🖌 BASH Style Guide

- Some programming styles are **personal preferences**.
- Some programming styles are **company guidelines**.
- Some programming styles are **community conventions**.

*This document contains my **personal** preferences only.*

These style guidelines are intended for Bash libraries and applications.
This is less relevant for simple shell scripts, although perhaps useful.

Please be kind.

I am sharing this in the hopes that some may find this interesting or useful. ~ @bex (https://github.com/beccasaurus)

> *Note: there are no shout outs to any projects or tools, generic Bash (https://en.wikipedia.org/wiki/Bash_(Unix_shell)) code only.*

⌨️ Variables

myVariable

Name non-global variables in `camelCase`

> This is probably the most controversial thing in this guide.
>
> Name variables however you like :P

`MYAPP_PUBLIC_VARIABLE`

Name public global variables in `UPPERCASE`

Prefix your global variable with something to identify to your users that the variable is associated with your script or program, e.g. `MYAPP_CONFIG_FILE` .

This helps to **avoid global naming collisions** with variables in other libraries which your users may be using.

`_myApp_privateVar`

*This recommendation is for **library code**.*
Scripts and programs do not generally need to prefix variables.

Prefix your private variables with something associated with your script or program, e.g. `_myApp_configFile` .

This helps to **avoid global naming collisions** with variables in other libraries which your users may be using.

> 💡 Reminder: variables you assign will be available to the scope of other functions that you call.
>
> This includes `local` variables.

> ℹ️ All private variables should be `local` variables inside functions (see more)

Example

```
myFunction() {
  local privateValue=42 # <-- locals are in scope for called functions
  anotherFunction
}

anotherFunction() {
  echo "The private value is $privateValue" # <-- 'local' value available
}

myFunction
# => "The private value is 42"
```

`declare`

Use `declare` to define variables with dynamic names.

Declare Dynamic Name Variable

```
# This dynamic variable name is set in a variable
variableName=foo

# Declare the variable using its name from a variable
declare "$variableName=42"

echo "$foo"
# => "42"

# This can also be used to modify the value
declare "$variableName=4"

echo "$foo"
# => "4"
```

Declare Dynamic Name Array

```
# This dynamic variable name is set in a variable
variableName=foo

# 'foo' is undefined and has a zero length
echo "${#foo[@]}"
# => 0

# Declare the variable using its name from a variable
declare -a "$variableName=(hello world)"

# 'foo' is defined as an array with a count of 2
echo "${#foo[@]}"
# => 2

# The first array item is "hello"
echo "${foo[0]}"
# => "hello"
```

Modify Dynamic Name Array

```
# This can also be used to push new values onto the array
declare -a "$variableName+=(goodnight moon)"

# Print all values of the array
echo "${foo[*]}"
# => "hello world goodnight moon"
```

> 💡 **Note:** Using `declare` in a function assigns the variable as a `local` .
>
> Bash 4.2 adds `declare -g` which assigns the variable in the global scope.

`typeset -n`

Use `typeset -n` to get a reference to a variable by using *the variable name*.

> ℹ️ Note: this is only available in Bash 4.3 and above

Example

```
hello="World"

# Modify the value of hello using a variable which contains the variable name "hello"
variableName=hello
typeset -n theVariable="$variableName"

echo "$theVariable"
# => "World"

theVariable="change me"

echo "$theVariable"
# => "change me"

echo "$hello"
# => "change me"
```

`(set -o posix; set)`

This is the correct way to get a definition of a variable:

```
foo=5
foo_list=(a b c)

(set -o posix; set) | grep ^foo
# foo=5
# foo_list=([0]="a" [1]="b" [2]="c")
```

> 💡 **Tip:** this is a great way to serialize variables including Bash arrays!
>
> The syntax provided by `(set -o posix; set)` can be safely `eval` ‘d to reload values.
>
> Consider using this for communicating variables across subshell boundaries.

`[ -n "${var+x}" ]`

If you need to check if a variable is defined:

Example

```
if [ -n "${var+x}" ]
then
  echo "The variable 'var' exists"
else
  echo "The variable 'var' has not been defined"
fi
```

Do not simply check if the variable is empty (*unless that’s what you are indending*):

Example

```
[ -z "$var" ] # Checks if 'var' is a zero length / empty string
              # but 'var' may be a defined variable
```

> 💡 **Tip:** It’s usually fine to simply check if a variable is `-n` zero-length or `-z` zero length, this code is much more understandable than `"${var+x}"` .
>
> Just make *sure* you are *intentional*.
>
> - Check for blank when that’s when you intend (`-z` `-n`)
> - Check for variable exists when that’s what you intend (`"${var+x}"`)

ℹ️ For more info on `[ -n "${var+x}" ]` here's a StackOverflow post (https://stackoverflow.com/questions/3601515/how-to-check-if-a-variable-is-set-in-bash) explaining it.

Note: I have run into problems using this without the `"` double quotes so I highly recommend using them!

