



Desktop OS

Advanced scripting

**DE HOGESCHOOL
MET HET NETWERK**

Hogeschool PXL – Dep. PXL-IT – Elfde-Liniestraat 26 – B-3500 Hasselt
www.pxl.be - www.pxl.be/facebook



sourcing a script

- standaard wordt een script uitgevoerd in een subshell
 - gevolg is dat je variabelen niet meer bestaan als het script is gestopt

```
student@desktop: ~/bin
#!/bin/bash

# Auteur: Desktop OS
# Datum: 25 oktober 2014
# Versie: 1.0

echo Ik ben $USER op $HOSTNAME

USER=joske
HOSTNAME=pcvanjoske

echo Ik ben $USER op $HOSTNAME
```

```
student@desktop: ~/bin
student@desktop:~/bin$ ./voorbeeldSourcing1.sh
Ik ben student op desktop
Ik ben joske op pcvanjoske
student@desktop:~/bin$ echo $USER $HOSTNAME
student desktop
student@desktop:~/bin$
```

sourcing a script

- je kan een script forceren om uitgevoerd te worden in dezelfde shell

```
student@desktop: ~/bin
#!/bin/bash

# Auteur: Desktop OS
# Datum: 25 oktober 2014
# Versie: 1.0

echo Ik ben $USER op $HOSTNAME

USER=joske
HOSTNAME=pcvanjoske

echo Ik ben $USER op $HOSTNAME
```

```
student@desktop: ~/bin
student@desktop:~/bin$ source ./voorbeeldSourcing1.sh
Ik ben student op desktop
Ik ben joske op pcvanjoske
student@desktop:~/bin$ echo $USER $HOSTNAME
joske pcvanjoske
student@desktop:~/bin$
```

OF

```
student@desktop: ~/bin
student@desktop:~/bin$ ./voorbeeldSourcing1.sh
Ik ben student op desktop
Ik ben joske op pcvanjoske
student@desktop:~/bin$ echo $USER $HOSTNAME
joske pcvanjoske
student@desktop:~/bin$
```

sourcing a script

- op deze manier kan je ook een script in een script laten uitvoeren, én ervoor zorgen dat ze beide in dezelfde shell worden uitgevoerd.

toegevoegd.sh

```
student@desktop: ~/bin
#!/bin/bash

# Auteur: Desktop OS
# Datum: 25 oktober 2014
# Versie: 1.0

echo Het toegevoegde script
vTekst=TOEGEVOEGD
echo inhoud variabele vTekst: $vTekst
```

voorbeeldSourcing2.sh

```
student@desktop: ~/bin
#!/bin/bash

# Auteur: Desktop OS
# Datum: 25 oktober 2014
# Versie: 1.0

source ./toegevoegd.sh

echo Het hoofdsript
echo inhoud variabele vTekst: $vTekst
```

```
student@desktop: ~/bin
student@desktop:~/bin$ ./voorbeeldSourcing2.sh
Het toegevoegde script
inhoud variabele vTekst: TOEGEVOEGD
Het hoofdsript
inhoud variabele vTekst: TOEGEVOEGD
student@desktop:~/bin$
```



sourcing a script

- als je een script wil uitvoeren met sourcing, moet je geen execute rechten hebben.
 - net zoals bij het gebruik van `"bash oef1.sh"`

```
student@desktop: ~/bin
student@desktop:~/bin$ ls -l | grep oef1.sh
-rw-rw-r-- 1 student student 86 Okt 21 10:08 oef1.sh
student@desktop:~/bin$ ./oef1.sh
bash: ./oef1.sh: Permission denied
student@desktop:~/bin$ source oef1.sh
Dit is mijn eerste script
Vandaag: 10/25/14
User: student
student@desktop:~/bin$ . oef1.sh
Dit is mijn eerste script
Vandaag: 10/25/14
User: student
student@desktop:~/bin$
```



script parameters

- script parameters zijn de argumenten die aan een script of een commando worden meegegeven.
 - Bvb: `optelsom.sh 15 387 85 97` (parameters 15, 378, 85 en 97)
- Parameters worden opgeslagen in het werkgeheugen. De verwijzing naar de parameter gebeurt via `$1`, `$2`, `$3`, ... `$9`
- Maximaal zijn er 9 verwijzigingen mogelijk.
- `$0` => verwijzing naar de naam van het commando zelf



