After splitting: a b c d
```

- Pathname expansion
  - Every non-quoted word containing *, ? or [is treated as a pattern and expanded to the list of matching filenames
    - * - matches any string (including empty string)
    - ? - matches any single character
    - [⋯] - matches any of the enclosed characters

```
$ echo "CSV files:" *.csv
CSV files: employees.csv salaries.csv

$ echo "CSV files: *.csv"
CSV files: *.csv
```

- Brace expansion
  - non-quoted word containing a brace pattern: w1{v1,v2,v3}w2 is expanded to a list of words:

```
      w1v1w2 w1v2w2 w1v3w2
```

```
$ echo ~/bash/{employees,salaries}.csv
~/bash/employees.csv ~/bash/salaries.csv
```

- Ranges of integers are supported: {x..y[..z]}

```
$ echo {1..10}
1 2 3 4 5 6 7 8 9 10

$ echo {2..10..2}
2 4 6 8 10
```

Control statements

Conditional expressions

- The traditional (sh-compat.) way - `[ CONDITIONS ]`

```
TEST="test"
[ "${TEST}" == "test" ] && echo "OK"
OK
```

- The Bash-specific syntax - `[[ CONDITIONS ]]`

```
TEST="test"
[[ "${TEST}" == "test" ]] && echo "OK"
OK
```

- Traditional version is just a regular command (used to be symbolic link to `test`)
- Built-in Bash syntax provides some extensions (e.g. regular expression matching)

Syntax	Meaning
`[ ]`	false
`[ ( EXPR ) ]`	EXPR is true
`[ ! EXPR ]`	EXPR is false
`[ EXPR1 -a EXPR2 ]`	both EXPR1 and EXPR2 are true
`[ EXPR1 -o EXPR2 ]`	either EXPR1 or EXPR2 is true
`[ -n STRING ]`	the length of STRING is nonzero
`[ STRING ]`	equivalent to -n STRING
`[ -z STRING ]`	the length of STRING is zero
`[ STRING1 = STRING2 ]`	the strings are equal
`[ STRING1 != STRING2 ]`	the strings are not equal
`[ NUM1 -eq NUM2 ]`	NUM1 is equal to NUM2
`[ NUM1 -ge NUM2 ]`	NUM1 is greater than or equal to NUM2
`[ NUM1 -gt NUM2 ]`	NUM1 is greater than NUM2
`[ NUM1 -le NUM2 ]`	NUM1 is less than or equal to NUM2
`[ NUM1 -lt NUM2 ]`	NUM1 is less than NUM2
`[ NUM1 -ne NUM2 ]`	NUM1 is not equal to NUM2
`[ FILE1 -ef FILE2 ]`	FILE1 and FILE2 have the same device and inode numbers
`[ FILE1 -nt FILE2 ]`	FILE1 is newer (modification date) than FILE2

Syntax	Meaning
`[ FILE1 -ot FILE2 ]`	FILE1 is older than FILE2
`[ -d FILE ]`	FILE exists and is a directory
`[ -e FILE ]`	FILE exists
`[ -f FILE ]`	FILE exists and is a regular file

`if-then-else` statement

```
if CONDITION_1 ; then
    # instructions ...
elif CONDITION_2 ; then
    # other instructions ...
elif CONDITION_N ; then
    # other instructions ...
else
    # even more instructions ...
fi
```

- Any command can serve as a condition
  - exit code == 0 means true
  - exit code <> 0 means false
- The semicolon (';') between the condition and 'then' keyword is mandatory (unless `then' is in a new line)
  - marks the end of the condition part

`case` statement

```
case WORD in
    PATTERN_1 [| PATTERN_2])
    # Instructions ...
    ;;
    PATTERN_3)
    # Instructions ...
    ;;
    *)
    # Default instructions ...
    ;;
esac
```

- Patterns are the same as in pathname expansion
- Using `;&` instead of `;;` causes a jump to the next case
  - Similar to omitting `break' in Java, C, C++ etc.

for statement

- Variant 1 - “for-each”

```
for VAR in WORD_1 WORD_2 ... ; do COMMAND_1 ; ... done
```

```
# Example
for FILE in $(ls -1); do
    echo "Found: ${FILE}"
done
```

- Variant 2 - “C-like”

```
for ((EXPR_1 ; EXPR_2 ; EXPR_3)) ; do COMMAND_1 ; ... done
```

```
# Example
for ((NUMBER=99 ; NUMBER>0 ; NUMBER--)); do
    echo "${NUMBER} bottles of beer on the wall..."
done
```

while statement

- Variant 1 - “while”