💬 Strings

`cmd` or `"value"`

If text *represents* a “value”, use `"double quotes"`

If text *represents* a command-line argument, use `no quotes`

Example

```
dogs setName "Rover"

# In the above example:
# - 'setName' represents a command
# - 'Rover' represents a value

# There's no *technical* reason for the above not to be written as:
dogs setName Rover
dogs "setName" Rover
"dogs" "setName" Rover
"dogs" "setName" "Rover"
# ^ These are all technically equivalent
```

> If you have a *real actual* reason for serious performance in a library:
>
> - use `no quotes` when possible
> - else `'single quotes'` to avoid variable interpolation
> - use `"double quotes"` only when interpolation is required

`grep` & `sed` & `awk`

When possible, use built-in Bash string manipulation and pattern matching over `grep`, `sed`, `awk`, et al. However, keep your code’s maintainability in mind and use these tools when it results in simpler code.

`tl;dr`

- Do not “blindly” reach for familiar tools such as `grep` and `sed` when Bash functionality would work just as well, if not better.

String Matching

For simple values, prefer Bash string matching over `grep` .

String Contains Example

```
var="Hello World"

# Goal: Determine if the value contains the text 'World'

# ❌ Don't do this
if echo "$var" | grep World; then # ...

# ✅ Do this
if [[ "$var" = *"World"* ]]; then # ...
```

String Matches Pattern Example

```
var=""

# Goal: Determine is the value starts with 'Hello'

# ❌ Don't do this
if echo "$var" | grep ^Hello; then # ...

# ✅ Do this
if [[ "$var" =~ ^Hello ]]; then # ...
```

String Manipulation

Prefer Bash string manipulation over `sed` .

String Replacement Example

```
var="Hello World"

# Goal: Replace 'Hello' with 'HELLO'

# ❌ Don't do this
var="$( echo "$var" | sed 's/Hello/HELLO/' )"

# ✅ Do this
var="${var/Hello/HELLO}"
```

String Extraction

Prefer Bash string manipulation over `awk` *depending on the need for performance.*

String Extraction Example

```
var="Hello World Goodnight Moon"

# Goal: Get the second space-delimited value

# ❌ Don't do this
var="$( echo "$var" | awk '{print $2}' )"

# ✅ Do this
var="${var*# }"
var="${var%% *}"
```

> ☝️ Caveat: whereas `'{print $2}'` is easy to understand, the following is not:
>
> ```
> var="${var*# }"
> var="${var%% *}"
> ```
>
> Keep in mind the performance requirements for your program and choose what is right for you.

`${cheat%%\*sheet}`

Here (https://tldp.org/LDP/abs/html/string-manipulation.html) is a useful reference for Bash string manipulation.

It’s really easy once you get the hang it it!

Just remember:

- `#` is the left
- `%` is the right

Substring Removal

> It is *very* common to want to remove a *part* of a string:

	Description	e.g. `.foo .foo .foo`
`/`	Remove first match	`${x/foo}` ➤ `. .foo .foo`
`//`	Remove all matches	`${x//foo}` ➤ `. . .`
`#`	Remove shortest match (from the left)	`${x#*.f}` ➤ `oo .foo .foo`
`##`	Remove longest match (from the left)	`${x##*.f}` ➤ `oo`
`%`	Remove shortest match (from the right)	`${x%oo*}` ➤ `.foo .foo .f`
`%%`	Remove longest match (from the right)	`${x%%oo*}` ➤ `.f`

> 💡 **Tip:** When using `#` and `%` you'll usually want to accompany it with `*`

Substring Replacement

	Description	e.g. `.foo .foo .foo`
`/`	Replace first match	`${x/foo/bar}` ➤ `.bar .foo .fooo`
`//`	Replace all matches	`${x//foo/bar}` ➤ `.bar .bar .bar`
`/#`	Replace match if at start of string	`${x/#foo/bar}` ➤ `.foo .foo .foo`
`/%`	Replace match if at end of string	`${x/%foo/bar}` ➤ `.foo .foo .bar`
`:`	Substring to right of provided index	`${x:3}` ➤ `o .foo .foo`
`::`	Substring to right of provided index of provided length	`${x:3:5}` ➤ `o .fo`

> 💡 **Related:** To get the length of a string: `${#varname}`

`shopt -s extglob`

Sometimes you need some more modern expressions in your replacements.

Here (https://www.linuxjournal.com/content/bash-extended-globbing) is a good reference of what `extglob` provides:

	Description from the bash man page
?(pattern-list)	Matches zero or one occurrence of the given patterns
*(pattern-list)	Matches zero or more occurrences of the given patterns
+(pattern-list)	Matches one or more occurrences of the given patterns
@(pattern-list)	Matches one of the given patterns
!(pattern-list)	Matches anything except one of the given patterns

The most common need for `extglob` is to match a series of repeating characters

- e.g. `+([\d])` for multiple digits

You have to enable `extglob` to use the extended pattern matching: `shopt -s extglob`

> 💡 **Tip:** If you want to be *kind* and disable `extglob` after using it (*unless it was already enabled*):
>
> ```
> # Check if extglob is enabled
> if shopt -q extglob
> then
>   # it was already enabled, go ahead and do your pattern matches
> else
>   shopt -s extglob # turn it on
>   # go ahead and do your pattern matches
>   shopt -u extglob # turn it back off
> fi
> ```

🗃 Arrays

`declare -a`

The Bash array is the most powerful tool in anyone's Bash arsenal.

It’s a very simple single-dimensional array of text values.

Because Bash `3.2.57` doesn’t support Associative Arrays (see Mac Support), this is the primary data structure upon which all Bash libraries and applications are built!

> 💕 Learn to love the single-dimensional Bash array

To declare a new array, use `declare -a` (see example above)

> 💡 **Reminder:** `declare -a` assigns the array as a `local` variable in functions.

