Getting started with Classic Perl OOP

OOP bless ->

While a lot of people recommend using advanced OOP systems for Perl such as Moose or Moo, there are many applications that use the classic OOP system of Perl. If you want or need to understand such applications, you'd better make yourself familiar with the classic OOP system of Perl.

Besides, both Moose and Moo are built on top of the classic system, so understanding it is important to gain better understanding of what the more high-level OOP systems do.

I am sure you have already seen code in Perl that used Object Oriented modules, but let's have a small example here:

```
use strict;
use warnings;

my $url = 'https://perlmaven.com/';
use WWW::Mechanize;
my $mech = WWW::Mechanize->new( agent => 'wonderbot 1.01' );
$mech->get( $url );
print $mech->content;
```

In this example we load the `WWW::Mechanize` module with the `use` statement, just as we would do with any other module. In OOP terminology this module is the **class**. Then we use the single-arrow notation to call the `new` **constructor**, passing a pair of values to it. This constructor returns an **object**, which is also called an **instance** of the class. This is assigned to a scalar variable.

Then we call the `get` **method** on the object passing a parameter to it (`$url`). Finally we call the `content` method of the object, this time without passing any parameter.

(In most other programming languages the dot `.` is used to invoke the methods of an object. In Perl, by the time OOP was added to the language, the `.` character was already used as the concatenation operator, so Larry Wall had to chose something else.)

Try to save the above code in a file called `get.pl` and run it using `perl get.pl`. If you have WWW::Mechanize installed, you will see the content of the home page of the Perl Maven site zooming by. If you don't have it installed you'll get an error message Can't locate WWW/Mechanize.pm in @INC.

Can't locate WWW/Mechanize.pm in @INC (you may need to install the WWW::Mechanize module)

Let's see what is needed to implement the skeleton of the above code.

We go step-by-step, first we create a package, then the constructor and then the other methods.

Creating a package

When perl encounters `use WWW::Mechanize`, it searches the directories listed in `@INC` array. In each one of them it looks for a subdirectory called `WWW` and inside the subdirectory a file called `Mechanize.pm`

In the same directory where the `get.pl` was saved, we create a subdirectory called `lib`, and inside that a subdirectory called `WWW`, and inside that directory a file called `Mechanize.pm`. We have a directory structure like this:

```
../get.pl
   lib/WWW/Mechanize.pm
```

The content of Mechanize.pm will be the following:

```
1. package WWW::Mechanize;
   1;
```

That's a `package declaration` and then the `1;` at the end of the file. So far this is just a module, it does not have any OOP feature. Let's try to run the script again: `perl get.pl`. You will get the exact same behavior as previously. If you had WWW::Mechanize installed, perl found it, if you did not have it, perl still could not find it. In neither case did perl find the version of WWW::Mechanize we started to put together. The reason is that the `lib` directory, where we put the module, is not in the `@INC`. There are various ways to change @INC. We'll do the most suitable to our case and we will run the perl script using `perl -Ilib get.pl`. The `-I` flag of perl will put the parameter it gets at the beginning of `@INC`. Once we run the script in the above way we get the following error:

```
Can't locate object method "new" via package "WWW::Mechanize" at get.pl line 6.
```

This means, perl found our (almost empty) version of `WWW::Mechanize`. Successfully loaded it, but when it tried to call `->new`, it could not find the implementation. Which is of course not surprising as we have not written it yet.

What is that 1; at the end?

A little detour, let's change the Mechanize.pm file, removing the 1; from the end so we have:

```
1. package WWW::Mechanize;
```

If we run the script now: `perl -Ilib get.pl`, we will get:

```
WWW/Mechanize.pm did not return a true value at get.pl line 5.
BEGIN failed--compilation aborted at get.pl line 5.
```

Perl needs to have a boolean true value at the end of each module. The boring `true` value is `1;`, but I've seen modules with `42;`, quotes from poetry, and even the string `"false";`. I am sure this freaks out some unsuspecting readers of that code. In reality all Perl cares about is that the value will be boolean true in the Perlish sense.