```
while CONDITION ; do COMMAND_1 ; ... done
```

```
COUNTER=100
while [[ $COUNTER > 0 ]] ; do
    echo "Countdown: ${COUNTER}"
    COUNTER=$((COUNTER-1))
done
```

- Variant 2 - “until”

```
until CONDITION ; do COMMAND_1 ; ... done
```

```
until ls *.txt 2>/dev/null ; do
    echo "Waiting for a text file in the pwd..."
    sleep 1s
done
```

Exercise

Given a CSV file called `employees.csv':

```
EMP_ID,FULL_NAME,AGE,PHONE
1,Jack Sparrow,34,555-1111
2,Hektor Barbossa,21,555-2222
3,Joshamee Gibbs,52,555-3333
4,Will Turner,43,555-4444
5,Elizabeth Swann,18,555-5555
```

And a CSV file called `salaries.csv':

```
EMP_ID,SALARY
5,1234
3,2341
2,3412
4,4123
1,4321
```

Write a script that will print the sum of the salaries of all employees above a certain age (given as a command-line argument), e.g.:

```
$ ./sum.sh 30 employees.csv salaries.csv
10785
```

Blocks, Functions, Processes

Command blocks

- `{ list; }`
  - Commands are executed in the current shell environment
    - No new process is created
    - Outer variables **can** be changed

```
TEST=1; { TEST=2; }; echo $TEST
2
```

- `( list )`
  - Commands are executed in a subshell environment
    - A child process is created
    - Outer variables **cannot** be changed

```
TEST=1; ( TEST=2; ); echo $TEST
1
```

- Status code of the block is the status code of its last command

```
{ true; false; } && echo "True" || echo "False"
False
```

```
true && { false; true; } && echo "True" || echo "False"
True
```

- Command blocks can have their own I/O redirections

```
echo "I have a cat." | {
    read LINE
    echo ${LINE/cat/dog}
} > output.txt
```

Functions

Functions are similar to blocks except they

- have names and positional parameters
- are executed when called by name
- can return status code (via `return` keyword)
- can have permanent redirections

```
[function] name() { list; } [redirections]
```

- The `function` keyword is optional

```
error() {
    # $* below refers to params of a function
    echo "ERROR: $*" >&2
    exit 1
}
[ -n "$1" ] || error "Missing parameter 1."
```

Subprocesses

Commands can be started in another process

- in foreground - parent waits for the child

```
echo "Starting..."
( sleep 10s; echo "subprocess done." )
echo "Parent process done."
```

- in background - parent resumes immediately

```
echo "Starting..."
( sleep 10s; echo "subprocess done." ) &
echo "Parent process done."
```

- Subprocess is connected to its parent
  - If parent dies, children die too
- Subprocess uses a copy of parent's env.
  - Variable changes don't affect the parent

Process-related parameters

- Special parameters
  - `$$` - PID of the current process (shell)
  - `$!` - PID of last job sent to background
- Built-in variables
  - `$PPID` - PID of the parent process (shell)
  - `$BASHPID` - actual PID of the current process

Job control

Job represents a group of processes

- `jobs` - prints a list of jobs for the current shell
- `fg, bg` - sends a job to foreground/background
- `wait` - waits for background jobs to complete

```
(echo "Task 1"; sleep 10s; echo "Done.") &
(echo "Task 2"; sleep 10s; echo "Done.") &
wait
```

Subshell

- Some processes are created by Bash as subshells
  - `$(⋯)` - command substitutions,
  - `(⋯)` command blocks (only top-level ones)
- Subshell is a forked shell without reinitialization
  - All environment variables are copied from parent including `$$` and `$PPID` (!)
  - `$BASHPID` stores the actual PID of a subshell

Exercise

Write a script that will:

- Run 10 background tasks in parallel
  - Each task prints its PID 10 times (one per second)
  - Each task redirects its output to a separate log file
- Wait for all tasks to end
- (optional) Print PID and log file name for each task

Debugging

Enabling debug mode

- Globally (for the whole script)
  - inside the script

```
#!/bin/bash -x
NAME=$1
echo 'Hello ${NAME}!'
```

  - when running the script

```
$ bash -x script.sh ...
```

- Selectively (in the middle of the script)

```
#!/bin/bash
NAME=$1
set -x   # Printing enabled
echo "Hello ${NAME}!"
set +x   # Printing disabled
```

Debugging expansions

- Print the commands just before execution

```
$ bash -x script.sh Pawel
+ NAME=Pawel
+ echo 'Hello Pawel'
Hello Pawel!
```

- Print the commands as they are written