script parameters

```
student@desktop: ~/bin
#!/bin/bash

# Auteur: Desktop OS
# Datum: 25 oktober 2014
# Versie: 1.0

echo commando-naam: $0
echo parameter 1: $1
echo parameter 2: $2
echo parameter 3: $3
echo parameter 4: $4
echo parameter 5: $5
echo parameter 6: $6
echo parameter 7: $7
echo parameter 8: $8
echo parameter 9: $9
```

```
student@desktop: ~/bin
student@desktop:~/bin$ ./voorbeeldParameters1.sh a b c d e f g h i
commando-naam: ./voorbeeldParameters1.sh
parameter 1: a
parameter 2: b
parameter 3: c
parameter 4: d
parameter 5: e
parameter 6: f
parameter 7: g
parameter 8: h
parameter 9: i
student@desktop:~/bin$
```



script parameters

- `$#` Verwijst naar het aantal gegeven parameters.
- `$*` Geeft als resultaat één string waarin alle parameters voorkomen, gescheiden door een delimiter gedefinieerd in de systeemvariabele IFS.
- `$@` Geeft als output alle parameters waarbij elke parameter als individuele string kan worden gebruikt.
- `$?` laatste return code
- `$$` PID van het script



shift through parameters

- slechts 9 parameters ?
- geen melding als de parameter niet bestaat ?
 - \$10 wordt aanzien als \$1 met en 0 erachter
 - \$11 wordt aanzien als \$1 met en 1 erachter
 - ...

```
student@desktop: ~/bin
#!/bin/bash

# Auteur: Desktop OS
# Datum: 25 oktober 2014
# Versie: 1.0

echo commando-naam: $0
echo parameter 1: $1
echo parameter 2: $2
echo parameter 3: $3
echo "..."
echo parameter 9: $9
echo parameter 10: $10
echo parameter 11: $11
```

```
student@desktop: ~/bin
student@desktop:~/bin$ ./voorbeeldParameters3.sh a b c d e f g h i j k l
commando-naam: ./voorbeeldParameters3.sh
parameter 1: a
parameter 2: b
parameter 3: c
...
parameter 9: i
parameter 10: a0
parameter 11: a1
student@desktop:~/bin$
```

shift through parameters

- shift : de verwijzigingen worden geshift!
\$2 -> \$1, \$3 -> \$2, ...
- de waarde van \$1 gaat bij iedere shift verloren
 - want \$1 krijgt de waarde van \$2
 - \$2 krijgt de waarde van \$3
 - \$3 krijgt de waarde van \$4
 - ...
 - \$# (=aantal parameters) wordt ook telkens 1 minder
 - de waarde van \$0 (=naam van het commando) blijft behouden



shift through parameters

```
student@desktop: ~/bin
#!/bin/bash

# Auteur: Desktop OS
# Datum: 25 oktober 2014
# Versie: 1.0

echo commando-naam: $0
echo parameter 1: $1
echo parameter 2: $2
echo parameter 3: $3
echo "..."
echo parameter 8: $8
echo parameter 9: $9
echo shift 2x
shift
shift
echo commando-naam: $0
echo parameter 1: $1
echo parameter 2: $2
echo parameter 3: $3
echo "..."
echo parameter 8: $8
echo parameter 9: $9
```

```
student@desktop: ~/bin
student@desktop:~/bin$ ./voorbeeldShift.sh a b c d e f g h i j k l
commando-naam: ./voorbeeldShift.sh
parameter 1: a
parameter 2: b
parameter 3: c
...
parameter 8: h
parameter 9: i
shift 2x
commando-naam: ./voorbeeldShift.sh
parameter 1: c
parameter 2: d
parameter 3: e
...
parameter 8: j
parameter 9: k
student@desktop:~/bin$
```

shift through parameters

```
student@desktop: ~/bin
#!/bin/bash

# Auteur: Desktop OS
# Datum: 25 oktober 2014
# Versie: 1.0

echo commando-naam: $0

vTeller=1
while [ $# -gt 0 ]
do
    echo parameter $vTeller: $1
    let vTeller++
    shift
done
```

```
student@desktop: ~/bin
student@desktop:~/bin$ ./voorbeeldShift2.sh a b c d e f g h i j k l
commando-naam: ./voorbeeldShift2.sh
parameter 1: a
parameter 2: b
parameter 3: c
parameter 4: d
parameter 5: e
parameter 6: f
parameter 7: g
parameter 8: h
parameter 9: i
parameter 10: j
parameter 11: k
parameter 12: l
student@desktop:~/bin$
```

while [\$# -gt 0] = zolang als er nog parameters zijn (aftellend naar 0...)



