Lecture 5: Command substitution. Input/Output (I/O). Conditional statements

Last update: 20230506

Table of Contents

1. Command substitution: $(...)

We have already seen that a value can be stored in a variable by explicit assignment (using the operator `=`), or if the user supplies variables as command line arguments (positional parameters) to a script or a function. In practice, however, one frequently wants to store the output of some command directly into the variable, or even the content of an external file. This can be achieved with the so-called *command substitution operator* `$( ... )`. For instance, we have already seen that the file size in bytes can be printed with the following:

```
stat -c %s someFile
```

But how can we fetch the printout of above command programmatically, and do some manipulation with it later in our code? This is precisely the case when we need to use the command substitution operator:

```
FileSize=$(stat -c %s someFile)
```

Now the size of file 'someFile' is stored directly in the variable **FileSize** and from this point onwards we can obtain content of that variable in the same way as the content of any other variable:

```
echo ${FileSize}
```

The operator `$( ... )` can do much more than that. For instance, it can literally in-line the output of any command at the place where this operator is used.

Example 1: How to produce the following single-line output, with the current timestamp embedded:

```
Today is Mo 20. Mai 15:33:07 CEST 2019 . What a nice day...
```

This can be achieved with:

```
echo "Today is $(date) . What a nice day..."
```

The command substitution operator literally in-lined the output of **date** command at the place where it was used. This way, we can very elegantly achieve the desired more complex functionality by combining in the very same command input multiple commands, which otherwise we would need to execute one-by-one.

Command substitution operator `$( ... )` is a very neat construct, and it is used frequently. One classical use case is to avoid hardwiring any specific information in your code, since that specification can change from one computer to another. In this way, we can improve a lot the portability of code.

Example 2: You are working in parallel on two computers, which do not have the same version of the command that you use in your code. You would like to use if possible all the latest functionalities of that command, but if that is not available, you would still like to run your code with the older version of that command. Can you make the code transparent to such a difference? You can do it schematically as follows:

```
Version=$(commandName -v) # flag '-v' typically prints the command version
[[ $Version -lt someTreshold ]] && use-older-functionalities
[[ $Version -ge someTreshold ]] && use-newer-functionalities
```

This is just a schematic solution — most likely the output of **commandName -v** will have some additional information that you need to filter out, but all that can be still done within the command substitution operator.

You can fearlessly nest the command substitution operators, like in the following example.

Example 3: How can you get programmatically only the name of the parent directory of the directory in which your script sits (knowing that the environment variable **PWD** holds the full absolute path of script's directory)?

To solve this problem, we need first to introduce two widely used **Linux** commands in this context: **basename** and **dirname**. The command **basename** is typically used in the following way: It takes as an argument the absolute path to some directory or file, and drops the part which corresponds to an absolute path. This is illustrated with the following code snippets:

```
$ DirectoryPath=/home/abilandz/Lecture/PH8124/Lecture_5
$ basename ${DirectoryPath}
Lecture_5 # only the directory name is printed
```

On the other hand, the command **dirname** does the opposite: It prints only the absolute path to the specified directory or file. If we reuse the above example:

```
$ DirectoryPath=/home/abilandz/Lecture/PH8124/Lecture_5
$ dirname ${DirectoryPath}
/home/abilandz/Lecture/PH8124 # only the abs. path is printed
```

The commands **basename** and **dirname** can be used in exactly the same way for files.

Therefore, the solution to our initial problem can be fairly elegant and concise, if we use these two commands in combination with command substitution operator:

```
$ DirectoryPath=/home/abilandz/Lecture/PH8124/Lecture_5
$ ParentDirectoryName=$(basename $(dirname $DirectoryPath))
$ echo $ParentDirectoryName
PH8124 # only the parent directory name is printed
```

We can use multiple commands within the same command substitution operator, they just need to be separated with delimiter `;` as in the following example:

```
Var=$(date;pwd)
echo "$Var"
```

The printout is

```
Thu May 14 11:58:28 CEST 2020
/home/abilandz/Lecture
```

It is perfectly fine to inline the output of function call with this operator as well:

```
echo "Output of my function is: $(someFunction) . Very nice!"
```

or to store the printout of a function in the variable:

```
Var=$(someFunction)
```