For recommendations on storing complex data, see Representing Objects below.

IFS=$'\n'

To load an array with items separated by newlines:

Example

```
# Create an array
declare -a items=()

# Run 'ls' and put each result into ar array
IFS=$'\n' read -d '' -ra items < <(ls)

# Read each line of a file into an array
IFS=$'\n' read -d '' -ra items < myFile
```

Alternatively, you may want a string separated by a character such as `:`

Example

```
# Create an array
declare -a items=()

# :-delimited string
textItems="foo:hello world:bar"

# Read the items into an array
IFS=: read -d '' -ra items < <(printf "$textItems")

echo "${#items[@]}"
# => 3

echo "${items[*]}"
# => "foo hello world bar"

# Or read a literal string in directly
IFS=: read -d '' -ra items <<< <(printf "foo:hello world:bar")

# ☝ Gotcha: if you do this, there will be a trailing newline in the 'bar\n' item
IFS=: read -d '' -ra items <<< "foo:hello world:bar"
```

`find -print0`

Related to `IFS`, to load an array with items from the `find` command:

Example

```
# Create an array
declare -a items=()

local filePath
while IFS= read -rd '' filePath
do
  items+=("$filePath")
done < <(find . -name "*.sh" -print0)
```

`declare -A`

I have nothing to say about Bash Associative Arrays 🤷‍♀️

I almost never use them because I try to natively support Bash `3.2.57`.

They're great, have fun with them!

> See 🍏 Mac Support

🏃‍♀️ Functions

`local`

It is so super critical that every variable you assign in a function be `local` .

Example

```
myFunction() {
  local varOne # <-- ✅ ALWAYS define variables as local
  varTwo=2     # <-- ❌ NEVER set globals unless you intend to
  declare -a varThree=() # declare defines vars as 'local' by default
}
```

> ℹ️ When perfoming `for` loops, the variable name will become assigned. This will be global unless you define it as `local` before your `for` loop.
>
> ```
> myFunction() {
>   local arg # <-- define your 'for' loop variables as local
>   for arg in "$@"
>   do
>     : # do something
>   done
> }
>
> myFunction "hello" "world"
> echo "$arg"
> # => "" # <-- if you don't use `local arg`, this will be "world"
> ```

`return`

Bash uses implicit returns, meaning the return code will be the `$?` return code of the last command run in your function - unless you explicitly `return` .

Recommend you check for error cases, e.g. wrong number of arguments or invalid arguments, and explicitly `return 1`.

Do not explicitly `return 0` unless you are sure the previous commands did not fail (*or if you do not care*).

Example

```
## # `myFunction`
##
## |      | Parameters |
## |------|------------|
## | `$1` | The one and only argument this function expects |
##
myFunction() {
  [ $# -ne 0 ] && { echo "Wrong number of arguments" >&2; return 1; }
  # do things
}
```

> 💡 **Tip:** Always write tests for the return values of your functions.

`declare -f`

If you need to view the source code of a function: `declare -f functionName`

```
myFunction() { echo Hello; }

declare -f myFunction
# myFunction ()
# {
#     echo Hello
# }
```

> 💡 **Tip:** If you need to copy a function, you can get the source code from `declare -f functionName`, replace the name at the start of the source code, and `eval` the source code.

`out` Function Variables (Return Values)

In most modern programming languages, it is common to invoke a method to get and use some kind of a **return value**.

Bash functions *do not have return values*, only status codes (e.g. `return 1`)

The most common way to use a Bash function to get a value is:

- Create a function which *returns its value* by printing it:

  ```
  myFunction() {
    printf "This is the return value"
  }
  ```

- Call that function in a subshell and use its output as the *"return value"*:

  ```
  local returnValue="$( myFunction )"
  echo "$returnValue"
  # => "This is the return value"
  ```

There are 2 main problems with this approach:

- **Performance:** this creates a new subshell (*which is really not necessary*)
- **Scope & Bugs:** the subshell *CANNOT MODIFY ANY GLOBAL VARIABLES*

  > *This has a lovely tendency to create bugs!*

> ℹ **Note:** the are often great reasons to run functions in subshells!

The best pattern for getting *return values* from functions is by using `out` variables.

Solution: use `out` variables

C# is an example of a language which supports `out` variables. The method can modify the value of a provided parameter (*the parameter is updated in its original scope*).

We can reproduce the same using Bash:

Example (Bash `out`-style variables)

```
main() {
  local name
  getName name
  echo "The return value is: $name"
}

# This returns the name of something.
# The first argument is the name of the 'out' variable.
getName() {
  local outVariableName="$1"
  local theReturnValue="Rover"

  # Assign the value to the provided variable name
  printf -v "$outVariableName" "$theReturnValue"
}

main
# => "The return value is: Rover"
```

> 💡 **Tip:** `printf -v $variableName` will not output to STDOUT, instead it will assign the value to the provided variable.

Note: `getName()` in this example never *prints* any value. Which isn't super useful.