Parameters controleren met regex

```
vwg@laptop: ~/PXLdemoFiles/ScriptingAdvanced
#!/bin/bash

if [ $# -eq 0 ]
then
    echo "U gaf geen parameter op. Probeer opnieuw met één parameter..."
elif [[ $1 =~ ^[a-zA-Z]+$ ]]
then
    echo "U gaf een string bestaande uit letters"
elif [[ $1 =~ ^[0-9]+$ ]]
then
    echo "U gaf een getal bestaande uit cijfers"
else
    echo "U gaf een mix van letters, cijfers of andere karakters"
fi
```

1,1 All

`=~` duidt op een regular expression
`[[...]]` nodig als er met pattern matching(==) of met regex (`=~`) wordt gewerkt

`^[a-zA-Z]+$` `^`: moet beginnen met
`[a-zA-Z]`: kleine of hoofdletter
het vorige 1 of meer keer
`$`: moet eindigen met

Dus: het moet beginnen met een letter, het mogen ook meerdere letters zijn en het moet ook eindigen met een letter

```
vwg@laptop: ~
vwg@laptop:~$ paramscontroleren01.sh
U gaf geen parameter op. Probeer opnieuw met één parameter...
vwg@laptop:~$ paramscontroleren01.sh 123
U gaf een getal bestaande uit cijfers
vwg@laptop:~$ paramscontroleren01.sh abc
U gaf een string bestaande uit letters
vwg@laptop:~$ paramscontroleren01.sh abc123def
U gaf een mix van letters, cijfers of andere karakters
vwg@laptop:~$ paramscontroleren01.sh ù#
U gaf een mix van letters, cijfers of andere karakters
vwg@laptop:~$
```

shell functions

- Wat?
 - groep van commando's
 - wordt aangeroepen door de functienaam
- Waarom?
 - Centraal onderhoud
 - Snelheid
 - functie wordt geprocessed vanuit werkgeheugen en niet vanuit file



shell functions

- een functie kan op 2 manieren gedefinieerd worden:

<pre>function functienaam { command1 command2 command... }</pre>	of	<pre>functienaam () { command1 command2 command... }</pre>
--	----	--

- functies moeten bovenaan staan!



shell functions

- Begin elk script met commentaar (wat doet het script?)
- Plaats ook commentaar doorheen het script om bepaalde stukjes code te verduidelijken (later wil je ook nog gemakkelijk achterhalen waarom iets in het script is opgenomen)
- Gebruik functies om herhaling van stukken code te vermijden



shell functions

- voorbeeld

```
student@desktop: ~/bin
#!/bin/bash

# Auteur: Desktop OS
# Datum: 26 oktober 2014
# Versie: 1.0

function sayHello
{
    echo Hello !
}

echo main script
sayHello
echo end main
```

```
student@desktop: ~/bin
student@desktop:~/bin$ ./voorbeeldFuncties1.sh
main script
Hello !
end main
student@desktop:~/bin$
```



shell functions

- functies werken ook met parameters

```
student@desktop: ~/bin
#!/bin/bash

# Auteur: Desktop OS
# Datum: 26 oktober 2014
# Versie: 1.0

function sayHello
{
    echo Hello $1 !
}

echo main script
sayHello $USER
echo end main
```

```
student@desktop: ~/bin
student@desktop:~/bin$ ./voorbeeldFuncties2.sh
main script
Hello student !
end main
student@desktop:~/bin$
```



shell functions

- de parameters van een functie komen niet overeen met de script parameters

student@desktop: ~/bin

#!/bin/bash

Auteur: Desktop OS

Datum: 26 oktober 2014

Versie: 1.0

function slowPrint

{

echo \$1

sleep \$timetowait

echo \$2

sleep \$timetowait

echo \$3

sleep \$timetowait

}

timetowait=\$1

slowPrint \$4 \$3 \$2

student@desktop: ~/bin

student@desktop:~/bin\$./voorbeeldFuncties3.sh 2 een twee drie

drie

twee

een

student@desktop:~/bin\$



shell functions

- voorbeeld functie met parameters en gebruik van shift

```
student@desktop: ~/bin
~/bin/bash

# Auteur: Desktop OS
# Datum: 26 oktober 2014
# Versie: 1.0

function slowPrint
{
    while [ $# -gt 0 ]
    do
        echo $1
        sleep $timetowait
        shift
    done
}

timetowait=$1
shift
slowPrint $@
```

```
student@desktop: ~/bin
student@desktop:~/bin$ ./voorbeeldFuncties4.sh 1 een twee drie
een
twee
drie
student@desktop:~/bin$
```