Finally, the very neat use case of the command substitution operator is to store the content of some external file in the variable. The relevant syntax is:

```
FileContent=$(< someFile)
```

In the above example, `<` is just a shortcut for the command **cat**, which can be used equivalently in this context:

```
FileContent=$(cat someFile)
```

This great functionality circumvents the necessity of dealing with too many temporary files during the code execution, when we are interested to keep the file content only at a particular time. With the above definitions, the following two commands yield exactly the same answer initially:

```
cat someFile # reads the content of a physical file
echo "${FileContent}" # obtain the same content from variable
```

However, if the content of the physical file `someFile` has changed or if it was deleted, that does not affect the value of variable **FileContent**. This is very handy when we need to initialize our script or function with the content of some external file — if we store that information in the variable, we have removed completely the dependency of our code on that external file.

The command substitution operator is frequently used in combination with the **for** loop, when we want to iterate over all elements in the output of some command. Also in this context the distinct elements of the list are separated with one or more empty characters. This is best illustrated with the following example:

Example 4: How can we loop over all files in the current directory and print the size of each file?

One simple solution (works only if filenames do not contain empty characters!) is provided with the following code snippet:

```
for File in $(ls $PWD); do
 [[ -f $File ]] && Size=$(stat -c %s $File) || continue
 echo "The size of ${File} is: ${Size}"
done
```

Note that if you would have used the lengthy output of **ls** by specifying the flag **-l**, then the loop variable **File** would loop over all entries in the command output separated with one or more empty characters, therefore also over the permissions field, user name, etc. This is illustrated in the following example:

```
for Var in $(date); do
 echo "Var = $Var"
done
```

The output is:

```
Var = Thu
Var = May
Var = 14
Var = 13:13:50
Var = CEST
Var = 2020
```

This was yet another example to illustrate the importance of empty character as being the default field separator in **Linux/Bash**.

By definition, the command substitution operator `$( ... )` takes only stdout stream — in rare cases when both 'stdout' and 'stderr' streams, or only 'stderr' stream, need to be taken, the following generic syntax can be used, respectively:

```
# both 'sdtout' and 'stderr' of 'someCommandInput' are stored in Var:
Var=$(someCommandInput 2>&1)

# only 'stderr' of 'someCommandInput' is stored in Var:
Var=$(someCommandInput 2>&1 1>/dev/null)
```

In the end, we would like to remark that the backticks `` ` ... ` `` do the same thing as command substitution operator `$( ... )`:

```
echo "Today is: $(date) . Thanks for the info."
echo "Today is: `date` . Thanks for the info."
```

Bash supports backticks in this context only for backward compatibility with some very old shells. There is, however, one important difference: Nesting of backticks `` ` ... ` `` does not work properly, only the nesting of command substitution operator `$( ... )` is reliable. That being said, `$( ... )` shall be preferred in **Bash** scripts over backticks `` ` ... ` ``.

2. Input/Output (I/O) and redirections

In the previous section we have seen how we can embed the output of one command into the input of another command with the command substitution operator `$( ... )`. Let us now make a further progress in this direction and clarify in more detail the input and output streams of **Linux** commands. By convention, each **Linux** command has three standard input/output (I/O) channels set. More concretely, each **Linux** command has a single way of:

- accepting input : **standard input (*stdin*)** = file descriptor 0
- producing output : **standard output (*stdout*)** = file descriptor 1
- producing error messages : **standard error (*stderr*)** = file descriptor 2

Each command that you execute has these three standard I/O channels set to some default values. By default, standard input is a keyboard (but it can be also a file redirection, touchscreen, etc.). On the other hand, standard output and standard error are by default set to screen. The most important things to remember is:

- *stdout* (file descriptor 1): This is the textual stream you see in the terminal if a command executed successfully;
- *stderr* (file descriptor 2): This is the textual stream you see in the terminal if a command failed (a.k.a. error message).

