

Command Substitution

In this lesson, you'll learn about command substitution and other ways of substituting commands.

WE'LL COVER THE FOLLOWING

- How Important is this Lesson?
- Command Substitution Example
- The Two Command Substitution Methods
 - The 'Dollar-Bracket' Method: `$()`
 - The 'Backtick' Method
- What You Learned
- What Next?
- Exercises

How Important is this Lesson?


The contents of this lesson are vital if you intend to write or work on shell scripts.

Command Substitution Example

When writing bash scripts you often want to take the standard output of one command and 'drop' it into the script as though you had written that into it.

An example may help illustrate this idea. Type these commands:

```
hostname # 'hostname' outputs the name of the host
echo 'My hostname is: $(hostname)' # Single quotes do not call the hostname command
echo "My hostname is: $(hostname)" # Double quotes do, similar to variable dereferencing
```



Type the above code into the terminal in this lesson.

If those lines are placed in a script, it will output the hostname of the host the script is running on. This can make your script much more dynamic. You can set variables based on the output of commands, add debug, and so on, just as with any other programming language.

You may have noticed that if wrapped in single quotes, the special meaning of the `$` sign is ignored again!

The Two Command Substitution Methods `#`

There are two ways to do command substitution:

```
echo "My hostname is: `hostname`" # backticks method
echo "My hostname is: $(hostname)" # dollar-bracket method
```



Type the above code into the terminal in this lesson.

These give the same output and the backticks perform the same function. So which should you use?

The 'Dollar-Bracket' Method: `$()` `#`

Type this:

```
mkdir tmp
cd tmp
echo $(touch $(ls ..))
cd -
rm -rf tmp
```



Type the above code into the terminal in this lesson.

What happened there?

- In **lines 1 and 2** you created and moved into a folder
- This temporary folder is cleaned up by **lines 4-5**
- **Line 3** is best read from the innermost parentheses outwards
 - The `ls ..` command is run in the innermost parentheses. This outputs the contents of the parent directory

- This output is substituted in over the `$(ls ..)`. The ‘words’ returned are placed as the arguments to the `touch` command. The `touch` command creates a set of empty files, based on the list of the parent directory’s contents
- The `echo` command takes the output of the above substitution, in this case nothing, as touch does not produce any output

So, in summary:

- **Line 3** outputs the list of files of the parent directory
- Those filenames are also created locally as empty files

Line 3 is an example of how subcommands can be nested. As you can see, the nesting is simple - just place a command wrapped inside a `$()` inside another command wrapped inside a `$()` and bash will then substitute the output of each command from the inside out for you in the appropriate order.

Now let’s look at the equivalent code with backticks.

The ‘Backtick’ Method

Type this out:

```
echo `touch `ls ..`  
cd ..
```



Type the above code into the terminal in this lesson.

To nest the **backtick** version, you have to *escape* the inner backtick with a backslash, so bash knows which level the backtick should be interpreted at.

To demonstrate the difficulty here, a simple example suffices:

```
echo $(echo hello1 $(echo hello2)) # Nesting echoes  
echo `echo hello1 `echo hello2`` # Nesting echoes with backticks fails  
echo `echo hello1 `echo hello2\`` # Nesting echoes with backticks and backslashes ok
```



Type the above code into the terminal in this lesson.

For historical reasons, the backtick form is still very popular, but I prefer the `$()` form because of the simplicity of managing nesting. You need to be aware of both, though, if you are looking at others' code!

If you want to see how messy things can get, compare these two lines:

```
echo `echo \`echo \\\`echo inside\\\`\\`  
echo $(echo $(echo $(echo inside)))
```



Type the above code into the terminal in this lesson.

and consider which one is easier to read (and write)!

What You Learned

- What *command substitution* is
- How it relates to quoting
- The two *command substitution* methods
- Why one method is generally preferred over the other

What Next?

Next you will cover **tests**, which allow you to use what you've learned so far to make your bash code conditional in a flexible and dynamic way.

Exercises

- 1) Try various command substitution commands, and plug in variables and quotes to see what happens.
- 2) Explain why three backslashes are required in the last example.