slowPrint \$@, slowPrint \$* en slowPrint "\$@" geven
een
twee
drie

slowPrint "\$*" geeft
een twee drie

shell functions

scope variabelen

```
student@desktop: ~/bin
#!/bin/bash

# Auteur: Desktop OS
# Datum: 26 oktober 2014
# Versie: 1.0

function nieuwevars
{
    x=2
    local y=6
    echo "In de functie is x de waarde $x en y de waarde $y  gegeven"
}

x=1
y=5
echo "bij starten van het script heeft x de waarde $x en y de waarde $y"
nieuwevars
echo "na afloop van de functie heeft x de waarde $x en y de waarde $y"
```

```
student@desktop: ~/bin
student@desktop:~/bin$ ./voorbeeldFuncties5.sh
bij starten van het script heeft x de waarde 1 en y de waarde 5
In de functie is x de waarde 2 en y de waarde 6  gegeven
na afloop van de functie heeft x de waarde 2 en y de waarde 5
student@desktop:~/bin$
```



exit

- exit-status
 - Elk commando geeft een return code (exit-status) terug aan zijn host-process.
 - Waarde tussen [0..255], 0 = Goed,
 - [1..255] is fout-code.
 - Exit codes with a special meaning
<http://tldp.org/LDP/abs/html/exitcodes.html>
 - Exit status kan bewaard worden door de shell variable ?

```
student@desktop:~$ cd bin
student@desktop:~/bin$ echo $?
0
student@desktop:~/bin$ cd onbestaandedir
bash: cd: onbestaandedir: No such file or directory
student@desktop:~/bin$ echo $?
1
```



exit

- exit
 - beëindigen van een shellscript, exit-status als argument
- return
 - beëindigen van een functie, exit-status als argument



exit

- voorbeeld met exit en return

```
student@desktop: ~/bin
#!/bin/bash

# Auteur: Desktop OS
# Datum: 26 oktober 2014
# Versie: 1.0

# functie gebruik van het script
function usage {
    echo "Usage: $0 filename"
    exit 1
}

# functie controleert of file bestaat
function does_file_exist {
    if [ -f $1 ]
    then
        return 0
    else
        return 1
    fi
}

# main, testen van functies
if [ $# -ne 1 ]
then
    usage
fi
```

```
does_file_exist $1
if [ $? -eq 0 ]
then
    echo File found
else
    echo File not found
fi
```

```
student@desktop: ~/bin
student@desktop:~/bin$ ./voorbeeldExit.sh
Usage: ./voorbeeldExit.sh filename
student@desktop:~/bin$ echo $?
1
student@desktop:~/bin$ ./voorbeeldExit.sh test oef1.sh
Usage: ./voorbeeldExit.sh filename
student@desktop:~/bin$ ./voorbeeldExit.sh test
File not found
student@desktop:~/bin$ echo $?
0
student@desktop:~/bin$ ./voorbeeldExit.sh oef1.sh
File found
student@desktop:~/bin$
```

case

- Strings vergelijken met een aantal gegeven strings
- *vaak gebruikt voor opties bij een script -> zie verder*
- syntax:

```
case string in
    str1)
        cmd-reeks1;;
    str2 | str3)
        cmd-reeks2;;
    *)
        cmd-reeks3;;
esac
```



Pipe-teken(|) als OR-functie
Sterretje (*) bedoelt: in alle andere gevallen

case

```
student@desktop: ~/bin
!/bin/bash

# Auteur: Desktop OS
# Datum: 26 oktober 2014
# Versie: 1.0

while [ -n "$(echo $1 | grep '-')" ] # zolang het argument een optie is
do
    case $1 in # welke optie
        -a) echo "optie a";;
        -b) echo "optie b";;
        *) echo "onbekende optie: $1";;
    esac
    shift
done
```

of while [[\$1 =~ ^-\.+]]