For instance, when the command **date** executes successfully, it produces the following:

```
$ date
Sun May 17 11:53:03 CEST 2020
```

The above printout is an example *stdout* stream of command **date**. On the other hand, when the command **date** fails, for instance when it is called with the flag which is not supported:

```
date -q
```

it will print the error message:

```
date: invalid option -- 'q'
```

The above printout is an example *stderr* stream of command **date**. This behavior is true for basically all **Linux** commands.

Since the two streams, *stdout* and *stderr*, are always set for a command, we will now see how to handle them programmatically. In practice, one can programmatically fetch the *stdout* of some command, parse through it, and depending on its content, issue some specific action. In a similar fashion, one can fetch programmatically *stderr* (i.e. error message) of some command, and depending on its content, issue some specific action to fix that particular problem. For that sake, we need to use their respective file descriptors. The following operators are available in **Bash** to handle *stdout* and *stderr* streams:

- `1>` : captures and redirects to a file only the successful output of command (*stdout*)

- `2>` : captures and redirects to a file only the error message if command failed (*stderr*)
- `&>` : captures and redirects to the same file both the successful output (*stdout*) and the error message (*stderr*)

For instance, if we want to redirect the *stdout* stream of **date** command into a file, we would use:

```
date 1> output.log
```

Whatever the command **date** was printing on the terminal, now is re-directed to the physical file named `output.log`. If that file does not exist, it will be automatically created at this point. The file's location in the file system can be specified also in this context both with an absolute and a relative path. If you now execute:

```
cat output.log
```

you get back the output of **date** command:

```
Sun May 17 11:53:03 CEST 2020
```

In this sense, by using `1>` redirection, the printout of some command during execution is stored permanently in the physical file on a local disk.

Analogously, we can also programmatically redirect the error message of command, we just need to change the file descriptor:

```
date -q 2> error.log
```

It is perfectly feasible to combine both examples on the same line:

```
someCommand 1> output.log 2> error.log
```

We can also redirect both *stdout* and *stderr* in the same file with `&>` operator:

```
someCommand &> outputAndError.log
```

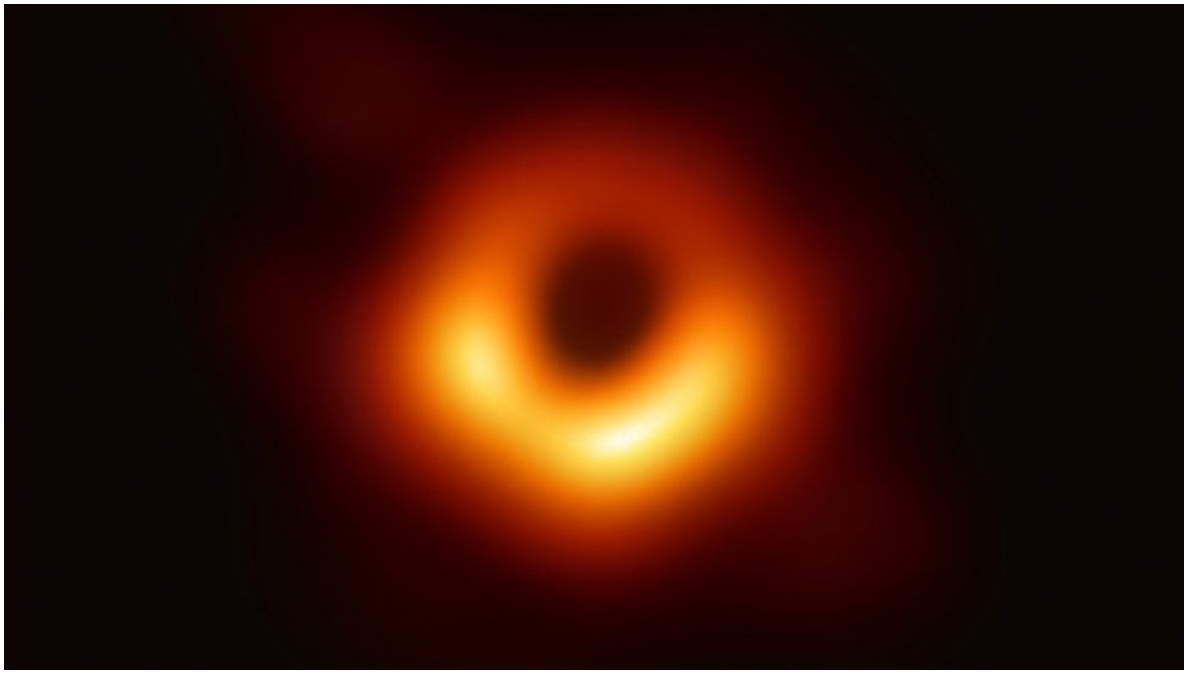
This way we can keep the whole printout which command has produced during execution permanently in some local files, separately for *stdout* and *stderr*, or combined . Then later at any point by inspecting those printouts in the files we can trace back the whole execution, which helps enormously the code development and debugging.

