

Windows Batch Scripting: Variables

- [Overview](#)
- [Part 1 – Getting Started](#)
- [Part 2 – Variables](#)
- [Part 3 – Return Codes](#)
- [Part 4 – stdin, stdout, stderr](#)
- [Part 5 – If/Then Conditionals](#)
- [Part 6 – Loops](#)
- [Part 7 – Functions](#)
- [Part 8 – Parsing Input](#)
- [Part 9 – Logging](#)
- [Part 10 – Advanced Tricks](#)

Today we'll cover variables, which are going to be necessary in any non-trivial batch programs. The syntax for variables can be a bit odd, so it will help to be able to understand a variable and how it's being used.

Variable Declaration

DOS does not require declaration of variables. The value of undeclared/uninitialized variables is an empty string, or `""`. Most people like this, as it reduces the amount of code to write. Personally, I'd like the option to require a variable is declared before it's used, as this catches silly bugs like typos in variable names.

Variable Assignment

The `SET` command assigns a value to a variable.

```
SET foo=bar
```

NOTE: Do not use whitespace between the name and value; `SET foo = bar` will *not* work but `SET foo=bar` will work.

The `/A` switch supports arithmetic operations during assignments. This is a useful tool if you need to validated that user input is a numerical value.

```
SET /A four=2+2  
4
```

A common convention is to use lowercase names for your script's variables. System-wide variables, known as environmental variables, use uppercase names. These environmental describe where to find certain things in your system, such as `%TEMP%` which is path for temporary files. DOS

is case insensitive, so this convention isn't enforced but it's a good idea to make your script's easier to read and troubleshoot.

WARNING: `SET` will always overwrite (clobber) any existing variables. It's a good idea to verify you aren't overwriting a system-wide variable when writing a script. A quick `ECHO %foo%` will confirm that the variable `foo` isn't an existing variable. For example, it might be tempting to name a variable "temp", but, that would change the meaning of the widely used "%TEMP%" environmental variable. DOS includes some "dynamic" environmental variables that behave more like commands. These dynamic variables include `%DATE%`, `%RANDOM%`, and `%CD%`. It would be a bad idea to overwrite these dynamic variables.

Reading the Value of a Variable


In most situations you can read the value of a variable by prefixing and postfixing the variable name with the `%` operator. The example below prints the current value of the variable `foo` to the console output.

```
C:\> SET foo=bar
C:\> ECHO %foo%
bar
```

There are some special situations in which variables do not use this `%` syntax. We'll discuss these special cases later in this series.

Listing Existing Variables

The `SET` command with no arguments will list all variables for the current command prompt session. Most of these variables will be system-wide environmental variables, like `%PATH%` or `%TEMP%`.


 Screenshot of the `SET` command

NOTE: Calling `SET` will list all regular (static) variables for the current session. This listing excludes the dynamic environmental variables like `%DATE%` or `%CD%`. You can list these dynamic variables by viewing the end of the help text for `SET`, invoked by calling `SET /?`

Variable Scope (Global vs Local)

By default, variables are global to your entire command prompt session. Call the `SETLOCAL` command to make variables local to the scope of your script. After calling `SETLOCAL`, any variable assignments revert upon calling `ENDLOCAL`, calling `EXIT`, or when execution reaches the end of file (EOF) in your script.

This example demonstrates changing an existing variable named `foo` within a script named `HelloWorld.cmd`. The shell restores the original value of `%foo%` when `HelloWorld.cmd` exits.

 Demonstration of the `SETLOCAL` command

A real life example might be a script that modifies the system-wide `%PATH%` environmental variable, which is the list of directories to search for a command when executing a command.

 Demonstration of the `SETLOCAL` command

Special Variables

There are a few special situations where variables work a bit differently. The arguments passed on the command line to your script are also variables, but, don't use the `%var%` syntax. Rather, you read each argument using a single `%` with a digit 0-9, representing the ordinal position of the argument. You'll see this same style used later with a hack to create functions/subroutines in batch scripts.

There is also a variable syntax using `!`, like `!var!`. This is a special type of situation called delayed expansion. You'll learn more about delayed expansion in when we discuss conditionals (if/then) and looping.

Command Line Arguments to Your Script

You can read the command line arguments passed to your script using a special syntax. The syntax is a single `%` character followed by the ordinal position of the argument from 0 – 9. The zero ordinal argument is the name of the batch file itself. So the variable `%0` in our script `HelloWorld.cmd` will be "HelloWorld.cmd".

The command line argument variables are `* %0`: the name of the script/program as called on the command line; always a non-empty value `* %1`: the first command line argument; empty if no arguments were provided `* %2`: the second command line argument; empty if a second argument wasn't provided `* ...`: `* %9`: the ninth command line argument

NOTE: DOS does support more than 9 command line arguments, however, you cannot directly read the 10th argument or higher. This is because the special variable syntax doesn't recognize `%10` or higher. In fact, the shell reads `%10` as postfix the `%0` command line argument with the string "0". Use the `SHIFT` command to pop the first argument from the list of arguments, which "shifts" all arguments one place to the left. For example, the the second argument shifts from position `%2` to `%1`, which then exposes the 10th argument as `%9`. You will learn how to process a large number of arguments in a loop later in this series.

Tricks with Command Line Arguments

Command Line Arguments also support some really useful optional syntax to run quasi-macros on command line arguments that are file paths. These macros are called variable substitution support and can resolve the path, timestamp, or size of file that is a command line argument. The documentation for this super useful feature is a bit hard to find – run 'FOR /?' and page to the end of the output.

- `%~I` removes quotes from the first command line argument, which is super useful when working with arguments to file paths. You will need to quote any file paths, but, quoting a file path twice will cause a file not found error.

```
SET myvar=%~I
```

- `%~fI` is the full path to the folder of the first command line argument
- `%~fsI` is the same as above but the extra `s` option yields the DOS 8.3 short name path to the first command line argument (e.g., `C:\PROGRA~1` is usually the 8.3 short name variant of `C:\Program Files`). This can be helpful when using third party scripts or programs that don't handle spaces in file paths.
- `%~dpI` is the full path to the parent folder of the first command line argument. I use this trick in nearly every batch file I write to determine where the script file itself lives. The syntax `SET parent=%~dp0` will put the path of the folder for the script file in the variable `%parent%`.
- `%~nxI` is just the file name and file extension of the first command line argument. I also use this trick frequently to determine the name of the script at runtime. If I need to print messages to the user, I like to prefix the message with the script's name, like `ECHO %~n0: some message` instead of `ECHO some message`. The prefixing helps the end user by knowing the output is from the script and not another program being called by the script. It may sound silly until you spend hours trying to track down an obtuse error message generated by a script. This is a nice piece of polish I picked up from the Unix/Linux world.

Some Final Polish

I always include these commands at the top of my batch scripts:

```
SETLOCAL ENABLEEXTENSIONS
SET me=%~n0
SET parent=%~dp0
```

The `SETLOCAL` command ensures that I don't clobber any existing variables after my script exits. The `ENABLEEXTENSIONS` argument turns on a very helpful feature called command processor extensions. Trust me, you want command processor extensions. I also store the name of the script (without the file extension) in a variable named `%me%`; I use this variable as the prefix to any printed messages (e.g. `ECHO %me%: some message`). I also store the parent path to the script in a variable named `%parent%`. I use this variable to make fully qualified filepaths to any other files in the same directory as our script.