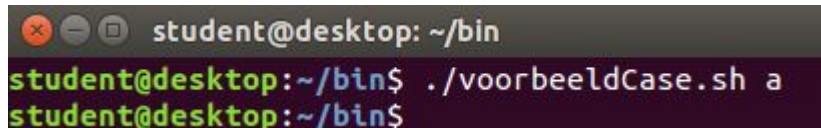


```
student@desktop: ~/bin
student@desktop:~/bin$ ./voorbeeldCase.sh -b -a -c
optie b
optie a
onbekende optie: -c
student@desktop:~/bin$
```

case

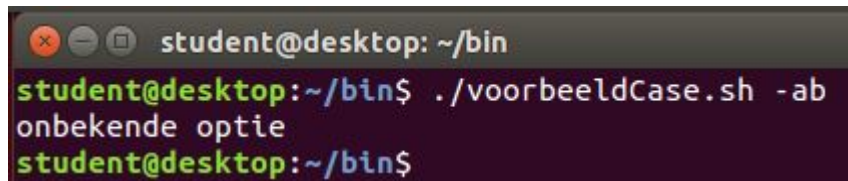
MAAR in dit voorbeeld:

- opties moeten met koppelteken beginnen



```
student@desktop: ~/bin
student@desktop:~/bin$ ./voorbeeldCase.sh a
student@desktop:~/bin$
```

- een optie “-ab” zal niet lukken



```
student@desktop: ~/bin
student@desktop:~/bin$ ./voorbeeldCase.sh -ab
onbekende optie
student@desktop:~/bin$
```

- geen opties mogelijk die zelf een argument hebben

=> oplossing getopt



get script options with getopt

- Opties zijn herkenbaar doordat ze beginnen met een koppelteken (-)
- Behandelen van opties in shell-scripts
 - zoeken naar opties met grep
 - zoeken naar opties met getopt (“get options”)



get script options with getopt

“:ab:c”

definieert alle opties die gebruikt kunnen worden
De lijst opties begint met een dubbele punt om aan te geven dat we zelf alle fouten afhandelen.
Elke optie die een argument kan hebben, wordt gevolgd door een dubbele punt

option

de naam van een variable die tijdens de werking van getopt wordt gebruikt.

getopt maakt gebruik van een eigen var **OPTARG** om het argument van een optie tijdelijk te bewaren.

de variable **OPTIND** bevat het volgnummer van de volgende optie of argument.

\$((...)) -> arithmetic expansion

je doet een wiskundige berekening en gebruikt het resultaat

```
student@desktop: ~/bin
#!/bin/bash

# Auteur: Desktop OS
# Datum: 26 oktober 2014
# Versie: 1.0

while getopt ":ab:c" option
do
    case $option in
        a) echo "optie a";;
        b) echo "optie b"
           echo "het argument van -b is $OPTARG";;
        c) echo "optie c";;
        *) echo "optie [-a] [-b arg ] [-c] args"
           exit 1;;
    esac
done
shift $((OPTIND-1))
echo parameter 1 is $1
```


get script options with getopt

```
student@desktop: ~/bin
student@desktop:~/bin$ ./voorbeeldGetopts1.sh -a -c test
optie a
optie c
parameter 1 is test
student@desktop:~/bin$ ./voorbeeldGetopts1.sh -b argb -a test
optie b
het argument van -b is argb
optie a
parameter 1 is test
student@desktop:~/bin$ ./voorbeeldGetopts1.sh -cb argb -a test
optie c
optie b
het argument van -b is argb
optie a
parameter 1 is test
student@desktop:~/bin$ ./voorbeeldGetopts1.sh test
parameter 1 is test
student@desktop:~/bin$
```



get script options with getopt

- voorbeeldGetopts2.sh



```
student@desktop: ~/bin
#!/bin/bash

# Auteur: Desktop OS
# Datum: 26 oktober 2014
# Versie: 1.0
# voorbeeldGetopts2.sh [-g][-b directory]
# -g toont gebruikers die ingelogd zijn
# -b toont bestanden van directory

function foutmelding
{
    echo "gebruik: $0 [-g][-b directory]"
    exit 1
}

while getopt ":gb:" option
do
    case $option in
        g) users=$(who)
           echo $users ;;
        b) bestanden=$(ls $OPTARG)
           echo $bestanden | tr ' ' '\n' ;;
        *) foutmelding ;;
    esac
done
```

get script options with getopt

- voorbeeldGetopts2.sh output

```
student@desktop: ~/bin
student@desktop:~/bin$ ./voorbeeldGetopts2.sh
student@desktop:~/bin$ ./voorbeeldGetopts2.sh -a
gebruik: ./voorbeeldGetopts2.sh [-g][-b directory]
student@desktop:~/bin$ ./voorbeeldGetopts2.sh -g
student :0 2014-10-26 11:19 (:0) student pts/12 2014-10-26 21:22 (:0) student pts/7 2014-10-26 20:15 (:0)
student@desktop:~/bin$ ls ../dir1
dir2 file1 file2
student@desktop:~/bin$ ./voorbeeldGetopts2.sh -b ../dir1
dir2
file1
file2
student@desktop:~/bin$ ./voorbeeldGetopts2.sh -b / | head -5
bin
boot
cdrom
dev
etc
student@desktop:~/bin$
```