If we re-execute the above examples, the previous content of specified files will be overwritten with the new information. If instead, you want the new information to be appended to the existing content of those files, use instead the operators: `1>>`, `2>>` and `&>>`.

If the file descriptor number is not specified, it is defaulted to 1, i.e. `>` is exactly the same as `1>`, and `>>` is exactly the same as `1>>`.

Especially in the older **Bash** scripts you will see also `2>&1` redirection, but it has exactly the same meaning as `&>`, which was added only in more recent versions of **Bash**. The redirector `2>&1` means literally: Send *stderr* (file descriptor 2) to the same place where *stdout* (file descriptor 1) was sent. When `2>&1` is used, the order matters — first we need to indicate where `1>` is

redirected, and only then it makes sense to use `2>&1`. Because of this limitation, in practice it is much easier to use `&>` in such a context.



There is also a black hole in **Linux**, and it is called `/dev/null`. It happens frequently that you do not want to see the printout of some verbose command in the terminal, and you do not want to waste the disk space either by redirecting it to some file. Quite frequently, commands can print some warnings on the screen. After you have acknowledged them and concluded that those warnings are harmless, you clearly do not want to see them again and again. This is precisely where the special file `/dev/null` becomes very handy, because whatever you redirect to it, it is lost forever.

Example 1: How to redirect only the successful output of a command to a file, and ignore completely the error messages (which are sometimes just the very annoying and harmless warnings)?

This problem is solved with the following code snippet:

```
someCommand 1>someFile 2>/dev/null
```

With the above construct, the file `someFile` will contain only the successful output of `someCommand`. On the other hand, all error messages are permanently lost, because they were redirected to `/dev/null`.

Example 2: How to set programmatically the separate *stdout* and *stderr* streams in your own code?

This question is answered with the following concrete example, in which a function expects some arguments from the user. If the user supplied arguments, the functions prints successful *stdout* stream, and if the user failed to provide arguments, it prints the error message via *stderr* stream:

```
function myFunction
{
 [[ $# -eq 0 ]] && echo "Error: No arguments" >&2 && return 1
 echo "Arguments supplied" >&1 && return 0
}
```

With such an implementation, it is now possible programmatically to handle both *stdout* and *stderr* streams of this function:

```
myFunction a b c 1>output.log 2>error.log
```

In the above use case, the user has supplied some arguments ('a', 'b', 'c'), and therefore only the file `output.log` is filled, with the message defined in the function body for the *stdout* stream, namely 'Arguments supplied'.

```
myFunction 1>output.log 2>error.log
```

In the above example, no arguments were supplied. This is treated as an error within the function and it triggers its *stderr* stream, which is the message defined as 'Error: No arguments' in the function body.

Let us also say a few words about the last file descriptor 0, *stdin* ('standard input'). In general, *stdin* comes from the keyboard, but we can also feed a command with the content of some file. Schematically, we would use:

```
someCommand < someFile
```

The operator `<` redirects the content of `someFile` into the argument of `someCommand`. In fact, `<` is nothing but the shortcut synonym for `0<` redirection. Because a lot of commands, e.g. **cat** or **more**, expect by default input from a file, the below three versions are all equivalent:

```
cat someFile
cat < someFile
cat 0< someFile
```

When you are checking the content of some file with **cat**, you are essentially redirecting its content into *stdin* (file descriptor 0) for the **cat** command.