Recommendation: Your functions should print "return values" *unless* an `out` variable is provided.

```
# Expects either zero arguments (in which case it will print)
#             or one argument (in which case it will assign)
getName() {
  local theReturnValue="Rover"
  if [ -z "$1" ]
  then
    printf "$theReturnValue"
  else
    printf -v "$outVariableName" "$theReturnValue"
  fi
}
```

Multiple Return Values

You can use the same pattern to return:

- Multiple return values
- Populate a provided array

Here is a sample that demonstrates both:

Example (*multiple `out` return values*)

```
# Sample data
DOG_NAMES=(Rover Spot Rex)
DOG_BREEDS=("Golden Retriever" "Pomeranian" "Daschund")
DOG_TOYS=("Bone:Squeeky Toy" "Bone" "Kong:Sqeeky Toy")

##
# This sample shows (a) multiple return values
#                   (b) conditionally printing -vs- assigning variables
#
# You might want to have this in two separate functions, e.g.
# - printDogInfo
# - loadDogInfo
##

# Print or get the information about a dog given its number (index)
#
# $1 - The dog index
# $@ - (Optional) `out` variable names
#
getDogInfo() {
  local __dogInfo__index="$1"; shift
  if [ $# -eq 0 ]
  then
    echo "Name: ${DOG_NAMES[__dogInfo__index]}"
    echo "Breed: ${DOG_BREEDS[__dogInfo__index]}"
    echo "Favorite Toys: ${DOG_TOYS[__dogInfo__index]//:/ }"
  else
    printf -v "$1" "${DOG_NAMES[__dogInfo__index]}"
    printf -v "$2" "${DOG_BREEDS[__dogInfo__index]}"
    IFS=: read -d '' -ra "$3" < <(printf "${DOG_TOYS[__dogInfo__index]}")
  fi
}

main() {
  local dogName
  local dogBreed
  declare -a dogToys

  # Call a function providing regular parameter
  # in addition to the names of multiple variables
  # to get as return values
  getDogInfo 0 dogName dogBreed dogToys
  echo "$dogName is a $dogBreed and loves their toys: ${dogToys[*]}"

  # Call getDogInfo normally without --out arguments
  getDogInfo
}

main

# Rover is a Golden Retriever and loves their toys: Bone Squeeky Toy

# Name: Rover
# Breed: Golden Retriever
# Favorite Toys: Bone Squeeky Toy
```

Specifying `out` Variable Names

In the examples this far, `out` variables have been specified as *optional* additional command line arguments.

The functions have conditionally assigned to those variables if they were provided.

Recommendations:

- Use separate functions named `printFoo` and `loadFoo`
- -or-
- Use the pattern shown above (*optional additional arguments*)

  > ^ *This has worked very well for me in my libraries!*

💻 Commands

`main()` function

In all of your BASH scripts, start using a `main()` function.

❌ Don’t Do This

```
# [myScript.sh]

myVar=5
myArray=()

for item in "$@"
do
  echo "the item: $item"
done
```

✅ Do This

```
# [myScript.sh]

printTheValues() {
  local myVar=5
  local myArray=()
  local item
  for item in "$@"
  do
    echo "the item: $item"
  done
}

main() { printTheValues "$@"; }

[ "${BASH_SOURCE[0]}" = "$0" ] && main "$@"
```

It's just a good habit to get into!

It's really nice because you get `local` variables.

And if (*or when*) your script begins to grow in scope and size:

- Using functions will help you organize it
- Your script will be easier to `source` and use from other files

$* or $@

Don't use these interchangably.

Explicitly use `$@` or `${array[@]}` when you intend to expand the value into multiple arguments.

Explicitly use `$*` or `${array[*]}` when you simply want to view or print all of the values in a single argument.

Programmers of other languages might want to think of `$@` as “splat” or “spread” params.

`${1:-default}`

To be honest, I don’t find these this useful.

But here is a nice reference for Bash Parameter Substitution (https://tldp.org/LDP/abs/html/parameter-substitution.html).

Programmers of other languages might want to think of this as `||=` or “or equals” operators.

If you want to set a parameter to a value only if it’s not already set:

Example (*parameter substitution*)

```
FOO=123

: "${FOO=456}" # <-- this won't set the value (already set)
: "${BAR=456}" # <-- this will set the value

echo "Foo: $FOO"
# "Foo: 123"

echo "Bar: $BAR"
# "Bar: 456"
```

Personally, I find this to be more readable:

Example (*check if variable is empty*)

```
FOO=123

[ -z "$FOO" ] || FOO=456 # <-- this won't set the value (already set)
[ -z "$BAR" ] || BAR=456 # <-- this will set the value

echo "Foo: $FOO"
# "Foo: 123"

echo "Bar: $BAR"
# "Bar: 456"
```

> ☝ Gotcha: using `${var=default}` may help prevent bugs.
>
> `: ${var=default}` will only assign if `var` does not exist
>
> If `var` is set to a blank string, this *will not assign*.
>
> In my other example, we are using `[ -z "$var" ]` which explicitly checks *“is this string empty?”* and sets the value if it is blank. The value could already be explicitly set to `""` but the second example will override that value.
>
> The two are **not equivalent**.