The constructor

So far we created a module but it is not a class yet.

The next step is adding a constructor that will create an instance (or object) of the class. If you come from another language or if you have seen Moose or Moo you might be surprised that we need to create a constructor, but this is how the classic OOP system of Perl works, and this is what happens in both Moose and Moo under the hood.

Let's change the Mechanize.pm file to have the following code:

```
 1. package WWW::Mechanize;

    sub new {
        my ($class, %params) = @_;
 5.
        my $self = {};
        while ( my($key,$value) = each %params ) {
            $self->{$key} = $value;
        }
10.     bless $self, $class;
11.
        return $self;
    }

15. 1;
```

The `new` **constructor** is just a regular subroutine declared with the `sub` keyword. Nothing is really special about it. Not even its name! In Perl, any subroutine can act as a constructor, the word `new` is not reserved, or special in any way, except that it is the most common name for the constructor.

In Object Oriented programming people usually use the term `method` for all the actions you can do on the class and on the object. (In our example that means `get`, `content` and even `new` are methods. In Perl there is no special keyword to create a method. Methods (including the constructor) are declared using the `sub` keyword just as regular functions are. The difference is how you call

them (methods are always called with the single-arrow notation) and what is their first parameter. When we call the `new` method on the class-name (in our case WWW::Mechanize), perl will look for the `new` subroutine in the `WWW::Mechanize` package and call that function passing all the parameters the user sent and, the name of the class (`WWW::Mechanize`) as the first parameter.

You can read more about methods, functions and subroutines in Perl.

When we call `WWW::Mechanize->new( agent => 'wonderbot 1.01' );` perl will execute then `new` functions with 3 parameters: `'WWW::Mechanize', 'agent', 'wonderbot 1.01'`

In the `new` method, we are expecting the first parameter to be the class-name, hence we put it in a variable called `$class`. For the rest of the parameters we are expecting a set of key-value pairs and we put them in the `%params` hash.

In the next step we create the skeleton of the `object` and put it in the `$self` variable. The object is going to be represented as a reference to a hash. It starts out empty: `{}` but we will fill it soon. Basically the `attributes` (or in other name the `members`) of the class will be kept in this hash. The `attribute` name will be the key in the hash, and the attribute value will be, well, the value in the hash.

This is what happens in the next 3 lines: We go over the `%params` hash holding the parameters we received from the user and assign each key-value pair to the hash reference we just created.

```
1.      while ( my($key,$value) = each %params ) {
            $self->{$key} = $value;
        }
```

BTW we called the variable `$self`. This variable name is arbitrary, we could have used anything there. Some people use the name `$this`, `$me`, or even just `$o`, but the most common name people use to hold the current object inside the implementation is `$self`.

So we have the skeleton of the object in `$self` already filled with values, but so far it is just a regular hash reference. Without any special powers.

The next call, `bless $self, $class;` is what turns a mere hash-reference into an `instance of the class`. A bit later I'll explain what is that magic and why do we need it.

Once that's done, we return the newly created object.

If we run the script again: `perl -Ilib get.pl`

We get the following error:

```
Can't locate object method "get" via package "WWW::Mechanize" at get.pl line 7.
```

That looks good. It means, the call to `new` worked and we need to implement `get` now. Before we do that, let's make another small detour.

Detour: what does the object look like?

First, change the script and a `print` statement right after the call to `new`

```
1. my $url = 'https://perlmaven.com/';
   use WWW::Mechanize;
   my $mech = WWW::Mechanize->new( agent => 'wonderbot 1.01' );
   print "$mech\n";
```

and run the script again: `perl -Ilib get.pl`

```
WWW::Mechanize=HASH(0x7ffaf8805480)
```

We see that the object is a reference to a hash, but we also see that it is somehow related to `WWW::Mechanize`.

