Object−Oriented Programming In Perl (part 1)

By Vikram Vaswani

Table of Contents

This Means War!

Object Oriented Programming.

A term that strikes fear into the heart of even the most determined newbie.

Conceptually, learning how to program in an object−oriented manner is a lot like going to war, with everything that preceded it akin to boot camp. Sure, you may know how to write functional, clean code, just as you may know how to clean a gun or live off dead beetles − but it's only when you're out in the jungle, with the bullets flying past your head, that those skills are actually put to the test. And so it is with OOP − it's a whole new programming paradigm, and it takes a little bit of getting used to.

The upside? If you survive, you'll come back to a hero's welcome, people will look at you with newfound respect, and getting a date on Saturday night will be a simple matter of flexing those OOP muscles and showing off those battle scars.

Think you can handle it?

You Say Tomahto, I Say Tomayto

All the bravado aside, OOP in Perl is somewhat complex, primarily because the language has a somewhat novel way of doing things. The first − and probably the most confusing thing − for a newbie trying to understand the intricacies of Perl objects is the profusion of terms used to describe them. You'll hear about "modules", packages", "classes", "references", "constructors" and a whole lot more...and each will leave you more confused than the last.

For the purposes of this tutorial, we're going to set a few ground rules which might help to simplify your understanding of how Perl objects work.

#1: Central to your understanding of OOP in Perl is the concept of "references". References are covered in detail in this here itty−bitty article.

#2: Once you've got that down, it's time to enter the weird and wonderful world of Perl "packages". For all intents and purposes, "packages", "classes" and "modules" are synonymous with each other (that's a little white lie, but don't let it bother you). Throughout this tutorial, we'll be using the word "packages".

#3: "Methods" and "objects" are thingummies which reside within a package. So are "constructors", "destructors" and "overrides". You don't need to worry about these until we get to the second part of this article.

Cool? OK, let's get started.

The Popeye Show

In Perl, a "reference" is exactly what it sounds like − an indicator, or pointer, to something else. It is a programming construct that allows programmers to access the value(s) stored in a variable without using the variable name.

By itself, this may not seem very significant − as a matter of fact, it may even seem slightly stupid ("look, Ma, I can access a variable without using its name!") − but references come in very handy when building reusable Perl objects. Let's take a quick example:

```
#!/usr/bin/perl # set variable $me = "popeye"; # set reference
$ref = \$me; # print values print "The value of \$me is
$me\n"; print "The value of \$ref is $ref\n";
```

In this case, we've first initialized a standard Perl scalar and then created a reference to it by prefixing it with a backslash[\]. Now, the reference and the variable itself are two different things − as you'll see when you run the script and get output that looks like this:

```
The value of $me is popeye The value of $ref is
SCALAR(0x80baa88)
```

Thus, the reference $ref is simply a pointer to an address in memory which stores the value of the variable $me. The word SCALAR that you see indicates the type of data, while the hexadecimal number following it is the actual memory address. Try memorizing it and see if you can say it backwards really fast.

Now, how about going full circle and using the reference to actually get the value of the variable and display it?

```
#!/usr/bin/perl # set variable $me = "popeye"; # set reference
$ref = \$me; # print values print "The value of \$ref is
$ref\n"; print "The value of the referenced variable is
$$ref\n";
```

And the output is:

```
The value of $ref is SCALAR(0x80baa88) The value of the
referenced variable is popeye
```

In this case, by placing a single $ before the variable containing the reference, you are able to get the original value of the reference (as opposed to some hexadecimal gibberish). Just so you know, this is referred to as "dereferencing" a variable. Geeks need new words for everything.

All Friends Here

You can also reference and dereference other types of Perl constructs − the next example shows you how to do this with arrays and hashes.

```
#!/usr/bin/perl # set array variable @friends = ("Rachel",
"Phoebe", "Monica", "Joey", "Chandler", "Ross"); # set hash
%animals = ('donald'=>'duck', 'mickey'=>'mouse',
'cheshire'=>'cat'); # set reference for array $ref =
\@friends; # set reference for hash $fer = \%animals; # print
values print "The value of \$ref is $ref\n"; print "The value
of \$fer is $fer\n"; print "\n"; print "The value of the first
array element is $$ref[0]\n"; print "The value for the key
\"mickey\" is $$fer{'mickey'}\n";
```

And the output is:

```
The value of $ref is ARRAY(0x80baa94) The value of $fer is
HASH(0x80bab18) The value of the first array element is Rachel
The value for the key "mickey" is mouse
```

Note how the data type of the referenced variable is clearly visible when you print the reference.