`[ while $# -gt 0 ]`

If you simply need to loop through arguments provided. in order:

Example (*`for` loop thru arguments*)

```
main() {
  local arg
  for arg in "$@"
  do
    echo "The argument is: $arg"
  done
}
```

If you want to process arguments with complex syntax, it’s usually nice to:

Example (*`while` thru arguments*)

```
main() {
  while [ $# -gt 0 ]
  do
    # ... look at "$1" and shift through arguments as necessary
    shift
  done
}
```

Example (*`while` thru arguments with case/esac*)

```
# Very simple argument parsing, e.g. --fileName FILE --path PATH
main() {
  local fileName
  local filePath
  while [ $# -gt 0 ]
  do
    case "$1" in
      --fileName)
        shift
        fileName="$1"
        ;;
      --path)
        shift
        filePath="$1"
        ;;
      *)
        echo "Unexpected argument: $1" >&2
        return 1
        ;;
    esac
    shift
  done
}
```

`case ... esac`

Speaking of `case` / `esac` , it's really simple and powerful for structuring your commands *and subcommands* (*and their subcommands…*).

Most of my functions are structured in the following way:

Example (*case/esac for subcommands*)

```
myFunction() {
  local command="$1"; shift
  case "$command" in
    --help)
      cat docs/HELP.md
      ;;
    config)
      local configCommand="$1"; shift
      case "$configCommand" in
        list)
          ls configFiles/*.txt
          ;;
        set)
          echo "$1" > "configFiles/$1.txt"
          echo "Saved configuration $1"
          ;;
        *)
          echo "Unknown 'myFunction config' command: $2" >&2
          return 1
          ;;
      esac
      ;;
    *)
      echo "Unknown 'myFunction' command: $1" >&2
      return 1
      ;;
  esac
}
```

> 💡 **Tip:** You can store each “case” in a separate shell source file and use a script to merge them all into one `case` / `esac` tree.

`getopts`

Nada.

I haven’t used it yet.

It’s a classic! Go ahead and try it out!

Most of my argument parsing is not classic `-a foo --list` arguments.

`- <<< "Foo"`

Finally, don't forget `STDIN` !

If you want to accept standard input, the convention is to read from standard input when a `-` argument is provided.

Example (*read from `STDIN` when `-` argument*)

```
# Print a value passed as an argument
# (if the '-' argument is provided, read from STDIN instead)
printProvidedValue() {
  if [ "$1" = - ]
  then
    echo "The value from STDIN is: $(</dev/stdin)"
  else
    echo "The value from argument is $1"
  fi
}

printProvidedValue "Hello, world"
# => "The value from argument is Hello, world"

printProvidedValue - <<< "Goodnight, moon"
# => "The value from STDIN is: Goodnight, moon"

echo "Hello, goodbye" | printProvidedValue -
# => "The value from STDIN is: Hello, goodbye"
```

If you're using Bash 4+, then *can* check if `STDIN` is present and use that when present:

Example (*read from `STDIN` when present*)

```
# If data is present in STDIN, print that
# otherwise print the provided argument
printProvidedValue() {
  if read -t 0 -N 0 # <-- read -N requires BASH 4+
  then
    echo "The value from STDIN is: $(</dev/stdin)"
  else
    echo "The value from argument is $1"
  fi
}

printProvidedValue "Hello, world"
# => "The value from argument is Hello, world"

printProvidedValue - <<< "Goodnight, moon"
# => "The value from STDIN is: Goodnight, moon"

echo "Hello, goodbye" | printProvidedValue -
# => "The value from STDIN is: Hello, goodbye"
```

🐚 Subshells

- Subshells are your friend - when you want them to be.
- Subshells are your enemy - when you least expect it.

Subshells are really lovely and lightweight (*thank you Bash!*)

I would still recommend avoiding them when writing library code.
If it’s unnecessary to “*shell out*”, then don’t do it.

Perhaps the very very very most important thing to remember about subshells is:

- They *can* read your global variables
- They *can* execute *functions*, not just binaries
- When shelling out to functions, they *can* read your `local` variables!
- They *can* **not** *modify* any variables
- Using `out` style variables (defined above) will *not* work

If you want to run a subshell and have it update a variable - no.

> 💡 **Tip:** if you really must pass complex state from a child subshell to the parent, you can use `( set -o posix; set ) | grep VARNAME` to serialize desired variables in the subshell and print out the results. The parent can `eval` the resulting code to inflate the deserialized objects (*including Bash arrays*).

`$(echo "Hello")`

When creating a subshell, always use the `$( ... )` syntax (*not the backtick syntax*)

`$(<myFile.txt)`

When you want the contents of a file, use `$(<file/path)`

This also works for `STDIN`: `echo "The STDIN is: $(</dev/stdin)"`

`$? code`

☝️ Gotcha: to get the `$?` exit code of a subshell, you most store the output of the program in a variable (*even if you don't intend to use it*)

Example (*get `$?` from subshell*)

```
: "$( ls this/dir/doesnt/exist &>/dev/null )"
echo "$?"
# => 0  <-- whaaaa??????? yep.

_="$( ls this/dir/doesnt/exist &>/dev/null )"
echo "$?"
# => 2  <--- this is correct, needed to store output in a variable
```

`STDOUT & STDERR`

If you want to store the `STDOUT` and `STDERR` of a subshell separately, you'll need to store one of them in a temporary file.