3. Code blocks and brace expansion: { ... }

Clearly, the file descriptors are an extremely nice feature, but they would be even nicer if we would be able to use them to handle the output streams of multiple commands in one go, instead of redirecting the output stream of each command separately. This is possible in **Bash** by using the *code blocks*.

The code block in **Bash** is basically any sequence of commands within curly braces `{ ... }`.

Before presenting the concrete use cases, we first summarize the general facts about code blocks:

1. `{ ... }` inherits the environment and can modify it globally;
2. `{ ... }` has its own `1>` and `2>` streaming facilities;
3. `{ ... }` does not launch a separate process. Therefore, the rest of a script or a function needs to wait for all commands in the code block to finish.

Consider the following code snippet:

```
echo "before code block"
{
 echo "inside code block"
 date
 dateee # intentionally introduced error
} 1>output.log 2>error.log
```

In the very last line, we have redirected the *stdout* and *stderr* streams of all commands within the code block in one go. If we now check the content of files `output.log` and `error.log`, we find the following lines in the file `output.log`:

```
inside code block
Do 23. Mai 08:56:49 CEST 2019
```

and the following line in the file `error.log`:

```
dateee: command not found
```

On the other hand, on the screen the only printout is:

```
before code block
```

because we did not redirect the first **echo** command anywhere.

Some other piece of code in the same script or function can be embedded into another code block, and then redirected to some other files. This way we can easily profile the code with redirectors, and decide what goes on the screen and what is dumped in files. Typically, code blocks `{ ... }` are used when it is not beneficial to break down some large monolithic script into functions.

When it comes to redirections, it is possible to treat loops analogously as code blocks. In particular, **for** and **while** loops have their own *stdout* and *stderr* streams, which can be redirected to the output files with `1>` and `2>` operators. In this way, we can easily disentangle what is happening in a particular loop, from what is happening in the rest of the code. Schematically, we would use for **for** loop:

```
for Var in someList; do
 ... some commands ...
done 1>output.log 2>error.log
```

We can simultaneously feed the **while** loop from an external file, and redirect its *stdout* and *stderr* in some other external files, schematically:

```
while read Line; do
 ... some commands ...
done <someFile.log 1>output.log 2>error.log
```

This way we can elegantly parse and modify programmatically the example file `someFile.log` line-by-line, save the modified new content immediately in the file `output.log`, and all errors which might occur during the editing we save in a separate file `error.log`.

To check the influence of code block on the environment in your terminal, you can execute the following code snippet:

```
Var=44
echo "Before : $Var"
{
 echo "Inside : $Var"
 Var=55
}
echo "After  : $Var"
```

Upon execution, this code snippet produces:

```
Before : 44
Inside : 44
After  : 55
```

From this example we can easily see that the code block inherits all settings from the global environment, and that all modifications made inside the code block (e.g. a variable gets a new value) are propagated outside to the global environment, after the code block terminates. The different behavior can be obtained by enclosing the particular code within different type of braces, namely the round braces `( ... )`, to define the *subshell* — this will be covered later.

Very conveniently, the code block `{ ... }` can be combined with the command chain operators, as the next example illustrates.

Example: Is it possible to condense the following lines into a single line:

```
someCommand
ExitStatus=$?
[[ $ExitStatus -eq 0 ]] && command1 && command2 && ...
```

By using the code blocks, this can be rewritten as:

```
someCommand && { command1 && command2 && ... ; }
```

Note the mandatory trailing semicolon `;` within code block in this context. This is important, because you need to indicate that `}` is not an argument to the last command within the code block — the last command input is terminated with semicolon `;`.

Brace expansion

We close this section with a side remark on curly braces. Besides being used to mark the code blocks, curly braces are also used in a completely different context to define programmatically the sequences, via the so-called *brace expansion*.

The syntax to generate sequences by using the brace expansion is demonstrated with the following concrete examples:

```
echo {1..10}
echo {1..10..2}
echo {10..1}
echo {a..f}
echo {-4..4}
```

The corresponding printouts are:

```
1 2 3 4 5 6 7 8 9 10
1 3 5 7 9
10 9 8 7 6 5 4 3 2 1
a b c d e f
-4 -3 -2 -1 0 1 2 3 4
```

Brace expansion is very frequently used in enumerating sequentially either files or directories.