eval

- de argumenten van het commando eval worden samengevoegd tot 1 string
- deze string wordt geëvalueerd, dit betekent dat dit wordt uitgevoerd op dezelfde manier als dit in je commandline ingegeven zou zijn (*→ shell expansion wordt dus toegepast op het argument*)
- je moet hier voorzichtig mee omgaan, dit is een krachtig commando. Indien je een andere oplossing kent, heeft deze de voorkeur.



(Eval command and security issues: <http://mywiki.woolledge.org/BashFAQ/048>)

eval

- voorbeelden:

```
student@desktop: ~/bin
student@desktop:~/bin$ x=test
student@desktop:~/bin$ echo $x
test
student@desktop:~/bin$ test=tada
student@desktop:~/bin$ echo $test
tada
student@desktop:~/bin$ y='$'$x
student@desktop:~/bin$ echo $y
$test
student@desktop:~/bin$ eval echo $y
tada
student@desktop:~/bin$
```

```
student@desktop: ~/bin
student@desktop:~/bin$ x="ls / | head -5"
student@desktop:~/bin$ $x
ls: invalid option -- '5'
Try 'ls --help' for more information.
student@desktop:~/bin$ eval $x
bin
boot
cdrom
dev
etc
student@desktop:~/bin$
```



(())

- wiskundige tests

```
student@desktop: ~/bin
student@desktop:~/bin$ x=5
student@desktop:~/bin$ (( $x < 10 )) && echo true || echo false
true
student@desktop:~/bin$ (( $x <= 10 )) && echo true || echo false
true
student@desktop:~/bin$ (( $x == 5 )) && echo true || echo false
true
student@desktop:~/bin$ ! (( $x == 5 )) && echo true || echo false
false
student@desktop:~/bin$ (( $x < 10 && $x > 0 )) && echo true || echo false
true
student@desktop:~/bin$ _
```



- *let ook weer op de spaties: ((_ <test> _))*


```
student@desktop: ~/bin
#!/bin/bash

# Auteur: Desktop OS
# Datum: 25 oktober 2014
# Versie: 1.0

i=1
while (( $# ))
do
    if [[ ! $1 =~ ^[0-9\-][0-9]*$ ]]
    then
        echo "parameter $i: $1 is geen getal"
    elif (( $1 < 10 ))
    then
        echo "parameter $i: $1 is kleiner dan 10"
    else
        echo "parameter $i: $1 is groter of gelijk aan 10"
    fi
    let i++
    shift
done
```

(())

- voorbeeld

```
student@desktop: ~/bin
student@desktop:~/bin$ ./voorbeeldArithmetic1.sh 0 8 14 a -3 --
parameter 1: 0 is kleiner dan 10
parameter 2: 8 is kleiner dan 10
parameter 3: 14 is groter of gelijk aan 10
parameter 4: a is geen getal
parameter 5: -3 is kleiner dan 10
parameter 6: -- is geen getal
student@desktop:~/bin$
```



let

- built-in shell functie
- wiskundige berekeningen

```
student@desktop: ~/bin
student@desktop:~/bin$ let x="3+4"
student@desktop:~/bin$ echo $x
7
student@desktop:~/bin$
```

of

```
student@desktop: ~/bin
student@desktop:~/bin$ x=$(( 3+4 ))
student@desktop:~/bin$ echo $x
7
student@desktop:~/bin$
```

- of werken met een teller

```
vteller=0
let vteller++
echo $vteller
1
```

of let vteller+=2



let

```
student@desktop: ~/bin
student@desktop:~/bin$ let y="$x + 5"
student@desktop:~/bin$ echo $y
12
student@desktop:~/bin$ let z="$x+$y"
student@desktop:~/bin$ echo $?
0
student@desktop:~/bin$ echo $z
19
student@desktop:~/bin$ let z="3-3"
student@desktop:~/bin$ echo $?
1
student@desktop:~/bin$ echo $z
0
student@desktop:~/bin$
```

`$?` is 0, behalve als de uitkomst 0 is van de wiskundige expressie, in dat geval is `$?` 1