Example (*get `STDOUT` and `STDERR` separately from subshell*)

```
theFunction() {
  echo "Hello from STDOUT"
  echo "Hello from STDERR" >&2
}

main() {
  local stderrFile="$( mktemp )"
  local stdout="$( theFunction 2>"$stderrFile" )"
  local exitCode=$?
  local stderr="$(<"$stderrFile")"
  rm "$stderrFile"
  echo "Ran theFunction"
  echo "STDOUT: $stdout"
  echo "STDERR: $stderr"
  echo "Exit Status: $?"
}

main
# => "Ran theFunction"
# => "STDOUT: Hello from STDOUT"
# => "STDERR: Hello from STDERR"
# => "Exit Status: 0"
```

📐 Math

`$(( i + 1 ))`

Bash can perform simple integer math

Example (*simple Bash math*)

```
x=42

echo "$(( x + 8 ))"
# => 50

echo "$(( x - 2 ))"
# => 40

: $(( x++ ))
echo "$x"
# => 43

echo "$(( x / 10 ))"
# => 4

echo "$(( x % 10 ))"
# => 3
```

`bc -l`

If you need to perform division / need floating point numbers, use `bc`

Example (*bc math*)

```
x=42

bc -l <<< "$x / 10"
# => 4.20000000000000000000

# Or this style:
echo "$x / 10" | bc -l
# => 4.20000000000000000000

# To limit the precision:
bc -l <<< "scale=2; $x / 10"
# => 4.20
```

🐶 Representing Objects

> This section is less relevant to users of BASH 4+ which includes Associative Arrays

When coding against Bash `3.2.57`, you're "stuck" with single dimensional Bash arrays.

This section covers just the basics of some of my favorite patterns for storing complex data in Bash arrays.

> 💕 I love representing objects in various ways in Bash arrays

`name:1;age:2;`

> Index lookup fields.

If your object has a set of *known* properties, you can simply map each property to a particular index identifier and be on your way!

```
Dog.getName() { eval "printf \"\${$1[0]}\""; }
Dog.getBreed() { eval "printf \"\${$1[1]}\""; }
Dog.getAge() { eval "printf \"\${$1[2]}\""; }

rover=("Rover" "Golden Retriever" "2")
spot=("Spot" "Daschund" "4")

Dog.getBreed rover
# => "Golden Retriever"

Dog.getAge spot
# => 4
```

> ℹ️ `eval` is used in these samples for Bash `3.2.57` compatibility
>
> Using Bash 4.3+ the above functions could be written as:
>
> ```
> Dog.getName() {
>   local dogArray
>   typeset -n dogArray="$1" # <--- See 'typeset -n' section for more info
>   printf "${dogArray[0]}"
> }
> ```

But, if:

- Your object has an unknown set of properties
- You want to save object space and only assign indices for set properties

Then, I recommend creating your own Index Lookup Field

Index Lookup Fields

If your object has multiple properties *with simple names**, then you can very easily create a key/value index (*like that of an associative array*)

Example (*complete key/value store - for simple keys*)

```
# Creating an object creates array with empty field lookup.
# For simplicity in this example, the user provides a simple
# object identifier to which is used in the array variable name.
# Really you should control the identifier yourself, e.g. random number.
Object.create() { eval "OBJECTS_$1=(\"\")"; }

# Find out if the object contains the given key
# $1 - Object identifier
# $2 - Simple key
Object.hasKey() {
  eval "[[ \"\${OBJECTS_$1[0]}\" = *\";\$2\"* ]]"
}

# $1 - Object identifier
# $2 - Simple key
Object.get() {
  local valueIndex
  if Object.hasKey "$1" "$2"; then
    __Object.getValueIndex "$1" "$2" valueIndex
    eval "printf \"\${OBJECTS_$1[\$valueIndex]}\""
  fi
}

# $1 - Object identifier
# $2 - Simple key
# $3 - Value
Object.set() {
  local valueIndex
  if Object.hasKey "$1" "$2"; then
    __Object.getValueIndex "$1" "$2" valueIndex
    eval "OBJECTS_$1[\$valueIndex]=\"\$3\""
  else
    # Add this field to the index (using the size of the current array)
    eval "OBJECTS_$1[0]=\"\${OBJECTS_$1[0]};\$2:\${#OBJECTS_$1[@]}\""
    # Add the value
    eval "OBJECTS_$1+=(\"\$3\")"
  fi
}

# Take a look at the underlying BASH array!
# $1 - Object identifier
Object.dump() { ( set -o posix; set ) | grep "^OBJECTS_$1="; }
```

```
# @private
# Get the index of the value field for the given key, if present in the object
# $1 - Object identifier
# $2 - Simple key
# $3 - `out` index value
__Object.getValueIndex() {
  if Object.hasKey "$1" "$2"; then
    local indexLookupField
    eval "indexLookupField=\"\${OBJECTS_$1[0]}\""
    indexLookupField="${indexLookupField#*;${2}:}" # Remove everything to the left of the index
    printf -v "$3" "${indexLookupField%%;*}" # Remove everything to the right of the index and return
  fi
}
```

And now, try it out:

```
Object.create rover
Object.get rover breed
# => ""
Object.set rover breed "Golden Retriever"
Object.get rover breed
# => "Golden Retriever"
Object.create spot

Object.set spot breed "Golden Retriever"
Object.set spot name "Spot"
Object.dump spot
# => OBJECTS_spot=([0]=";breed:1;name:2" [1]="Golden Retriever" [2]="Spot")
```

We basically just made our own Associative Array which works for *simple keys* (*that do not contain some separator we define*).

> ℹ️ Our simple key/value store also has a *nearly* `O(1)` constant time lookup.
>
> - As the number of keys + length of their characters increases, the Bash string manipulation to extract the index for each key will become slower. But it is not `O(N)` , there is no looping, the quick string manipulation gets us a *pretty good* data structure!

