COMPILING THE CODE IN ERLANG

Erlang code is compiled to bytecode in order to be used by the virtual machine. You can call the compiler from many places: `$ erlc flags file.erl` when in the command line, `compile:file(FileName)` when in the shell or in a module, `c()` when in the shell, etc.

It's time to compile our useless module and try it. Open the Erlang shell, type in:

```
1> cd("/path/to/where/you/saved/the-module/").
"Path Name to the directory you are in"
ok
```

By default, the shell will only look for files in the same directory it was started in and the standard library: `cd/1` is a function defined exclusively for the Erlang shell, telling it to change the directory to a new one so it's less annoying to browse for our files. Windows users should remember to use forward slashes. When this is done, do the following:

```
2> c(useless).
{ok,useless}
```

If you have another message, make sure the file is named correctly, that you are in the right directory and that you've made no mistake in your module. Once you successfully compile code, you'll notice that a `useless.beam` file was added next to `useless.erl` in your directory. This is the compiled module.

Let's try our first functions ever:

```
3> useless:add(7,2).
9
4> useless:hello().
Hello, world!
ok
5> useless:greet_and_add_two(-3).
Hello, world!
-1
6> useless:not_a_real_function().
** exception error: undefined function
useless:not_a_real_function/0
```

The functions work as expected: `add/2` adds numbers, `hello/0` outputs "Hello, world!", and `greet_and_add_two/1`does both! Of course, you might be asking why `hello/0` returns the atom 'ok' after outputting text. This is because Erlang functions and expressions must **always** return something, even if they would not need to in other languages. As such, `io:format/1` returns 'ok' to denote a normal condition, the absence of errors.

Expression 6 shows an error being thrown because a function doesn't exist. If you have forgotten to export a function, this is the kind of error message you will have when trying it out.

Note: If you were ever wondering, '.beam' stands for *Bogdan/Björn's Erlang Abstract Machine*, which is the VM itself. Other virtual machines for Erlang exist, but they're not really used anymore and are history: JAM (Joe's Abstract Machine, inspired by Prolog's WAM and old BEAM, which attempted to compile Erlang to C, then to native code. Benchmarks demonstrated little benefits in this practice and the concept was given up.

There are a whole lot of compilation flags existing to get more control over how a module is compiled. You can get a list of all of them in the Erlang documentation. The most common flags are:

-debug_info

Erlang tools such as debuggers, code coverage and static analysis tools will use the debug information of a module in order to do their work.

-{outdir,Dir}

By default, the Erlang compiler will create the 'beam' files in the current directory. This will let you choose where to put the compiled file.

-export_all

Will ignore the `-export` module attribute and will instead export all functions defined. This is mainly useful when testing and developing new code, but should not be used in production.

-{d,Macro} or {d,Macro,Value}

Defines a macro to be used in the module, where *`Macro`* is an atom. This is more frequently used when dealing when unit-testing, ensuring that a module will only have its testing functions created and exported when they are explicitly wanted. By default, *`Value`* is 'true' if it's not defined as the third element of the tuple.

To compile our `useless` module with some flags, we could do one of the following:

```
7> compile:file(useless, [debug_info, export_all]).
{ok,useless}
8> c(useless, [debug_info, export_all]).
{ok,useless}
```

You can also be sneaky and define compile flags from within a module, with a module attribute. To get the same results as from expressions 7 and 8, the following line could be added to the module:

```
-compile([debug_info, export_all]).
```

Then just compile and you'll get the same results as if you manually passed flags. Now that we're able to write down functions, compile them and execute them, it's time to see how far we can take them!

Note: another option is to compile your Erlang module to native code. Native code compiling is**not** available for every platform and OS, but on those that support it, it can make your programs go faster (about 20% faster, based on anecdotal evidence). To compile to native

code, you need to use the `hipe` module and call it the following way:`hipe:c(Module,OptionsList).` You could also use `c(Module,[native]).` when in the shell to achieve similar results. Note that the .beam file generated will contain both native and non-native code, and the native part will not be portable across platforms.

Source : http://learnyousomeerlang.com/modules#what-are-modules