> *This wiki is automatically published from ohmyzsh/wiki. To edit this page, go to ohmyzsh/wiki, make your changes and submit a Pull Request.*

General Code Style

While you should follow the code style that's already there for files that you're modifying, the following are required for any new code.

Indentation

Indent 2 spaces. No tabs.

Use blank lines between blocks to improve readability. Indentation is two spaces. Whatever you do, don't use tabs. For existing files, stay faithful to the existing indentation.

Line Length and Long Strings

Maximum line length is 80 characters.

If you have to write strings that are longer than 80 characters, this should be done with a "here document" or an embedded newline if possible. Literal strings that have to be longer than 80 chars and can't sensibly be split are okay, but it's strongly preferred to find a way to make it shorter.

Bad:

```
long_string_1="I am an exceptionallllllllllly
looooooooooooooooooooooooooooooooooooooooong string."
```

Good:

```
cat <<END;
I am an exceptionallllllllllly
looooooooooooooooooooooooooooooooooooooooong string.
END
```

Good:

```
long_string_2="I am an exceptionallllllllllly
 looooooooooooooooooooooooooooooooooooooooong string."
```

Pipelines

Pipelines should be split one per line if they don't all fit on one line.

If a pipeline all fits on one line, it should be on one line.

If not, it should be split at one pipe segment per line with the pipe on the newline and a 2 space indent for the next section of the pipe. This applies to a chain of commands combined using '|' as well as to logical compounds using '||' and '&&'.

Bad:

```
command1 | command2 | command3 | command4 | command5 | command6 | command7
```

Good:

```
command1 \
  | command2 \
  | command3 \
  | command4
```

Good: **All fits on one line**

```
command1 | command2
```

Use environment variables

When possible, use environment variables instead of shelling out to a command.

Bad:

```
$(pwd)
```

Good:

```
$PWD
```

TODO: Add a list of all environment variables you can use.

If / For / While

Put `; do` and `; then` on the same line as the `while`, `for` or `if`.

Good:

```
for dir in ${dirs_to_cleanup}; do
  if [[ -d "${dir}/${ORACLE_SID}" ]]; then
    log_date "Cleaning up old files in ${dir}/${ORACLE_SID}"
    rm "${dir}/${ORACLE_SID}/"*
    if [[ "$?" -ne 0 ]]; then
      error_message
    fi
  else
    mkdir -p "${dir}/${ORACLE_SID}"
    if [[ "$?" -ne 0 ]]; then
      error_message
    fi
  fi
done
```

Variables

Naming Conventions

Meaningful self-documenting names should be used. If the variable name does not make it reasonably obvious as to the meaning of the variable, appropriate comments should be added.

Bad:

```
local TitleCase=""
local camelCase=""
```

Good:

```
local snake_case=""
```

Uppercase strings are reserved for global variables. (WARNING: In functions, only variables explicitly declared as local like `local foo=""` are really local.)

Bad:

```
local UPPERCASE=""
```

Good:

```
UPPERCASE=""
```

Variable names should not clobber command names, such as `dir` or `pwd`.

Bad:

```
local pwd=""
```

Good:

```
local pwd_read_in=""
```

Variable names for loop indexes should be named similarly to any variable you're looping through.

Good:

```
for zone in ${zones}; do
  something_with "${zone}"
done
```

Use local variables

Ensure that local variables are only seen inside a function and its children by using `local` or other `typeset` variants when declaring them. This avoids polluting the global name space and inadvertently setting or interacting with variables that may have significance outside the function.

Bad:

```
function func_bad() {
  global_var=37  #  Visible only within the function block
                 #  before the function has been called.
}

echo "global_var = $global_var"  # Function "func_bad" has not yet been called,
                                 # so $global_var is not visible here.

func_bad
echo "global_var = $global_var"  # global_var = 37
                                 # Has been set by function call.
```

Good:

```
func_good() {
  local local_var=""
  local_var=37
  echo $local_var
}

echo "local_var = $local_var" # local

func_good
echo "local_var = $local_var" # still local

global_var=$(func_good)
echo "global_var = $global_var" # move function result to global scope
```

In the next example, lots of global variables are used over and over again, but the script "unfortunately" works anyway. The `parse_json()` function does not even return a value and the two functions shares their variables. You could also write all this without any function; this would have the same effect.

***Bad:* with global variables**

```
#!/bin/bash

parse_json() {
  parent_prop=$1
  prop=$2

  # "'TODO: fix this hack' is the Snooze button of development" - @iamdevloper
  result=`echo $json \
    | sed 's/\\\\\//\//g' \
    | sed 's/^[\ ]*//g' \
    | sed 's/[{}]//g' \
    | awk -v k="$parent_prop" '{n=split($0,a,"\","); for (i=1; i<=n; i++) print
a[i]}' \
    | sed 's/\"\:\"/\|/g' \
    | sed 's/"[\,]/ /g' \
    | sed 's/\"//g' \
```

```
    | grep "$prop|" \
    | sed "s/^$prop|//g"`
}

parse_ubuntuusers_json() {
  json=`curl -s -X GET 'https://suckup.de/planet-ubuntuusers-json/json.php?
callback='`

  parse_json "posts" "title"
  mapfile -t titles_array <<< "$result"

  parse_json "posts" "date"
  mapfile -t dates_array <<< "$result"

  counter=0
  for i in "${titles_array[@]}"; do
    echo "${titles_array[$counter]} | ${dates_array[$counter]}"
    let counter+=1
  done
}

parse_ubuntuusers_json

echo "foobar: $counter - $i"
```

In shell scripts, it is less common that you really want to reuse the functionality, but the code is much easier to read if you write small functions with appropriate return values and parameters.

