Constructor and accessors in classic Perl OOP

bless package @_ $_[0]

While for new applications it would be recommended to use Moo or Moose for writing Object Oriented Perl code, sometimes you cannot use them. For example when you maintain an old code base, or when your management does not let you install modules form CPAN.

We'll create a simple class representing a date. (Year, Month, Day).

We call our class "My::Date". It is usually recommended to put each class in its own file and to keep all the files implementing modules or classes in the lib directory. As the `::` separator is mapped to subdirectories, the code of the `My::Date` class should be placed in the `lib/My/Date.pm file of the project.`

(aka "best practices")

```
package My::Date;
use strict;
use warnings;

sub new {
    my ($class, %args) = @_;
    return bless \%args, $class;
}

1;
```

The code of the script using it should go in the scripts/ or `bin/` subdirectory of the project. In our case it is the file `bin/date.pl` and contains the following:

```
use strict;
use warnings;

use FindBin;
use File::Spec;
use lib File::Spec->catdir($FindBin::Bin, '..', 'lib');
```

```perl
use My::Date;

my $d = My::Date->new(year => 2013, month => 1, day => 27);
say $d;
```

The directory of the project looks like this:

```
$ tree
.
├── bin
│   └── date.pl
└── lib
    └── My
        └── Date.pm
```

First the basic boiler-plate of use strict, and `use warnings` and the declaration of the minimal Perl version. (The only reason this code requires 5.10 is so that I can use the `say` function. Otherwise 5.6 could be used as well.)

Then we change @INC to a directory relative to the script, and we load the `My::Date` module that contains our class definition.

A class in Perl is just a module with slightly different behavior.

Then we called the constructor (the `new` method) and printed the content of the returned value.

```
My::Date=HASH(0x7fea348eb300)
```

So far the usage was simple. The class definition itself starts with the `package` keyword that declares a new name-space. The `1;` at the end of the file is not very interesting. There needs to be a true value there.

The interesting part is the `new` method. While we could call our constructor any name, the accepted best practice is to use the name `new`. When the user calls `My::Date->new()` perl will call the `new` function in the `My::Date` package passing the name of the class "My::Date" as the first parameter. Before all the parameters the user passed to the constructor.

Hence `$class` will hold "My::Date" and `%args` will capture all the key-value pairs passed to the constructor.

The call to `bless` associates the hash-reference to `%args` with the name-space passed in the `$class` variable and returns the already blessed reference. This is returned to the caller. This is the object. (Or instance.)

When the user printed `$d`, she printed this blessed reference. That's why the output contains `My::Date=` in addition to the address of the hash reference.

Attributes

In all this code no attributes were mentioned. The keys passed to the constructor will be kept as keys in the blessed hash-reference. These are the attributes of the object. Perl, in this classic way of Object Oriented code does not provide any mechanism to check if all the required attributes were passed. If the values of the attributes were valid, or if there were additional attributes. There are tools to implement these, but for now we accept the classic behavior that will turn every key-value pair into attributes.

Accessors: Getters, setters

Having a constructor create an object is great, turning every parameter into an attribute without any check is lazy, but we would like to be able to retrieve and maybe even to change the content of the attributes.

For this, in classic Perl, we have to implement our getters and setters:

This code, as part of the Date.pm file implements the getters/setters for the 3 attributes we plan to handle:

```perl
sub year {
    my ($self, $value) = @_;
    if (@_ == 2) {
        $self->{year} = $value;
    }
    return $self->{year};
}

sub month {
    if (@_ == 2) {
        $_[0]->{month} = $_[1];
    }
    return $_[0]->{month};
}

sub day {
    return $_[0]->{day} = @_ == 2 ? $_[1] : $_[0]->{day};
}
```

When the user calls `$d->year(1001)` perl first checks what is in `$d`. In our case there is a reference `blessed` into the `My::Date` namespace. Therefor, perl will look for the `year` function in the `My::Date` package. If not found it will throw an exception:

```
Can't locate object method "year" via package "My::Date" at bin/date.pl line 28
```

If it finds the function, it will call it passing `$d` as the first parameter, before any parameters passed by the user. In our case `$d` will arrive in the variable `$self`, and 1001 will arrive in `$value`.

The name `$self` is not reserved, but most Perl programmers will use this name to contain the current instance. (This is similar to "self" in Python and "this" in Java.)

As the object, hold in `$self` is a reference to a hash, and the attributes are just keys in this hash, we can access the "year" attribute using `$self->{year}`. Because in this example we wanted to use the same function (`year`) to be both the setter and the getter, we need to first check if the user has passes 2 arguments - meaning the user want to set the attribute. In that case we assign the `$value` to the `$self->{year}` attribute.

In any we return the content of `$self->{year}`. That is, this accessor will always return the content of the attribute, but if it gets two parameters it will first set it to this new value.

The other two functions, `month` and `day` do exactly the same, they just don't use temporary variables.

This code will use the new accessors:

```perl
use strict;
use warnings;
use 5.010;

use FindBin;
use File::Spec;
use lib File::Spec->catdir($FindBin::Bin, '..', 'lib');
use My::Date;

my $d = My::Date->new(year => 2013, month => 1, day => 27);
say $d;
say $d->year;
say $d->month;
say $d->day;

say '';
say $d->year(1001);
say $d->year;

say '';
say $d->month(12);
say $d->month;
say '';
say $d->day(9);
say $d->day;
```