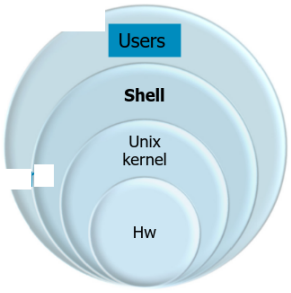
Shells

The outmost layer of the OS. It provides tje user interface, which interprets the user commands. It was the olny interface before the introduction of graphical servers.



Shell command expansion

Some characters have special meaning within the shell. Bash provide complex substitution mechanisms:

- After dividing the command line into tokens, the shell expands or solves these tokens, i.e., it applies different types of replacement
  - Braces, tilde, variables and parameters, commands, arithmetic expressions, etc.
- The substitution is complex and takes place with a specific order

Parentheses

(), [], {}

- Enclose variables, arithmetic operations, etc.
- In some cases, they are subject to automatic expansion (brace expansion)
  Example:

```
> name=Jean
> echo $namePaul

> echo {$name}Paul
{Jean}Paul
> echo ${name}Paul
JeanPaul
```

Quoting

"Quoting" means the use of for quotation marks

- Quotes ' '
  - Variables within quotes are not expanded
  - They cannot be nested
- Double quotes " "
  - Variables within double quotes are expanded
  - They can be nested
- Backslash \
  - Identifies the escape character, which remove the special meaning of the character that follows it

Example:

```
> myVar="A string"
> echo $myVar
A string

> echo 'v = $myVar'
```

```
v = $myVar

> echo "v = $myVar"
v = A string

> echo \$myVar
$myVar

> echo "double quote\""
double quote”
```

Using the output of command

The standard output of a command can be captured by

- Enclosing the command in `$(...)`
- Enclosing the command in backquotes ``
  In particular, the output of a command can be stored in a variable

```
> d=$(date)
> echo $d
> Fri Nov 22 10:00:0 \
CET 2013
> d=`date`

> out=`cat file.txt`
> echo $out
... file content ...

> out=`< file.txt`
> echo $out
... file content ...
```

Command execution

In a shell, a command can be executed

- Directly
  - `cd /home ; ls`
  - The current shell executes the command; change directory to **/home**; executes `ls` ; at the end the working directory is **/home**
- Indirectly
  - `(cd /home; ls)`
  - The current shell executes the command in a subprocess; change directory to **/home**; executes `ls` ; at the end the working directory is **the original directory**

History

A shell

- Keeps the list of the last submitted commands
  - In bash, the list is stored in file .bash_history
    - Stored in the user home directory
- Shell commands allow to reference this list

Command	Meaning
`history`	Displays the list of the last submitted commands
`!n`	Executes command number `n` in the history list
`!str`	Executes last command beginning by `str`

Command	Meaning
`^str1^str2`	Executes last command replacing `str1` by `str2`

Aliasing

In shell you can define new names to existing commands. The `alias` command allows defining these names

```
> alias lista= "ls -a"
> lista
...

>unalinas lista
>lista
command not found: lista
```

Execution

Scripts Are normally stored in files with .sh extension (or .bash). But recall that the extensions are not used UNIX/Linux to determine the file type.
They can be executed using two techniques

- Direct execution
- Indirect execution

Direct execution

```
./scriptname args
```

The script is executed from the command line as a normal executable file

- The script file must have the execute permission
  - `chmod +x ./scriptname`
- The first line of the script can specify the name of the script interpreter
  - `#!/bin/bash` or `#!/bin/sh`
- It is possible to execute the script using a specific shell
  - `/bin/bash ./scriptname args`

Indirect execution

```
source ./scriptname args
```

The source command executes the script given as its argument

- It is the current shell to run the script
  - "The current shell sources the script"
- It is not necessary that the script is executable
- The changes made by the script to environment variables remain in effect in the current shell

The script is executed by a sub-shell

- i.e., by a new shell process
- Environment (variables) of the original process and of the new one are not the same
- Changes to the environment variables made by the script, and used within the script, are lost at exit

Script debugging

Ther are not specific tools to debug bash scripts. You can ofcourse use "echo".