Don't just restrict yourself to variables, either − you can easily create a reference to a Perl subroutine like this:

```
#!/usr/bin/perl # set variable $me = "popeye"; # subroutine
sub whoami { print "My name is $me\n"; } # reference to
subroutine $ref = \ # print values print "The value of \$ref
is $ref\n"; # and run the subroutine via reference
```

And the output is:

```
The value of $ref is CODE(0x80baaac) My name is popeye
```

Signed, Anonymous

Just so you know, Perl also allows you to combine two steps into one by creating a reference directly, without using the backslash operator. For example, you could create an array reference like this (as in the example on the previous page):

```
# set array variable @friends = ("Rachel", "Phoebe", "Monica",
"Joey", "Chandler", "Ross"); # set reference for array $ref =
\@friends; # print values print "The value of \$ref is
$ref\n";
```

or like this

```
# set array reference $arrayref = ["Rachel", "Phoebe",
"Monica", "Joey", "Chandler", "Ross"]; # print values print
"The value of \$arrayref is $arrayref\n";
```

By enclosing the array elements in square braces, Perl allows you to create a reference directly (in a prime example of bad geek humour, this is known as an anonymous array, primarily because it doesn't have a name.) When you print the value of $arrayref, you'll see something like this:

```
The value of $arrayref is ARRAY(0x80b5670)
```

You can also create an "anonymous hash" by enclosing the key−value pairs in curly braces

```
$hashref = {'donald'=>'duck', 'mickey'=>'mouse',
'cheshire'=>'cat'};
```

and "anonymous subroutines" by using the "sub" keyword without a name after it.

```
$subref = sub { print "My name is $me\n"; };
```

The Copycat Crime

There's one important caveat when dealing with references, and it relates to copying them. Just as you can copy the value of one variable into another, Perl also allows you to copy references. The following example demonstrates this clearly.

```
#!/usr/bin/perl # set variable $icecream = "peach"; # set
reference $alpha = \$icecream; # copy reference $beta =
$alpha; # print state print "BEFORE\n"; print "icecream =
$icecream\n"; print "alpha(ref) = $alpha\n"; print "alpha(val)
= $$alpha\n"; print "beta(ref) = $beta\n"; print "beta(val) =
$$beta\n";
```

And the output is:

```
BEFORE icecream = peach alpha(ref) = SCALAR(0x80baa88)
alpha(val) = peach beta(ref) = SCALAR(0x80baa88) beta(val) =
peach
```

However, note what happens when you change the value of the copied reference.

```
#!/usr/bin/perl # set variable $icecream = "peach"; # set
reference $alpha = \$icecream; # copy reference $beta =
$alpha; # print state print "BEFORE\n"; print "icecream =
$icecream\n"; print "alpha(ref) = $alpha\n"; print "alpha(val)
= $$alpha\n"; print "beta(ref) = $beta\n"; print "beta(val) =
$$beta\n"; # change beta $$beta = "raspberry"; # print state
print "AFTER\n"; print "icecream = $icecream\n"; print
"alpha(ref) = $alpha\n"; print "alpha(val) = $$alpha\n"; print
"beta(ref) = $beta\n"; print "beta(val) = $$beta\n";
```

The output now makes for interesting reading.

```
BEFORE icecream = peach alpha(ref) = SCALAR(0x80baa88)
alpha(val) = peach beta(ref) = SCALAR(0x80baa88) beta(val) =
peach AFTER icecream = raspberry alpha(ref) =
SCALAR(0x80baa88) alpha(val) = raspberry beta(ref) =
SCALAR(0x80baa88) beta(val) = raspberry
```

The lesson here? Simple. When you copy variables, changing one has no effect on the other. But when you copy references, since a reference is a direct hook into memory, changing one of the copies affects the original value of the variable and also (obviously) the other copies of the reference.

And Now For Sumthing New...

Next up, packages. In Perl−lingo, a package can best be thought of as a self−contained unit of user−defined variables and subroutines, which can be re−used over and over again. In fact, this very reusability is the reason Perl programmers are so fond of packages, frequently using them to build "modules" which they then distribute to all and sundry.

Defining a package takes place via the "package" keyword, which is followed by the name of a package. The statement

```
package Sumthing;
```

would tell the Perl interpreter that the code following it belongs to the package named "Sumthing".

Once you've defined your package, you can add variable declarations and subroutine calls to it exactly as you would normally.

```
#!/usr/bin/perl package Sumthing; # subroutine sub add { $sum
= $alpha+$beta; print "Sum is $sum\n"; } # variables $alpha =
10; $beta = 4;
```

In the example above, the package named "Sumthing" contains a couple of variables, and a function that adds them up and prints them out.

Typically, the scope of the package definition extends to the end of the file, or until another "package" keyword is encountered.

A quick aside on modules here: once you've got your package together, you can convert it to a module by saving it to a file called "Sumthing.pm" (the .pm extension signifies a Perl Module, which is nothing but a package or collection of packages designed to be reusable). All Perl module files end in the extension .pm, and Perl module names typically begin with a capital letter.

```
# Sumthing.pm − a Perl module which performs addition package
Sumthing; # subroutine sub add { $sum = $alpha+$beta; print
"Sum is $sum\n"; } # variables $alpha = 10; $beta = 4; 1;
```

Note the

```
1;
```

at the end of the module file − this is typically done so that a use() or require() function call to the module succeeds.

