


### Virtual Machine Update

The course VM has been updated, comprising the latest updates of the operating system, as well as CLion 2018.1, the latest Swift toolchain etc. The new version will be deployed on the lab workstations as soon as possible. In the meanwhile, enjoy the new version by downloading it from Moodle.

### Exercise 27 – "Hello, World!" – Command-line

Write a  Swift program printing "Hello, World!" to standard output.

- Use only the command-line interface to create (**nano**), compile (**swiftc**) and run the program!
- Also try to run your program as a shell script, by either using the **swift** utility or putting the right shebang line.

### Location of **swift** and **swiftc** binaries

The **swift** and **swiftc** binaries can be found in `/home/student/swift` (which is a symbolic link to another directory). Feel free to add this directory to the `$PATH` variable:

```
$ cd && nano .bash_aliases
```

```
export PATH=$PATH:/home/student/swift
```

```
$ source .bash_aliases
```

### Exercise 28 – "Hello, World!" – CLion

Repeat the previous exercise in CLion.

1° Open CLion and create a **New Project** for a **Swift Executable**.

Note that a file `main.swift` is created which will print `Hello, World!` to the console. Along this file, you also find `Package.swift` and `CMakeLists.txt`, latter being used to build the binaries of your program.

2° Select **Run** > **Build** to build the binaries.

3° The first time you will run the executable requires a little configuration. Select **Run** > **Run 'HelloWorld'**. In the dialog, under **Executable**, choose **Select other...**. To the current path, append `/.build`. Once the treeview refreshed, select the executable `HelloWorld` in the subfolder `debug`. Choose **Apply** and **Run**. The message `Hello, World!` should now be shown in the console. This step is only needed the first time you run your program.

### Adding more files to the target

When creating more .swift files that should be included in the build target, make sure to list these files also in the CMakeLists.txt file, e.g.

```
1 cmake_minimum_required(VERSION 3.10)
2 project(HelloWorld)
3
4 add_custom_target(HelloWorld ALL
5     COMMAND /home/student/swift-4.1-RELEASE-ubuntu16.04/usr/bin/swift build
6     WORKING_DIRECTORY ${CMAKE_SOURCE_DIR}
7     SOURCES Sources/main.swift Sources/foo.swift Sources/bar.swift)
```


### Alternative: Using the Atom editor to create Swift programs under Ubuntu

Another way of writing Swift programs under Ubuntu is to use the [Atom](#) editor, which is described in [this Medium article](#). Note that the necessary Atom packages are already installed on the course VM.

### Mac Users

On macOS, Xcode is probably the most appropriate choice for writing Swift programs, allowing furthermore app development for macOS and iOS. You can either use regular Xcode projects or Playgrounds. iPad users may also check out [Swift Playgrounds](#), an app for learning Swift programming in a playful way.

## Exercise 29 – Web Server

Write a  Swift console application that emulates a web server, which serves both static files as well as scripts. Essentially, the functionality could be written in a function

```
func serve(url:String, params: [String:String]?) -> (code:Int, description:String)
```

which takes the url of the incoming request as well as an *optional* dictionary params. It returns a tuple comprising the HTTP status code and a short description of the status.

- A constant dictionary will map the URLs to the corresponding files, e.g. /about will serve the static file about.html
- If an incorrect URL is given, an HTTP error 404 (Resource not found) is returned.
- Static files, ending in .html, will be served by printing the file name to the console. *Hint:* The method .hasSuffix might be useful!
- Any file not ending in .html is considered a script. Scripts can only be executed with a set of parameters. If no parameters are given, return an HTTP error 500 (Internal server error). Otherwise, print the parameter key-value pairs to the console.

- If a static file or a script could be successfully served respectively executed, the HTTP status 200 (OK) is returned.
- Furthermore, each request to a certain URL shall be logged in a history (e.g. a String array).

Try the server with several requests. Print the status codes and descriptions each time, as well as the final browsing history. A possible output could look like this:

```
---> Requesting /about ...
Serving static file about.html ...
---> Server response: Status 200: OK. ---

---> Requesting /test ...
---> Server response: Status 404: Resource at /test not found. ---

---> Requesting /download ...
---> Server response: Status 500: Internal server error: Parameters expected. ---

---> Requesting /download ...
Will execute download.php with the following parameters:
file=ChamberLeak.zip
mode=Staubsauger
---> Server response: Status 200: OK. ---

Browsing history:
- /about
- /test
- /download
- /download
```