Example 1: How to make 100 new directories named `Dir_0, Dir_1, ... Dir_99`?

The solution is very elegant by using the brace expansion mechanism:

```
mkdir Dir_{0..99}
```

Example 2: How to make 100 new files named `File_0.data, File_1.data, ... File_99.data`?

We can both prepend and append strings to the brace expansion, so also in this case there is a very elegant solution:

```
touch File_{0..99}.data
```

Brace expansion can be used also in combination with arbitrary string patterns.

Example 3: How to make three new files named `someLengthyFileName.log`, `someLengthyFileName.png` and `someLengthyFileName.pdf` in one go?

The solution is:

```
touch someLengthyFileName.{log,png,pdf}
```

which clearly saves a lot of typing.

Multiple brace expansions can be combined in the same command input. For instance, having already named directories or files sequentially, we can easily manipulate only a subset of them, by using the brace expansion.

Example 4: Imagine that in some directory you have the following files:

```
File_0.log File_1.log ... File_999.log
File_0.inf File_1.inf ... File_999.inf
File_0.dat File_1.dat ... File_999.dat
```

How to delete each 4th file within the interval 111 to 222, whose extension is `.log` or `.inf`, but not `.dat`? If you use the brace expansion, the solution is very simple and elegant:

```
ls File_{111..222..4}.{log,inf} # always do 'ls' before deleting!
rm File_{111..222..4}.{log,inf}
```

Without brace expansion the solution would take much more work. It is also possible to nest brace expansion, but this is rarely used in practice.

4. Conditional statements

We have already seen how to branch the code execution in **Bash** by using the command chain `&&` and `||`. For more complicated cases, however, a more elegant and flexible solution can be reached with *conditional statements*, which in **Bash** work very similar like in most programming languages. For simpler cases, we can use **if-elif-else-fi** conditional statement, while the syntax of **case-in-esac** is better suitable for more complicated cases.

A) if-elif-else-fi

The typical use case of **if-elif-else-fi** conditional statement is to branch the code execution depending on the outcome of the test construct `[[ ... ]]`. Schematically:

```
if [[ someExpression ]]; then
 some code when someExpression succeeds
elif [[ someOtherExpression ]]; then
 some code when someOtherExpression succeeds
 ... even more elif statements ...
else
  some code when all tests above failed
fi
```

You can have as many different **elif**'s branches as you wish, but the very last branch must start with the keyword **else**, and it has to be closed with the keyword **fi**. The keyword **then** does not need to be placed on the same line with keywords **if** and **elif**, a completely equivalent syntax is:

```
if [[ someExpression ]]
then
 some code when someExpression succeeds
elif [[ someOtherExpression ]]
then
 some code when someOtherExpression succeeds
 ... even more elif statements ...
else
  some code when all tests above failed
fi
```

However, if the keyword **then** is placed on the same line with keywords **if** and **elif**, it has to be separated with semicolon `;`.

Another typical use case of **if-elif-else-fi** conditional statement is to branch the code execution depending on whether a command or a function execution succeeded (exit status 0) or failed (exist status 1 to 255). Schematically:

```
if someCommand; then
  some code when someCommand succeeded
elif someOtherCommand; then
  some code when someOtherCommand succeeded
  ... even more elif statements ...
else
  some code when all commands above failed
fi
```

In practice, you frequently need to check only the exit status of a command, and do not need to see any output stream when executing that command. That can be achieved with:

```
if someCommand &>/dev/null; then
```

In the same way, you can use all other file descriptors, like `1>` and `2>`, as a part of **if** or **elif** statement.

It is possible to use the command chain within the same **if** or **elif** statement:

```
if someCommand && someFunction; then
```

In this example, the corresponding branch will be executed if the exit status of all chained commands is 0.

Finally, it is also possible to execute sequentially different commands within the same **if** or **elif** statement:

```
if command1; someFunction; command2; then
```

In this example, the corresponding branch will be executed only if the exit status of the very last command **command2** is 0, the exit status of previous commands play no role in this version.

