Shell Scripting 7: Details & Advanced Features

1. Shells and Variables
2. Control Constructs
3. Shell Expansions
4. Functions and Arrays
5. Bash Development
6. Practical Advice
7. **Bash Details and Advanced Features**
   - **command interpretation details**
   - **exit status details**

©Norman Carver

Command Interpretation Details

When Bash reads and executes a command, it does the following:

1. Reads its input from a file or from the user's terminal.
2. Breaks the input into **words** and **operators**, obeying the **quoting rules**.
3. **Alias** expansion is performed (but not in scripts by default).
4. Parses the tokens into **simple commands** and **compound commands**.
5. Performs the various shell **expansions**.
6. Performs any necessary **redirections** and removes the redirection operators and their operands from the argument list.
7. Executes the commands.
8. Optionally waits for the command to complete and collects its exit status

©Norman Carver

Command Interpretation Details (contd.)

There are seven kinds of **expansion**:

1. brace expansion
2. tilde expansion
3. parameter and variable expansion
4. command substitution
5. arithmetic expansion
6. word splitting
7. filename expansion

The order of expansions is:
brace expansion; tilde expansion; parameter, variable, and arithmetic expansion and command substitution (done in a left-to-right fashion); word splitting; filename expansion.

©Norman Carver

Command Interpretation Details (contd.)

Only brace expansion, word splitting, and filename expansion can change the number of words of the expansion; other expansions expand a single word to a single word. The only exceptions to this are the expansions of `"$@"` and `"${name[@]}"` .

Variable assignment `var=[val]`:
Word splitting is not performed, with the exception of `"$@"`.

Test-command `[[ ... ]]`:
Word splitting and filename expansion are not performed on the words between the `[[` and `]]`; tilde expansion, parameter and variable expansion, arithmetic expansion, command substitution, process substitution, and quote removal are performed.

Command substitution:
If the substitution appears within double quotes, word splitting and filename expansion are not performed on the results.

©Norman Carver

Command Interpretation Details (contd.)

Word Splitting:
The shell scans the results of parameter expansion, command substitution, and arithmetic expansion that did not occur within double quotes for word splitting.

The shell treats each character of `$IFS` as a delimiter, and splits the results of the other expansions into "**words**."

The *default value* of `IFS` is: `<space><tab><newline>`

Word splitting behavior can be changed by changing the value of `IFS`.

E.g., doing `IFS=$'\n'` will cause splitting only on newlines.

This can be useful when capturing multiple lines which contain filenames that may include embedded whitespace.

©Norman Carver

Command Interpretation Details (contd.)

Word splitting details:

- If `IFS` is unset or the default, then any sequence of `IFS` characters serves to delimit words.
- If `IFS` has a value other than the default, then sequences of the whitespace characters space and tab are ignored at the beginning and end of the word, as long as the whitespace character is in the value of `IFS` (an `IFS` whitespace character).
- Any character in `IFS` that is not `IFS` whitespace, along with any adjacent `IFS` whitespace characters, delimits a field.
- A sequence of `IFS` whitespace characters is also treated as a delimiter.
- If the value of `IFS` is null, no word splitting occurs.
- Explicit null arguments (`""` or `''`) are retained.
- Unquoted implicit null arguments, resulting from the expansion of parameters that have no values, are removed.
- If a parameter with no value is expanded within double quotes, a null argument results and is retained.
- If no expansion occurs, no splitting is performed.

©Norman Carver

Exit Status Details

Many commands that are run in shell scripts can *fail*.

Writing *robust* shell scripts means including code that ensures commands have succeeded before proceeding in the script.

Otherwise, invalid data may be passed on, leading to incorrect results or unexpected and difficult to interpret error messages.

(Remember also that we may need to suppress error messages from commands by *redirecting standard error:* "`2>/dev/null`".)

Success/failure of commands is most commonly determined by their **exit status**.

While the exit status of a **simple command** is clear, this is not necessarily the case for other types of Bash commands.

©Norman Carver

Exit Status Details (contd.)

Pipelines are the most complex:

- pipeline exit status is the exit status of the *last command* in the pipeline
- if Bash `pipefail` *option* enabled, then is exit status of last (rightmost) command *to exit with a non-zero status*, or zero if all commands exit successfully
- if "!" symbol precedes the pipeline, exit status is the negation of the exit status
- shell *waits* for all commands in the pipeline to terminate before returning status

Enable or disable `pipefail` option with:
"`set -o pipefail`" or "`set +o pipefail`".

©Norman Carver

Exit Status Details (contd.)

Other types of commands have exit status as follows:

- **lists**:
  exit status is the exit status of the last command executed in the `;`/`&&`/`||` list
- **while/until**:
  exit status is the exit status of the last command executed in body, or zero/success if none was executed
- **for-in**: exit status of the last command that executes, but if empty list (no commands executed) then exit status is zero
- **for**: exit status of the last command executed in body; failure if any of the three `for` form expressions is invalid
- **if-elif-else**: exit status of the last command executed, or zero/success if no condition tested true
- **case**: zero/success if no pattern is matched, or exit status of the (one) `commands-list` that was executed

©Norman Carver