However, it is possible to "debug" a script in the following way

- Full (the whole script)
  - It is obtained by indicating a "debug" option at the level of the entire script
- Partial (only a few lines of the script)
  - It is obtained by indicating a "debug" option at the level of some lines of the script using the `set` command

Possible options for both partial and full debug:

- `-o noexec`, `-n`
  - Executes a syntactic check, but the script is not executed
- `-o verbose`, `-v`
  - Displays the executed commands
- `-o xtrace`, `-x`
  - Displays the execution trace of the entire script
- `-o nounset`, `-u`
  - Prints a error for undefined variables

That is:
Fully debug

- From a shell command
  - `/bin/bash -n ./scriptname args`
- Inside the script
  - `#!/bin/bash -v`
    - `#!/bin/bash -x`
  - ...

Partial debug

- `set -o verbose ... set +o verbose`
- `set -v ... set +v`
- `set -x ... set +x`

Syntax general rules

- The bash language is relatively "high level", and it allows to mix
  - Standard shell commands
    - ls, wc, find, grep, ...
  - Standard constructs of the shell language
    - Input and output variables and parameters, operators (arithmetic, logic, etc.), control constructs (conditional, iterative), arrays, functions, etc.
- Often instructions/commands are written in separate lines
  - on the same line, they must be separated by `;`
- Comments
  - Character `#` indicates the presence of a comment on the line
  - A comment begins by character # and terminates at the end of line
- exit allows terminating a script returning an error code

```
exit
```

`exit [0|1|2|3|…]`

- In shell, 0 means TRUE

```
#!/bin/bash
# This line is a comment
rm –rf ./../newDir/
mkdir ./../newDir/
cp * ../newDir/
```

```
ls ../newDir/ ; #; is superflu
# 0 is TRUE in shell programming
exit 0
```

Arguments

The arguments of the command line passed to the script are identified by `$`

- Positional parameters
  - `$0` is the script name
  - `$1`, `$2`, `$3`, ... indicate the arguments passed to the script on the command line. The `shift` command shifts the parameters to the left but $0 remains unchanged.
- Special parameters
  - `$*` Is the entire list (string) of arguments (excluding the script name)
  - `$#` Is the number of parameters (excluding th`e script name)
  - `$$` Is the process PID

```
#!/bin/bash
# Using command line parameters
echo "Running process is $0" # The double quotes expands the variables
echo "Parameters: $1 $2 $3 etc." #
echo "Number of parameters $#"
echo "List of parameters $*"
shift
echo "Parameters: $1 $2 $3 etc."
shift
echo "Parameters: $1 $2 $3 etc."
exit 0
```

```
> ./arguments.sh Aa Bb Cc
Running process is ./arguments.sh
Parameters: Aa Bb Cc etc.
Number of parameters 3
List of parameters Aa Bb Cc
Parameters: Bb Cc  etc.
Parameters: Cc   etc.
```

Variables

Variables can be

- Local (shell variables)
  - Available only in the current shell
- Global (environment variables)
  - Available in all sub-shells
  - Are exported by the current shell to all the process executed by the shell
- Main features of shell variables
  - Are not declared
    - A variable is created by assigning a value to the variable name
  - Are case sensitive
    - `Var`, `VAR`, and `var` are different variables
  - Some names are reserved for special purposes
- The list of all defined variables and associated value is displayed by command `set`
- The `unset` command clears the value of a variable

```
unset name
```

Local (shell) variables

Characterized by a name and associated content

- The content specifies the type
  - Constant, string, integer, vector or matrix
- The contents associated to a name are strings (even if a string can be interpreted as a numeric value)
- Setting
  - `name="value"` . No blanks around '='. Double quotes are mandatory if the string includes blank characters
- Usage
  - `$name`

```
> var=Hello
> echo $var
Hello
> var=7+5 #Variables are strings !!
> echo $var
7+5
> i="Hello world!"
> echo $i
Hello world!
> i=$i" Bye!!!"
> echo $i
Hello world! Bye!!!
> i=Hello world
world: command not found
```

Global (environment) variables

The export command allows creating an environment variable visible by other processes.

```
export name
```

Notice that

- Some environment variable names are predefined and reserved
- When a shell is executed these variables are automatically initialized starting from "environment" values
- These variable names are typically uppercase
- Can be displayed by means of the printenv (or env) command