B) case-in-esac

On the other hand, the syntax of **case-in-esac** conditional statement is more elaborate, but also more powerful. The generic syntax looks like:

```
case someValue in
 firstOption) some code when this option is met ;;
 secondOption) some code when this option is met ;;
 ... even more options ...
 *) some code when all specified options are not met ;;
esac
```

The thing to remember is that in **case-in-esac** conditional statement a specific branch of code execution is embedded within round brace `)` and double semicolon `;;` (yes, double semicolon, no empty character is allowed between semicolons here!). This peculiar syntax, the unbalanced round brace `)` and the double semicolon `;;` are special to **case-in-esac** conditional statement, and therefore easy to remember.

The usage of **case-in-esac** conditional statement is best illustrated with a few concrete examples.

Example: How to implement the support for options in your script or function?

Schematically, for the simplest cases, that can be achieved with the following code snippet:

```
Flag=$1
case $Flag in
 -a)
     echo "The option -a has been specified!"
     echo "For the option -a we do the following..."
     ... some code ...
```

```
  ;;
 -b)
     echo "The option -b has been specified!"
     echo "For the option -b we do the following..."
     ... some code ...
  ;;
  *)
     echo "The specified flag is not supported"
     return 1 # bail out with error exit status
  ;;
esac
```

For more elaborate cases on how to parse command-line arguments in such context, see **Bash** built-in command **getopts**.

Multiple options can be grouped with `|` (OR) under the same statement, schematically:

```
case someValue in
 firstOption | secondOption | ... )
      ... some code when one option from this group is met ...
    ;;
 someOtherOption | yetAnotherOption | ... )
      ... some code when one option from this group is met ...
    ;;
    ... even more options ...
*) some code when all specified options are not met ;;
esac
```

The **case-in-esac** conditional statement recognizes the so-called POSIX brackets. The most important examples are:

- `[[:alpha:]]` Alphabetic characters [a-zA-Z]
- `[[:digit:]]` Digits [0-9]
- `[[:alnum:]]` Alphanumeric characters [a-zA-Z0-9]

Example use case:

```
Var=someValue
case $Var in
 [[:alpha:]])
   echo "Var is a single alphabetic character"
 ;;
 [[:digit:]])
   echo "Var is a digit"
 ;;
 *)
   echo "Var is something else"
 ;;
esac
```

As the final remark, when developing the code using conditional statements, sometimes we are not sure immediately what to implement in the particular branch. We cannot leave that branch empty or only insert a comment, because both will produce an error:

```
if [[ ${Var1} -gt ${Var2} ]]; then
 # I will implement this part later
fi
```

The error message is:

```
line 4: syntax error near unexpected token `fi'
line 4: `fi '
```

For this sake, we need to use the so-called 'do-nothing' command as a placeholder. The syntax of 'do-nothing' command is simply a colon `:`.

The correct solution to the above problem is:

```
if [[ ${Var1} -gt ${Var2} ]]; then
 : # I will implement this part later
fi
```

'Do nothing' command `:` does literally nothing, except that it always returns the exit status 0, i.e. it always succeeds in what it needs to do, which is not surprising given that fact that it does nothing:

```
:
echo $?
# prints 0
```

Quite remarkably, even such a simple command has some interesting and frequent use cases.

Example: How to empty the already existing file, keeping all file permissions intact?

```
: > someFile
```

Literally, we have redirected nothing into the existing file, therefore its content is now nothing. Note that we have kept all file permissions intact in a process. Therefore, this is in general not the same as deleting the existing file, and then creating a new empty file with the same name:

```
rm someFile
touch someFile
```

because now the permissions of a new file are set to default permissions, and we need to invest some additional work to set again our own permissions.

Example: Infinite loop in **Bash**.

The simplest implementation is:

```
while :; do
 ... some code ...
done
```

Example: Ignore the exit status of command.

This is the common idiom:

```
someCommand || :
```

The above construct always evaluates to true, irrespectively of what was the exit status of 'someCommand'.

We can also use 'do-nothing' `:` command to write a multi-line comment in **Bash** in combination with the so-called *here-documents* — this will be covered later.