# Command substitution

$( <COMMANDS> )

` <COMMANDS> `

The command substitution expands to the output of commands. These commands are executed in a subshell, and their stdout data is what the substitution syntax expands to.

All trailing newlines are removed (below is an example for a workaround).

In later steps, if not quoted, the results undergo [word splitting](https://wiki.bash-hackers.org/syntax/expansion/wordsplit) and [pathname expansion](https://wiki.bash-hackers.org/syntax/expansion/globs). You have to remember that, because the word splitting will also remove embedded newlines and other IFS characters and break the results up into several words. Also you'll probably get unexpected pathname matches. If you need the literal results, quote the command substitution!

The second form `COMMAND` is more or less obsolete for Bash, since it has some trouble with nesting ("inner" backticks need to be escaped) and escaping characters. Use $(COMMAND), it's also POSIX!

When you [call an explicit subshell](https://wiki.bash-hackers.org/syntax/ccmd/grouping_subshell) (COMMAND) inside the command substitution $(), then take care, this way is wrong:

$((COMMAND))

Why? because it collides with the syntax for [arithmetic expansion](https://wiki.bash-hackers.org/syntax/expansion/arith). You need to separate the command substitution from the inner (COMMAND):

$( (COMMAND) )

## Specialities

When the inner command is only an input redirection, and nothing else, for example

$( <FILE )

# or

` <FILE `

then Bash attempts to read the given file and act just if the given command was cat FILE.

## A closer look at the two forms

In general you really should only use the form $(), it's escaping-neutral, it's nestable, it's also POSIX. But take a look at the following code snips to decide yourself which form you need under specific circumstances:

Nesting

Backtick form `…` is not directly nestable. You will have to escape the "inner" backticks. Also, the deeper you go, the more escape characters you need. Ugly.

echo `echo `ls`` # INCORRECT

echo `echo \`ls\`` # CORRECT

echo $(echo $(ls)) # CORRECT

Parsing

All is based on the fact that the backquote-form is simple character substitution, while every $()-construct opens an own, subsequent parsing step. Everything inside $() is interpreted as if written normal on a commandline. No special escaping of nothing is needed:

echo "$(echo "$(ls)")" # nested double-quotes - no problem

Constructs you should avoid

It's not all shiny with $(), at least for my current Bash (3.1.17(1)-release.  Update: Fixed since 3.2-beta together with a misinterpretion of '))' being recognized as arithmetic expansion [by redduck666]). This command seems to incorrectly close the substitution step and echo prints "ls" and ")":

echo $(

# some comment ending with a )

ls

)

It seems that every closing ")" confuses this construct. Also a (very uncommon ) construct like:

echo $(read VAR; case "$var" in foo) blah ;; esac) # spits out some error, when it sees the ";;"

# fixes it:

echo $(read VAR; case "$var" in (foo) blah ;; esac) # will work, but just let it be, please ;-)

Conclusion:

In general, the $() should be the preferred method:

* it's clean syntax
* it's intuitive syntax
* it's more readable
* it's nestable
* its inner parsing is separate

## Examples

To get the date:

DATE="$(date)"

To copy a file and get cp error output:

COPY\_OUTPUT="$(cp file.txt /some/where 2>&1)"

Attention: Here, you need to redirect cp STDERR to its STDOUT target, because command substitution only catches STDOUT!

Catch stdout and preserve trailing newlines:

var=$(echo -n $'\n'); echo -n "$var"; # $var == ""

var=$(echo -n $'\n'; echo -n x); var="${var%x}"; echo -n "$var" # $var == "\n"

This adds "x" to the output, which prevents the trailing newlines of the previous commands' output from being deleted by $().

By removing this "x" later on, we are left with the previous commands' output with its trailing newlines.