If you now comment-out the call to `bless` in the Mechanize.pm file, and run the script again you will see:

```
HASH(0x7fab54005480)
```

This is just a plain reference to a hash without any relation to `WWW::Mechanize`.

This can be useful when you see something like that printed out by mistake or during a debugging session, so you can recognize if a reference is a plain reference or if it was blessed into a class.

Detour: why do we need to bless the reference?

Now change the `get.pl` script back so it will call `$mech->get($url)` after calling `new`, but keep the `bless` commented out and run the script. The resulting error is:

```
Can't call method "get" on unblessed reference at get.pl line 7.
```

Now enable the call to `bless` in Mechanize.pm and run the script again. This time the error will be:

```
Can't locate object method "get" via package "WWW::Mechanize" at get.pl line 7.
```

In the first case, perl did not know what to do at all. In the second case, when we had the blessing, perl already knew it needs to look for the `get` method in the `WWW::Mechanize` class. It just could not find it, because we have not implemented it yet.

If we have a call `Person->new`, perl will look for a functions called 'new' in the 'Person' name-space. (or module or package). If we call `$foo->do_something` then if $foo holds a simple reference, perl

does not know what to do with this code and it gives you the error about unblessed reference.

OTOH if $foo is blessed - if it is connected to a name-space - then perl will look for a function called `do_something` in that name-space.

That's what `bless` does. It connects a reference to a name-space.

Adding a method

Now that we have an object, we would like to go on and implement the `get` method:

```
1. sub get {
       my ($self, $address) = @_;

       require LWP::Simple;
5.     $self->{content} = LWP::Simple::get($address);

       return;
   }
```

The implementation of the `get method` is just a regular subroutine. When we call `$mech->get($url)` in the script, Perl will look for the `get` subroutine in `WWW::Mechanize` and will call it passing all the arguments the user sent (In our case it is the content of the `$url` variable), and passing the object as the first parameter. In case of `$mech->get($url)`, perl will call `get($mech, $url)`.

In the implementation of `get` we copy the first parameter to the `$self` variable, and the second parameter to the `$address` variable. (We'd probably use there the name `$url`, but in our example that would just create confusion with the variable in the script. So for the example let's call it `$address`.)

As mentioned above, the name `$self` is not special in any way, but it is quite common to use this variable to represent the current object inside the implementation of the class.

Then we use the LWP::Simple module, as a simple way to fetch the web page. It is very simple and there are other and more robust ways to fetch the content of a web page, but it is good enough for our current purposes.

The call `LWP::Simple::get($address);` will download the page and then we assign it to a key in the hash representing the object: `$self->{content}`.

Basically this is a new attribute of the object that was added during run time. As the object is just a hash we can do this. There is no need to declare the attributes up front.

Adding a getter

An `accessor` (or getter) is just another simple subroutine, that receives the current object as the first (and only) parameter and returns the content of the respective element of the hash:

```
1. sub content {
       my ($self) = @_;

      return $self->{content};
5. }
```

A little warning

The object is just a reference to a hash. It is fully exposed to the user as well. There is nothing in perl that would limit the access of the user to the content of the hash so a programmer could access the content of the object in the script:

```
1. $mech->{content} = "Something else";
```

There is nothing we can really do about it, at least not easily, but it is important we are aware of this possibility.

The full example

Finally, let's see the whole `Mechanize.pm` code:

```
 1. package WWW::Mechanize;

    sub new {
        my ($class, %params) = @_;
 5.
        my $self = {};
        while ( my($key,$value) = each %params ) {
            $self->{$key} = $value;
        }
10.     bless $self, $class;
11.
        return $self;
    }

15. sub get {
        my ($self, $url) = @_;

        require LWP::Simple;
        $self->{content} = LWP::Simple::get($url);
20.
```

```
21.     return;
    }

    sub content {
25.     my ($self) = @_;

       return $self->{content};
    }

30. 1;
```

Exercise

Change the implementation of Mechanize.pm to use the `agent` attribute. You will probably need to replace LWP::Simple with some other, more complex tool.