A shell colon does nothing. Use it anyway.

I've written more shell scripts than I can count, but I still stumble upon tricks that honestly blow my mind far too often than I care to admit. Latest thing that blew my head clean off? The shell colon.

Published 23 July 2026 at 03:53 UTC
Modified 26 July 2026 at 05:33 UTC
Author Filip Roséen
Tags #shell #posix #unix

▶ **Table of Contents**

In a land far-far away..

... there was once a far too cold cup of coffee next to a freshly brewed far-too-hot one. Four different terminals where three could have been closed an hour ago, and a shell script which I really (really) did not want to write.

Who would have thought a single colon would be the one to save the ~~day~~ night?

> **Note**: Want your mind blown straight away? See more colons in the limelight.

Checking for required arguments

This is a familiar dance, it's pretty much muscle memory by this point. You have a script, it takes a few arguments, and some of them are mandatory; alright, an if-statement like so many times before:

```
if [ -z "$1" ]; then
   echo "missing argument, aborting." 1>&2
   exit 1
fi

echo "Hello $1!"
```

Though.. what if I told you the above four lines could be replaced by just... one?

```
: "${1:?missing argument, aborting.}"
echo "Hello $1!"
```

```
$ bash example.sh
example.sh: line 1: 1: missing argument, aborting.

$ bash example.sh refp
Hello refp!
```

And look what happens if we refer to a variable with a proper name — it's the same behavior as previously but easier to spot; the diagnostic includes the name of our variable!

```
: "${GREET_NAME:?missing argument, aborting.}"
echo "Hello $GREET_NAME!"
```

```
$ bash greet.sh
greet.sh: line 1: GREET_NAME: missing argument, aborting.
```

Parameter expansion and the story of `:?`

There are two things going on in the previous snippet, and you are correct in identifying that one part is using parameter expansion:

- The syntax `${name:?diagnostic}` checks whether `$name` is unset or empty — if it is, the diagnostic is printed to *stderr* and the shell exits with a non-zero status, otherwise;
- if the variable is set, it is equivalent to `$name`.

That.. other colon

So that's one colon, but what about that other one, the one who sits alone at the beginning of the line?

- `:` is the null-command — a builtin that does nothing but evaluate its arguments and discard the result.
- `:` is old — it goes all the way back to the 1971 Thompson shell where it doubled as a label and Unix's very first comment marker.
- `:` two eyes staring at you in the dark, with love.

More colons in the limelight

Perhaps we have already established that there is more to `:` than meets the eye, but to prove the real magic of the null-command — here are a few usages that blew my mind.

```
: "${DATA_DIR:=/var/data}"        # set defaults, : swallows the result
: "${RETRIES:=3}"                 # instead of running it as a command
```

```
: > error.log                     # truncate error.log
: > error.log > access.log        # truncate both error.log and access.log
```

```
( : < dataset.json ) && echo YES # is dataset.json readable?
( : >> result.json ) && echo YES # is result.json writable?
```

```
trap : INT                        # trap requires a command
sleep 60                          # sleep is interruptible
```

```
set -u                            # error on unset variables
: "$DEPLOY_ENV" "$HOST"           # check DEPLOY_ENV and HOST
```

```
if some-command; then
    :                             # command required
else
    echo "command failed"
fi
```

Con-colon-sion

So, if you are like me and prefer less typing (gotta go fast) — the null-command and parameter expansion are a pair worth studying before your coffee goes cold.

And also.. isn't this — magic?

```
set  : : : : : : : : : : : : : : : : : : : :

while : colons are more than "${1:?magic}"; do
    echo "$*" && shift
done
```

Note: The above example is safe to run locally, try it!

Frequently Asked Questions

After reading a few comments online, it seems I skipped over some things worth explaining. I will keep this section updated as questions come up.

- **Why do I need the null-command? Doesn't the expansion happen without the colon?**

  The parameter expansion will happen regardless, but without a null-command or similar usage the shell will treat the resulting string as a command to run.

  ```
  % ${HELLO:=123}
  zsh: command not found: 123
  ```

  If we prefix our parameter-expansion with the null-command, the result is discarded, but the expression is still evaluated (setting `HELLO` to `123`).

  ```
  % : ${HELLO:=123}
  % echo $HELLO
  123
  ```

- **Why use the null-command when I could do `VAR=${VAR:-default-value}`?**

  This at its core boils down to personal preference, but using our beloved colon we can shrink the number of potential typos to one (rather than two):

  ```
  : "${DATA_DIR:=/var/data}"           # <- DATA_DIR mentioned once (1)
  DATA_DIR="${DATA_DRI:-/var/data}"    # <- oops                    (2)
  ```

- **Why would I do any of these when it hurts readability?**

  There are a few common misconceptions about the — very contrived — examples presented in this article, questions regarding their usefulness, or if one should ever-ever use these things at all.

  Let's take the below as an example.

  ```
  if some-command; then
      :                              # command required
  else
      echo "command failed"
  fi
  ```

Of course you can use `if ! some-command` and get rid of the branching, but that isn't the point of the snippet — the point is to show that the null-command can be used in places where a command is required, but nothing is desired.

It was never meant to be read as *"oh, if I have an if-statement I should use null-command"*, it is intended to be read as *"oh, if I am in a context which requires a command, I can use the null-command to no-op it"*.

Clever real-world usage of `:` in the wild

User ifphilipe posted what follows as a comment on news.ycombinator.com, which is the direct equivalent of needing a command which does absolutely nothing.

> I use the colon as EDITOR with Git when I want to do an interactive rebase combined with auto squash without having to edit the todo list. I have an alias[1] for that which I call a quick interactive rebase:
>
> ```
> riq = -c sequence.editor=: rebase --interactive
> ```

Could this article have used the above rather than the so much debated if-statement? If I was clever enough to come up with it — yes.. but would it be as comprehensible in terms of understanding what `:` really is? I strongly suspect not.