The Hash-bang line, or how to make a Perl scripts executable on Linux

In the very first scripts we wrote we did not have this construct, but this can be useful for Perl scripts on Unix-like systems such as Linux and Mac OSX.

Not really required, you can easily skip this article and come back later when you would like to understand what does the #!/usr/bin/perl mean at the beginning of many Perl scripts.

Before going in the details, let me just point out that this line is also called she-bang or Shebang, or sh-bang and a couple of other names.

The first program people usually learn is the "Hello world" program. This is the script:

```
1. use strict;
2. use warnings;
3.
4. print "Hello World\n";
```

We can save it in a file called hello.pl, open the terminal (or Cmd in Windows). cd to the directory where we saved the file and run the script by typing perl hello.pl.

That is, we run Perl and tell it to run our script.

Would it be possible to run the script without running perl first? Would it be possible to just run hello.pl?

On Unix-like systems it is quite easy. On Windows it is a different story that will be addressed separately.

hash-bang on Unix-like systems

Let's try to run the script:

```
$ hello.pl
-bash: hello.pl: command not found
```

Our environment cannot find the script.

What if we give it the path to the script (which is in the current directory anyway):

```
$ ./hello.pl
```

```
-bash: ./hello.pl: Permission denied
```

Now it already finds the script, but we don't have permission to run it.

On Linux, or Mac OSX, or on any Unix system we can make every file "Executable" by flipping a bit in files's standard attributes in the so-called inode table. It can be easily done using the chmod command. We will use chmod u+x hello.pl:

First we use the ls -l command of the Unix shell to show the situation before, then we use chmod to change the mode, and then we show the situation after. The u+x part of the operation tells chmod to add executable (x) rights to the user (u) who owns this file, but not for anybody else. (chmod +x hello.pl would give executable rights to everyone on the system.)

```
$ ls -l hello.pl
-rw-r--r--  1 gabor  staff  50 Apr 21 10:11 hello.pl

$ chmod u+x hello.pl

$ ls -l hello.pl
-rwxr--r--  1 gabor  staff  50 Apr 21 10:11 hello.pl
```

Please note the additional x as the 4th character of the response.

Let's try to run it again:

```
$ ./hello.pl
./hello.pl: line 1: use: command not found
./hello.pl: line 2: use: command not found
./hello.pl: line 4: print: command not found
```

Much better :)

Now we can already execute the script but it does not do what we want. It actually complains that it cannot find the commands 'use' and 'print'. What happened here is that the shell we use (probably bash) tried to interpret the commands in the file, but it does not find commands such as use and print in Linux/Unix. Somehow we need to tell the shell that this is a perl script. That's what the hash-bang is used for.

If we edit the file and add

```
#!/usr/bin/perl
```

as the very first line to the script and without spaces, and then we run the script again:

```
$ ./hello.pl
Hello World
```

it already works as expected.

However, if we try to run it without the ./ it still cannot find it:

```
$ hello.pl
-bash: hello.pl: command not found
```

In order to solve this we need to change the PATH environment variable. As our focus is mostly the hash-bang line, I don't want to go into further extensive explanation so let me just give you the command:

```
$ PATH=$PATH:$(pwd)
```

will append the current working directory to the list of directories in the PATH environment variable. Once we do this we can now run:

```
$ hello.pl
Hello World
```

How does the hash-bang line work?

We added #!/usr/bin/perl as the first line of our script:

When we run the script, we run it in our current shell environment. For most people on Linux/Unix that will be Bash. Bash will read the first line of the script. If it starts with a hash and a bang (hash-bang) #! then Bash will run execute the application that has its path on the hash-bang line (in our case /usr/bin/perlwhich is the standard location of the perl compiler-interpreter on most modern Unix-like system.

The hash-bang line holds the path to the Perl compiler-interpreter.

If the first line did not start with #! as was the case with our original script, Bash would assume this script is written in Bash and would try to understand itself. That's what generated those errors.

Alternative hash-bang lines using env

While we used #!/usr/bin/perl as our hash-bang line there can be other as well. For example if we have installed another version of perl in a different location and we would like our scripts to use that, then we can put the path to that version of perl. For example #!/opt/perl-5.18.2/bin/perl.

The advantage of setting a hash-bang (and turning on the executable bit) is that user does not have to know the script is written in Perl and if you have multiple instances of Perl on your system the hash-bang line can be used to pick which perl to be used. This will be the same for all the people on the specific machine. The drawback is that the perl listed in the hash-bang line is only used if the script is executed as./hello.pl or as hello.pl. If it is executed as perl hello.pl it will use the version of perl that is found first in the directories listed in PATH. Which might be a different version of perl from the one in the hash-bang line.

Because of this, on modern Linux/Unix systems, people might prefer to use #!/usr/bin/env perl as the hash-bang line. When Bash sees this line it will first run the env command passing the name perl to it. env will find the first perl in the directories of PATH, and run that. So if we have #!/usr/bin/env perl in our script it will always use the first perl in our PATH. Both when it is executed as ./hello.pl and when it is executed as perl hello.pl. This too has a disadvantage, because this relies on the users setting their PATH environment correctly.

Here is a table that tries to explain the 4 cases:

```
  the hash-bang              Which perl is used to run the script when
call it either of these ways:
                             ./hello.pl                   perl hello.
pl


  /usr/bin/perl              /usr/bin/perl                first perl
in the PATH
  /usr/bin/env perl          first perl in the PATH       first perl
in the PATH
```

Flags on the hash-bang line

On the hash-bang line, after the path to perl we can add command-line flags to perl. You will probably see lots of scripts starting with #!/usr/bin/perl -w or maybe #!/usr/bin/env perl -w. The -w in this hash-bang turns on warnings. This is quite similar to what use warnings does, but this is the old style. You won't see this in most of the modern Perl scripts.

Another common flag that you might see on the hash-bang lines are -t and -T. They turn on the so-called **taint-mode** that will help you write more secure code.