More on how to use modules later.

Private Spaces

The important thing to remember here is that the variables and subroutines are private to a particular package, and cannot be accessed outside it. So, if you were to create another package named "Sumotherthing", as the following example demonstrates, "Sumotherthing" would not be able to access subroutines and variables from "Sumthing"...

```
#!/usr/bin/perl package Sumthing; # subroutine sub add { $sum
= $alpha+$beta; print "Sum is $sum\n"; } # variables $alpha =
10; $beta = 4; package Sumotherthing; # variables $alpha = 2;
$beta = 40; # subroutine sub multiply { $product =
$alpha*$beta; print "Product is $product\n"; } # print the
value of alpha # since no "package" statement has been
encountered, this statement will only # "see" the package
Sumotherthing print "The value of \$alpha is $alpha\n";
```

...unless it explicitly called variables and functions in "Sumthing" by preceding the variable/subroutine name with the package name.

```
#!/usr/bin/perl package Sumthing; # subroutine sub add { $sum
= $alpha+$beta; print "Sum is $sum\n"; } # variables $alpha =
10; $beta = 4; package Sumotherthing; # variables $alpha = 2;
$beta = 40; # subroutine sub multiply { $product =
$alpha*$beta; print "Product is $product\n"; } # print the
value of alpha # since no "package" statement has been
encountered, this statement will only # "see" the package
Sumotherthing print "The value of \$alpha is $alpha\n"; # now
switch to the other package and print package Sumthing; print
"The value of \$alpha is $alpha\n"; # you could also use the
double−colon notation to access constructs from # other
packages # this prints the value of "alpha" from Sumthing
print "The value of \$alpha from package #1 is " .
$Sumthing::alpha . "\n"; # while this prints the value of
"alpha" from Sumotherthing print "The value of \$alpha from
package #2 is " . $Sumotherthing::alpha . "\n"; # works for
subroutines too! # this runs the add() routine from Sumthing
# and this runs the multiply() routine from Sumotherthing
ultiply();
```

And the output is:

```
The value of $alpha is 2 The value of $alpha is 10 The value
of $alpha from package #1 is 10 The value of $alpha from
package #2 is 2 Sum is 14 Product is 80
```

Just so you know, packages aren't something to be afraid of. By default, all the Perl code you write resides within the domain of the uber−package, the one they call "main". The following snippet will demonstrate this:

```
#!/usr/bin/perl $language="Francais"; print "Parlez−vous
$main::language?\n";
```

In this case, since there is no mention of a specific package, the variable $language is created in the domain of the package "main".

Finally, the "package" keyword, by itself, indicates that subsequent variables will not be accepted by Perl unless they are explicitly preceded by a package name.

Bless() Me, For I Have Sinned!

Finally, it's time to get down and play in the mud with a few "objects". In Perl, an "object" is simply a reference that belongs to a specific package. And the method used to associate a reference with a specific package is referred to as "blessing", because the function used to perform the association is the bless() function.

For example,

```
package Easterbunny; sub new { my $this = {}; bless $this;
return $this; }
```

In this case, the first line within the subroutine creates an anonymous hash reference (this was covered a few pages back). The next line bless()es this reference and associates it with the package Easterbunny and sends it back as a return value.

Now, take a minute to read this definition of a constructor from the Perl manual. It says that "... a constructor is merely a subroutine that returns a reference to something blessed into a class, generally the class that the subroutine is defined in..."

In other words, you just wrote your first constructor.

Obviously, the new() constructor, or subroutine, above can do a whole lot more than just bless() references. For example, you could have it spit out a few choice insults to the poor programmer who decided to invoke it, as in the following example.

```
package Easterbunny; sub new { my $this = {}; print "Grow up.
Get a life. You suck.\n"; bless $this; return $this; }
```

Who says this job ain't fun?!

Old Bunnies For New

Finally, it's time to do something with the package you just created. If you haven't done it already, save the package from the last example to a file named "Easterbunny.pm" in your Perl library path with all the other .pm files (in case you don't know the location of those files, try checking the @INC variable). Remember that since this is now a module, you need to add the all−important

```
1;
```

to the end of the file. Then create a Perl script which looks like this:

```
#!/usr/bin/perl use Easterbunny; $bob = new Easterbunny;
```

The first line of the file looks for the Perl module "Easterbunny.pm" and reads it in for use.

The second line creates a new instance of Easterbunny by invoking the new() constructor, and assigns it to $bob.

```
You can also invoke the new() constructor in any of the
following ways: $bob = Easterbunny::new(); $bob =
Easterbunny−>new();
```

And the output will look something like this:

```
Grow up. Get a life. You suck.
```

And on that cheery note − goodbye and see you in a couple of weeks, when I'll be covering a few other OO concepts in Perl.