Local vs Global:

```
> v=one
> echo $v
one
> bash
> ps -l
#… Two bashes running
> echo $v

> exit #now we are in the bash where v was defined
> echo $v
one
```

```
> v=one
> echo $v
one
> export v
> bash
> ps –l
… Two bashes running
> echo $v
one
```

```
> exit
> echo $v
one
```

Predefined variables

Variable	Meaning
`$?`	Stores the return value of the last process: 0 on success, other than 0 (between 1 and 255) on error. Value 0 corresponds to the TRUE value (unlike in C language)
`$SHELL`	Current shell
`$LOGNAME`	Username used for login
`$HOME`	User home directory
`$PATH`	List of the directories, delimited by `:` used for searching the executable files and commands
`$PS1`	Main prompt (usually `$` for users, `#` for root)
`$PS2`	Auxiliary prompt (usually `>`)
`$IFS`	Lists the characters that delimits the "words" in an input string (see read shell command)

Read from stdin

The `read` function allows reading a line from standard input

```
read [options] var1 var2 ... varn
```

- read can be possibly followed by a list of variables
- The "words" of the read line will be assigned in turn to each variable
- Possible excess words are all stored (as a string) in the last variable
- If no variables are specified, the complete input string is stored in variable `REPLY`

Supported options

- `-n nchars`
  - Returns after reading nchars characters without waiting for newline
- `-t timeout`
  - Timeout on reading
  - Returns 1 if a string is not typed within timeout seconds
- etc

```
> read v1 v2
input line string
> echo $v1
input
> echo $v2
line string
> read
> One two three
> echo $REPLY
One two three
> read
One two three
> v=$REPLY
> echo $v
One two three
```

Write to stdout

Output on stdout can be performed using

```
echo
printf
```

- `printf` syntax is similar to C language printf
  - Uses escape characters
  - It is not necessary to delimit fields by `" "`
- `echo` displays its arguments, delimited by blank, and terminated by newline
  - Options
    - -n eliminates the newline
    - -e interprets escaped (...) characters
      - `\b` backspace
      - `\n` newline
      - `\t` tab
      - `\\` backslash
      - etc.

```
echo "Printing with a newline"
echo –n "Printing without newline"
echo –e "Deal with \n escape \t\t characters"
printf "Printing without newline"
printf "%s \t%s\n" "Hello. It's me:" "$HOME"
```

Arithmetic expressions

Several notations can be used for defining arithmetic expressions

- Command `let "..."`
- Double parentheses `((…))`
- Square parentheses `[…]`
- Syntactic statement `expr`. Archaich, not to be used
  - Evaluates an expression by means of a new shell
  - Less efficient
  - Normally not used

```
> i=1
> ((v1=i+1))
> ((v2=$i+1))
> v3=$(($i+1))
> v4=$((i+1))
> echo $i $v1 $v2 $v3 $v4
1 2 2 2 2

> i=1
> let v1=i+1
> let "v2 = i + 1"
> let v3=$i+1
> echo $i $v1 $v2 $v3
1 2 2 2

> i=1
> v1=$[$i+1]
> v2=$[i+1]
> echo $i $v1 $v2
1 2 2
```

if-then-fi

The conditional statement `if-then-fi` checks if the exit status of a sequence of commands is equal to 0 standard format. If so it executes one or more commands.

```
if condExpr
then
     statement
fi
```

short

```
if condExpr ; then statement
fi
```

else

```
if condExpr
then
     statement
else
     stetament
fi
```

elif

```
if condExpr
then
     statement
elif condExpr
then statement
else
     stetament
fi
```

conditional statement

It can be written in two ways

```
test parameter operator paramenter
#test one ==2
```

```
[ param operator parameter ] #INCLUDE SPACES ON SIDES
#[ one == 2 ]
```

operator for numbers:

Operator	Description	Alternative (e.g., in (( )))
`-eq`	Equal to	`==`
`-ne`	Not equal to	`!=`
`-gt`	Greater than	`>`
`-ge`	Greater than or equal to	`>=`
`-lt`	Less than	`<`
`-le`	Less than or equal to	`<=`
`!`	Logical NOT (Negation)	`!`

operator for strings:

Operator	Description	Alternative (e.g., in)
`=`	Strings are equal (Same as `strcmp` returning 0)	`==`