/dev/urandom

Rather than having users provide their own simple key (*which we used as part of the variable name*), you can control the Bash variable name yourself and hand each user an object identifier.

This example uses `/dev/urandom` to get a random string to act as an object identifier:

Example (*using randomly generated Object IDs*)

```
# $1 - `out` variable name to store object identifier
Object.create() {
  local objectId="$( cat /dev/urandom | base64 | tr -dc 'a-zA-Z0-9' | fold -w 32 | head -n 1
)"
  eval "OBJECTS_$objectId=(\"\")"
  printf -v "$1" "$objectId"
}

# ...
```

Now try the same example above but using the new `Object.create`:

```
main() {
  local rover
  Object.create rover
  Object.set $rover breed "Golden Retriever"
  Object.dump $rover
  # => OBJECTS_PnIIwehw9DGvuwgXCK8ecWF309M3RyU5=([0]=";breed:1" [1]="Golden Retriever")

  local spot
  Object.create spot
  Object.set $spot name "Spot"
  Object.set $spot breed "Daschund"
  Object.dump $spot
  # => OBJECTS_HXBPI288osbbhGlYnA7gWqgH1Dur38lV=([0]=";name:1;breed:2" [1]="Spot"
[2]="Daschund")
}

main
```

📦 Defining Blocks

`cmd { ... }`

Sometimes your commands will need to store commands to *run later*.

When storing commands, you MUST store them as arrays of arguments (*like* `$@`)

❌ Don't Do This

```
commandToRunLater="ls \"some/dir\" -l"
```

✅ Do This

```
commandToRunLater=("ls" "some/dir" "-l")
```

But usually users will want to provide these commands to you in an elegant way.

`{ local command }` and `{{ subshell }}`

My favorite convention is:

- Allow users to provide commands using the syntax: `{ cmd *args }`
- `{{ cmd *args }}` double curly braces implies the command should be run in a subshell

It's pretty easy to support!

```
# $1 - A name identifier for this command to run later
# $@ - { ... } arguments
saveCommandForLater() {
  local commandName="$1"; shift
  local runInSubshell
  if [ "$1" = "{" ] || [ "$1" = "{{" ] # block representing a command!
  then
    [ "$1" = "{{" ] && runInSubshell=true
    eval "COMMAND_$commandName=(\"$runInSubshell\")"
    shift
    while [ $# -gt 0 ]
    do
      [ "$1" = "}" ] || [ "$1" = "}}" ] && return 0
      eval "COMMAND_$commandName+=(\"\$1\")"
      shift
    done
  else
    echo "Expected saveCommandForLater with a { ... } block" >&2
    return 1
  fi
  echo "Missing } or }} closing braces for saveCommandForLater command" >&2
  return 1
}

# $1 - A name identifier for this command to run later
runCommandFromEarlier() {
  local runInSubshell
  eval "runInSubshell=\"\${COMMAND_$1[0]}\""
  if [ -n "$runInSubshell" ]
  then
    local _
    eval "_=\$( \"\${COMMAND_$1[@]:1}\" )"
  else
    eval "\${COMMAND_$1[@]:1}"
  fi
}
```

And give it a try:

```
saveCommandForLater hello { echo "Hello, world!" }
runCommandFromEarlier hello
# => "Hello, world!"

# Now confirm that {{ ... }} runs in a subshell and can't modify variables
x=42
saveCommandForLater subshellSetVariable {{ eval "x=5" }}
saveCommandForLater localSetVariable { eval "x=5" }

runCommandFromEarlier subshellSetVariable
echo "$x"
# => 42

runCommandFromEarlier localSetVariable
echo "$x"
# => 5
```

`do ... end`

Another way to extend your Bash library or framework's syntax is providing your own `do/end` blocks.

Bash has a `done` keyword (*similar to `fi` and `esac`*) for closing blocks.

There is no `end` keyword, so I like using this for my own syntax.

To do this with my libraries, I have what I call an `END_STACK`.

Here is an example:

```
describe "Group of tests" do
  example "my test" do
    : # some predefined DSL commands
  end
  example "different test" do
    : # some predefined DSL commands
  end
end
```

Below, let's put this into action:

Example

```
# --- some fake library code
describe() { END_STACK+=("My Library:Describe Block:$1"); }
example() { END_STACK+=("My Library:Test Block:$1"); }
# ---

END_STACK=()
end() { [ "${#END_STACK[@]}" -gt 0  ] && unset END_STACK["$(( ${#END_STACK[@]} - 1 ))"]; }

printEndStack() { ( set -o posix; set ) | grep ^END_STACK=; }

printEndStack

describe "Group of tests" do
  printEndStack

  example "my test" do
    : # some predefined DSL commands
    printEndStack
  end

  printEndStack

  example "different test" do
    : # some predefined DSL commands
    printEndStack
  end

end

printEndStack
```

Outputs:

```
END_STACK=()
END_STACK=([0]="My Library:Describe Block:Group of tests")
END_STACK=([0]="My Library:Describe Block:Group of tests" [1]="My Library:Test Block:my
test")
END_STACK=([0]="My Library:Describe Block:Group of tests")
END_STACK=([0]="My Library:Describe Block:Group of tests" [1]="My Library:Test
Block:different test")
END_STACK=()
```

Any commands running can *check if they are in a current END_STACK scope* and perform actions accordingly, knowing that they are in that scope.