```
$ bash -v script.sh Pawel
NAME=$1
echo "Hello ${NAME}!"
Hello Pawel!
```

- Combined

```
$ bash -xv script.sh Pawel
```

```
NAME=$1
+ NAME=Pawel
echo "Hello ${NAME}!"
+ echo 'Hello Pawel!'
Hello Pawel!
```

Other methods

- Disable pathname expansion (globbing)

```
$ set -f
$ echo *
*
```

- Add log messages (to stderr)

```
[[ -z $DEBUG ]] && echo "Converting ${FILE} to PDF..." >&2
convert ${FILE} ${FILE/%.png/.pdf}
```

- Report intermediate failures (exit codes)

```
set -e    # exit immediately if uncought error found
convert ${FILE} ${FILE/%.png/.pdf} || echo "Failed!" >&2
```

- Monitor pipeline I/O

```
sort -nr $1 | tee after_sort.txt | cut -d, -f1 | ...
```

Recommendations

- Use `set -o errexit` (a.k.a. `set -e`)
  - Script will exit on first unchecked error
  - Use '||' to catch (and log) checked errors
- Use `set -o nounset` (a.k.a. `set -u`)
  - Script will fail when using undeclared variable
  - Won't fail if default value is provided
- Use `set -o pipefail` to catch a failure in pipeline
  - Exit code will come from the failing command
- Create a function for reporting failures
- Use ShellCheck to check the script for bugs (https://www.shellcheck.net)

Optimization

Why should I care?

- For one-time scripts, you probably shouldn't
- Frequently used tools
  - Application startup/shutdown scripts
  - Development tools (e.g. mvn, Bash completion for git)
- Tools that operate on many/large files

Measurement

```
$ time <command> <args...>
```

- Measures the time spent on execution of the command
  - **Real** - wall clock time - from start to finish
  - **User** - time spent in user-mode code
  - **Sys** - time spent on system calls

Built-ins vs externals

- Prefer built-ins over externals
  - Every external command is a new subprocess
  - `read` may be 30 times faster than `cut, sed, awk`

```
$ time for ((i=0;i<10000;i++)); do
>     awk -F, '{print $1}' <<< "a,b,c" > /dev/null
> done

real    0m28,179s
user    0m13,662s
sys     0m15,117s
```

```
$ time for ((i=0;i<10000;i++)); do
>     IFS=',' read -r FIRST <<< "a,b,c" > /dev/null
> done

real    0m0,748s
user    0m0,373s
sys     0m0,370s
```

Toolbox vs. Multi-Tool

- Simple operations
  - use more specialized tool
    - `cut` is faster than `sed` or `awk`
    - `ls` is faster than `find`
- Complex tasks
  - consider using more powerful tool

```
awk -F, -f - salaries.csv <<-"EOF"
  BEGIN { sum = 0 }
  NR > 1 { sum += $2 }
  END { print "Average salary:", sum / (NR-1) }
EOF
```

- Really complex cases
  - Bash may not be the best tool

Processes

- For code blocks, prefer `{}` over `()`
  - `()` creates a subprocess
- Prefer input redirections over pipelines
  - pipeline creates subprocess for each part

```
# Not recommended
cat data.csv | while read LINE; do
  # Do something...
done

# Recommended
while read LINE; do
  # Do something...
done < data.csv
```

Best practices

Readbility

- Treat your script like a regular program
  - Indent your code
  - Group the code into (reusable) functions
    - some even recommend a `main' function
  - Use meaningful names for variables and functions
  - Use long command options (X11 style, GNU)
    - `getopt` (external) tool can help with parsing
  - Don’t hesitate to put some comments
  - You can even write unit tests
  - DRY, KISS etc.

Style

- Always
  - use `${VAR}` syntax instead of `$VAR`
    - `${APP}_scripts` vs. `$APP_scripts`
  - double-quote viariables (`"${VAR}"`)
    - prevents shell injection
- When compatibility with `sh` is not required
  - use `$()` instead of backquotes
  - use `[[ ⋯ ]]` instead of `[ ⋯ ]` and `test`
  - Don’t use `function` keyword (deprecated)

Usability

- Include help/usage in your script
- Add long option variants
- Validate input parameters
- Provide readable error messages
- Write warnings and errors to `stderr`
- Use various exit codes

Portability

- Use `#!/usr/bin/env bash` instead of `#!/bin/bash`
- Reduce number of dependencies
  - Prefer built-ins over externals
- Pay attention to the version of Bash
- Pay attention to variants of dependencies
  - `awk` may be link to: `mawk`, `gawk`, `nawk`
  - Even core utils (ls, find, grep) may vary