***Good:* with local variables**

```
#!/bin/zsh

parse_json() {
  local json=`cat $1`
  local parent_prop=$2
  local prop=$3

  # "'TODO: this hack' is the Snooze button of development" - @iamdevloper
  echo $json \
    | sed 's/\\\\\//\//g' \
    | sed 's/^[\ ]*//g' \
    | sed 's/[{}]//g' \
    | awk -v k="$parent_prop" '{n=split($0,a,"\","); for (i=1; i<=n; i++) print
a[i]}' \
    | sed 's/\"\:\"/\|/g' \
    | sed 's/"[\,]/ /g' \
    | sed 's/\"//g' \
    | grep "$prop|" \
    | sed "s/^$prop|//g"
}

parse_ubuntuusers_json() {
```

```
  local temp_file=`mktemp`
  local json=`curl -s -X GET 'https://suckup.de/planet-ubuntuusers-json/json.php?
callback=' -o $temp_file`

  local titles=`parse_json "$temp_file" "posts" "title"`
  local titles_array
  mapfile -t titles_array <<< "$titles"

  local dates=`parse_json "$temp_file" "posts" "date"`
  local dates_array
  mapfile -t dates_array <<< "$dates"

  local counter=0 i
  for i in "${titles_array[@]}"; do
    echo "${titles_array[$counter]} | ${dates_array[$counter]}"
    let counter+=1
  done

  rm $temp_file
}

parse_ubuntuusers_json

echo "foobar: $counter - $i"
```

Constants and Environment Variable Names

All caps, separated with underscores, declared at the top of the file. Constants and anything exported to the environment should be capitalized.

Constant:

```
readonly PATH_TO_FILES='/some/path'
```

Constant and environment:

```
declare -xr ORACLE_SID='PROD'
```

Some things become constant at their first setting (for example, via `getopts`). Thus, it's okay to set a constant in `getopts` or based on a condition, but it should be made `readonly` immediately afterwards. Note that `declare` doesn't operate on global variables within functions, so `readonly` or `export` is recommended instead.

```
VERBOSE='false'
while getopts 'v' flag; do
  case "${flag}" in
    v) VERBOSE='true' ;;
  esac
done
readonly VERBOSE
```

Read-only Variables

Use `readonly` or `declare -r` to ensure they're read only. As globals are widely used in shell, it's important to catch errors when working with them. When you declare a variable that is meant to be read-only, make this explicit.

```
zip_version="$(dpkg --status zip | grep Version: | cut -d ' ' -f 2)"
if [[ -z "${zip_version}" ]]; then
  error_message
else
  readonly zip_version
fi
```

Functions

Naming Conventions

Lower-case, with underscores to separate words. Parentheses are required after the function name. The `function` keyword is optional when `()` is present after the function name, but it aids readability and prevents conflicts with alias declarations.

The opening brace should appear on the same line as the function name.

Bad:

```
function my_bad_func {
  ...
}
```

Good:

```
function my_good_func() {
  ...
}
```

Private or utility functions should be prefixed with an underscore:

Good:

```
_helper-util()  {
  ...
}
```

Use and check return values

After a script or function terminates, a `$?` from the command line gives the exit status of the script, that is, the exit status of the last command executed in the script, which is, by convention, 0 on success or an integer in the range 1 - 255 on error.

Bad:

```
my_bad_func() {
  # didn't work with zsh / bash is ok
  #read lowerPort upperPort < /proc/sys/net/ipv4/ip_local_port_range

  for port in $(seq 32768 61000); do
    for i in $(netstat_used_local_ports); do
      if [[ $used_port -eq $port ]]; then
        continue
      else
        echo $port
      fi
    done
  done
}
```

Good:

```
my_good_func() {
  # didn't work with zsh / bash is ok
  #read lowerPort upperPort < /proc/sys/net/ipv4/ip_local_port_range

  for port in $(seq 32768 61000); do
    for i in $(netstat_used_local_ports); do
      if [[ $used_port -eq $port ]]; then
        continue
      else
        echo $port
        return 0
      fi
    done
  done

  return 1
}
```

Check return values

Always check return values and give informative error messages. For unpiped commands, use `$?` or check directly via an if statement to keep it simple. Use nonzero return values to indicate errors.

Bad:

```
mv "${file_list}" "${dest_dir}/"
```

Good:

```
mv "${file_list}" "${dest_dir}/" || exit 1
```

Good:

```
if ! mv "${file_list}" "${dest_dir}/"; then
  echo "Unable to move ${file_list} to ${dest_dir}" >&2
  exit 1
fi
```

Good: **use "$?" to get the last return value**

```
mv "${file_list}" "${dest_dir}/"
if [[ $? -ne 0 ]]; then
  echo "Unable to move ${file_list} to ${dest_dir}" >&2
  exit 1
fi
```

Features and Bugs

Command Substitution

Use `$(command)` instead of backticks.

Nested backticks require escaping the inner ones with `\`. The `$(command)` format doesn't change when nested and is easier to read.

Bad:

```
var="`command \`command1\``"
```

Good:

```
var="$(command "$(command1)")"
```

Eval

Eval is evil! Eval munges the input when used for assignment to variables and can set variables without making it possible to check what those variables were. Avoid `eval` if possible.

References

- Shell Style Guide
- BASH Programming - Introduction HOW-TO
- Linux kernel coding style