Feel free to play with this pattern for your own DSLs!

🔬 Testing

`it.needs_tests()`

Your code needs tests.

That is it.

No excuses.

If you're writing a library or framework, take a look at the available testing frameworks for Bash and shell scripts. Pick one, learn it well, and write a robust test suite.

📖 Documentation

Your code needs documentation.

That is it.

No excuses.

> ℹ️ **Note:** I haven't found any existing tools that I like for generating documentation.

My Function

Recommendation: Document your functions and doce with Markdown (*and extract it into a website*)

I write my functions like this:

Example (*Markdown commented function*)

```
## # `myFunction`
##
## It does something
##
## #### Example
##
## ```sh
## myFunction cool things
## ```
##
## | | Parameter |
## |-|-----------|
## | `$1` | Description of the first parameter |
## | `$2` | Description of the second parameter |
## | `$@` | Description of splat of additional arguments |
##
## - Returns `1` if the function something doesn't not exist |
##
myFunction() {
  :
}
```

I copy/paste the template between my functions.

It's worth it.

Then I `grep` for ALL `##` comments across my files and put them into one or more Markdown files.

> 💡 **Tip:** Host your Markdown files with something like GitHub Pages!

`>> "$apiDocs.md"`

Let's say that my project tree has a `src/` folder with 10 subfolders.

If I want to create 1 markdown file for each of those 10 sections of the codebase:

Example (*generate Markdown files from comments*)

```
generateDocs() {
  local dir
  for dir in src/*
  do
    local folderName="${dir%%*/}"
    grep -rl "^[[:space:]]*##" "$dir"   | \
    sort --version-sort                  | \
    xargs -n1 grep -h "^[[:space:]]*##" | \
    sed 's/^[[:space:]]*//'              | \
    sed 's/^##[[:space:]]\?//' >> docs/$folderName.md
  done
}

generateDocs
```

You'll end up with `docs/dogs.md` and `docs/cats.md` etc 📚

Add other scripts to add headers / footers etc to make your docs your own 💕

🍏 Mac support

💡 **Recommendation:** Support Mac out-of-the-box, do not use newer Bash features.

> Many users of Bash use Mac and the easiest solution to supporting them is to author your Bash code to support `3.2.57` out of the box.
>
> (Optional) If so desired, create a branch of your code which branches on `$Bash_VERSION` and uses an alternate, optimized version of your program which supports more modern Bash features such as associative arrays and indirect variable references.
>
> See these alternate solutions to Bash 4.3+ features:
>
> - Indirect Variable References
> - Associative Arrays

3.2.57(1)-release

Mac ships with a version of Bash which was released in 2002 (https://en.wikipedia.org/wiki/Bash_(Unix_shell)#Release_history): Bash 3.2.57(1)-release.

Why?

This is the last version of Bash which was shipped under the GPLv2 (https://en.wikipedia.org/wiki/GNU_General_Public_License#Version_2) license, future versions switched to GPLv3 (https://en.wikipedia.org/wiki/GNU_General_Public_License#Version_3).

Apple decided to stick with the GPLv2 (https://en.wikipedia.org/wiki/GNU_General_Public_License#Version_2) version, even though it is 15 years old at the time of writing.

`zsh`

> ℹ Starting with macOS Catalina, Macs now use `zsh` (https://en.wikipedia.org/wiki/Z_shell) as the default shell.
>
> *This document is not relevant to other shells, e.g. `zsh`*

`BASH 4` + `BASH 5`

Bash 4 and 5 added useful features for developers and script authors, e.g.

- Associative Arrays (*string key/value pairs*)
- Indirect Variable References (*reference a variable using a dynamic variable name*)

If you want your Bash script to support Mac, you will not be able to use these features.

`$variableName`

If your Bash scripts use variables with dynamic names, you will need to use `eval`.

In Bash 4.3 (*unsupported on Mac*), you can use `typeset -n` to get an indirect reference to a variable by name and modify that variable.

```
hello="World"

# Modify the value of hello using a variable which contains the variable name "hello"
variableName=hello
typeset -n theVariable="$variableName"

echo "$theVariable"
# => "World"

theVariable="change me"

echo "$theVariable"
# => "change me"

echo "$hello"
# => "change me"
```

Bash 3.2.57 required `eval` to work with variables with dynamic names.

```
hello="Hello, world!"

variableName=hello
eval "echo \"$variableName\""
# => "Hello, world!"
```

Docker

If you are not developing on a Mac, it is recommended that you test your application against Mac's `3.2.57` version of Bash using Docker and/or configure your cloud test provider to run your tests on a Mac device.

Example

Create a `Dockerfile` :

```
FROM bash:3.2.57
```

Build the image:

```
docker build -t bash3257 .
# Sending build context to Docker daemon  972.3kB
# Step 1/1 : FROM bash:3.2.57
#  ---> 4d4010b2347d
# Successfully built 4d4010b2347d
# Successfully tagged bash3257:latest
```

Run a Bash script in the local folder by mounting the folder and running it in a container via `bash` :

```
# [foo.sh]
echo "hi from $BASH_VERSION"
```

```
$ docker run --rm -it -v "$PWD:/scripts" bash3257 bash scripts/foo.sh
# hi from 3.2.57(1)-release
```