Operator	Description	Alternative (e.g., in)
`!=`	Strings are not equal (Same as `!strcmp` returning 0)	`!=`
`-n string`	True if the length of *string* is non-zero (non-NULL)	N/A
`-z string`	True if the length of *string* is zero (NULL or empty)	N/A

operator for files and dir:

Operator	Description
`-d`	Argument is a **directory**
`-f`	Argument is a **regular file** (not a directory or special file)
`-e`	Argument **exists** (file or directory)
`-r`	Argument has **read** permission
`-w`	Argument has **write** permission
`-x`	Argument has **execution** permission
`-s`	Argument has **non-null dimension** (size is greater than zero)

logical operators:

Operator	Context	Description
`!`	Inside `[ ]`	Logical **NOT** (Negates the condition)
`-a`	Inside `[ ]`	Logical **AND** (Both conditions must be true)
`-o`	Inside `[ ]`	Logical **OR** (At least one condition must be true)
`&&`	Command sequence	**AND** (Execute the second command only if the first command succeeds/returns exit code 0)
||	Command sequence	

examples:

```
if [ 0 ] # true, interpreted as a string
if [ 1 ] # true
if [ -1 ] # true
if [ ] # NULL is false
if [ str ] # a random string is true, e.g., "abc" or abc is true

if [ $v1 –eq $v2 ]
then
     echo "v1==v2"
fi

if [ $v1 -lt 10 ]
then
     echo "$v1 < 10"
else
     echo "$v1 >= 10"
fi

if [ "$a" -eq 24 -a "$s" = "str" ]
then
     echo "a is 24 and s= `str`"
fi
```

for-in

```
for var in list
```

Executes the commands for each value taken by variable `var`
The list of values can be given explicitly (list) or implicity (result of shell commands)

```
for var in list
do
     statements
done
```

examples:

```
for foo in 1 2 3 4 5 6 7 8 9 10
do
     echo -n $foo
done


#1 2 3 4 5 6 7 8 9 10
```

```
for str in foo bar echo charlie tango
do
     echo -n $str
done


#foo bar echo charlie tango
```

```
num="2 4 6 9 2.3 5.9"
for file in $num
do
     echo -n $file
done

#2 4 6 9 2.3 5.9
```

```
a=$(ls *.txt)

for i in $a
do
     echo "I: $i" #writes all files that are .txt in the directory
done
```

```
rm –f number.txt
for i in $(echo {1..50})
do
     echo –n "$i " >> number.txt
done
```

while-do-done

iterates while the condition is true.

```
while [ cond ]
do
     statements
done
```

example

```
limit=10
var=0
while [ "$var" -lt "$limit" ]
do
     echo "Here var is equal to $var"
     let var=var+1
done
exit 0
```

```
#!/bin/bash
echo "Enter password: "
read myPass
while [ "$myPass" != "secret" ]; do
     echo "Sorry. Try again."
     read myPass
done

exit 0
```

Break, continue and ':'

`break` and `continue` statements have the same meaning in shell and in C language
`break` unstructured exit from the cycle
`continue` skip to the next iteration of the cycle
character `:` can be used:

for creating null instructions

```
if [ -d "$file" ]; then
     : # Empty instruction
fi
```

For indicating a True condition

```
while : #equivalent to while [ 0 ]
```

Arrays

Any variable can be defined as an array, explicit declatarion is not required. No restriction on the dimension of the array and on the use of contiguous indices. Indeces start at 0.
definition
Element-wise:

```
name[index]="value"
```

By means of a list of values:

```
name = (list of values separated by blanks)
```

reference
The use of { } is mandatory

A single element

```
${name[index]}
```

All elements

```
${name[*]} # * or @
```

Number of elements:

```
${#name[*]}
```

lenght of i-th element

```
${#name[i]}
```

Statement unset eliminates:
an element:

```
unset name[index]
```

an array:

```
unset name
```

examples:

```
> vet=(1 2 5 hello)
> echo ${vet[0]}
1
> echo ${vet[*]}
1 2 5 hello
> echo ${vet[1-2]}
2 5
> vet[4]=bye
> echo ${vet[*]}
1 2 5 hello bye
> unset vet[0]
> echo ${vet[*]}
2 5 hello bye
> unset vet
> echo ${vet[*]}
> vet[5]=100
> vet[10]=50
> echo ${var[*]} #not contihous indexes
100 50
```