Object−Oriented Programming In Perl (part 2)

By Vikram Vaswani

Table of Contents

Tip O' The Iceberg

The last time out, I explained what references were, and showed you a few examples of how they can be used in your Perl scripts. I also explained the basics of Perl packages, and showed you how to build and use them in a script.

Of course, that was just the tip of the iceberg. In this second article, I'm going to delve deeper into the mysteries of Perl objects, and throw light on a few other things that may not yet be completely clear to you. Keep reading!

Building Your Very Own Automobile

Before we get into the nitty−gritty of object methods and properties, I want to take a minute to make sure that we're all clear on the basics. And I'm going to use an example I've used before to communicate the concept to those of you who may still not be too comfortable with it.

Take your car, the vehicle that transports you from place to place, and consider this, within the framework of OOP, as an "object". This object has certain basic properties − shape, size, colour − and certain basic functions − start, stop, accelerate and decelerate.

Now take it one step further. Every car can, in fact, be considered a subset of the class Automobile, which defines the basic characteristics and functions of every automobile on the planet. Once the class Automobile spawns a new car, or "object", its individual characteristics (color, shape) can be changed without affecting either the parent class or other spawned objects.

In Perl, a class is equivalent to a package − and the package might look something like this:

```
# this is Automobile.pm # a Perl module to create new
automobiles package Automobile; # constructor sub new { my
$this = {}; bless $this; return $this; } # subroutine (aka
"object method") sub init { # code to initialize variables
(aka "object properties") # this won't stay empty for long −
keep reading! } # a few other object methods sub start { #
some code } # a few other object methods sub stop { # some
code } sub accelerate { # some code } sub decelerate { # some
code } # remember to end the module with this 1;
```

In order to use this, you simply need to instantiate a new object in your Perl script by calling the constructor, like this:

```
#!/usr/bin/perl use Automobile; # create objects $his = new
Automobile; $hers = new Automobile; # run a few object methods
# let's take this baby out for a spin $his−>start(); # well,
hello there, pretty lady! # care to join me? $hers−>start(); #
this is fun! $his−>accelerate(); $hers−>accelerate(); # was
that a speed trap? $his−>decelerate(); $hers−>decelerate(); #
busted! $his−>stop(); $hers−>stop();
```

You're already familiar with the method of creating a new object − the example above simply demonstrates the manner in which other methods can be applied to the object. These methods are nothing more than subroutines within the package definition.

Ferrari Or Porsche?

Now let's toss something new into the mix, by adding a few object properties to the constructor.

```
# this is Automobile.pm # a Perl module to create new
automobiles # constructor sub new { my $this = {}; # object
properties $this−>{colour} = undef; $this−>{make} = undef;
$this−>{shape} = undef; bless $this; return $this; } # other
subroutines
```

Once you've added the variables to the constructor, you also need a way to initialize them − which is where the init() subroutine comes in. Let's add a bit of code to that, shall we?

```
# this is Automobile.pm # a Perl module to create new
automobiles # other subroutines sub init { my $this = shift;
$this−>{color} = shift; $this−>{make} = shift; $this−>{shape}
= shift; } # other subroutines
```

And now let's modify the test script − I'll use the init() method to create two completely different Automobiles.

```
#!/usr/bin/perl use Automobile; # create objects $his = new
Automobile; $his−>init("black","Ferrari","cigar"); $hers = new
Automobile; $hers−>init("silver","Porsche","torpedo"); # run a
few object methods # let's take this baby out for a spin
$his−>start(); # and so on
```

This time around, once an object is created, the init() method sets up some default properties like shape, make and colour. (You're probably wondering what all the shift()−ing is about in the subroutines above − don't worry about it for the moment, I've explained it all on the next page.)

And finally, how about doing something with the properties sent to the init() method? Let's add one more method, which lets you brag about your new car (and demonstrate how object properties can be used, too!)

```
# this is Automobile.pm # a Perl module to create new
automobiles # other subroutines sub brag { my $this = shift;
print "Guess what? I've just got myself a brand−spanking−new
$this−>{color} $this−>{make}, shaped like a $this−>{shape}.
Sure, I'll take you for a ride in it. Call me." } # other
subroutines And let's make a final modification to our test
script: #!/usr/bin/perl use Automobile; $his = new Automobile;
$his−>init("yellow", "Lamborghini", "missile"); $his−>brag():
```

And here's what you should see when you run it:

```
Guess what? I've just got myself a brand−spanking−new yellow
Lamborghini, shaped like a missile. Sure, I'll take you for a
ride in it. Call me.
```

And, as you'll see if you experiment a little, you can create as many Automobiles as you like, each with different properties, and the brag() method will display an appropriate message for each one. Ain't OOP cool?

Shift()ing Things Around

I promised I'd explain about the numerous calls to shift() in the init() subroutine (for those of you who don't know, the shift() function removes the first element of an array). It's actually pretty simple, and has to do with the difference between calling a subroutine and calling a method on an object.

Let's suppose you were to call the subroutine

```
Automobile::init("silver")
```

In this case, the @_ argument list to the subroutine init() would look like this:

```
@_ = ("silver");
```

However, if you were to call the method

```
$his = new Automobile; $his->init("silver");
```

the @_ argument list would store two items − an object reference and the argument itself, or

```
@_ = ($ref, "silver");
```

You need to use the shift() function to remove the reference from the argument list − which is why most object methods begin with

```
my $this = shift;
```

Tick, Tick

Now that you've (hopefully) understood the fundamental principles here, let's move on to something slightly more practical. This next package allows you to create different Clock objects, and initialize each one to a specific time zone. You could create a clock which displays local time, another which displays the time in New York, and a third which displays the time in London − all through the power of Perl objects. Let's take a look:

```
# Clock.pm # a module to create a simple clock # each Clock
object is initialized with two offsets (hours and minutes) #
indicating the difference between GMT and local time package
Clock; # constructor sub new { my $self = {}; # define
variables to store timezone offset from GMT, in hours and
minutes, # and city name $self->{offsetSign} = undef;
$self->{offsetH} = undef; $self->{offsetM} = undef;
$self->{city} = undef; bless($self); return $self; } # method
to set the offset hour and minute wrt GMT # this accepts an
offset direction # and two numerical parameters, hours and
minutes. sub set_offset { my $self = shift;
$self->{offsetSign} = shift; $self->{offsetH} = shift;
$self->{offsetM} = shift; $self->{city} = shift; } # method to
display current time, given offsets sub display { my $self =
shift; # use the gmtime() function, used to local convert time
to GMT # returns an array @GMTTime = gmtime(); $seconds =
@GMTTime[0]; $minutes = @GMTTime[1]; $hours = @GMTTime[2]; #
calculate time if($self->{offsetSign} eq '+') { # city time is
ahead of GMT $minutes += $self->{offsetM}; if($minutes > 60) {
$minutes -= 60; $hours++; } $hours += $self->{offsetH};
if($hours >= 24) { $hours -= 24; } } else { # city time is
behind GMT $seconds = 60 - $seconds; $minutes -=
$self->{offsetM}; if($minutes < 0) { $minutes += 60; $hour--;
} $hours -= $self->{offsetH}; if($hours < 0) { $hours = 24 +
$hours; } } # make it look pretty and display it
$self->{localTime} = $hours.":".$minutes.":".$seconds; print
"Local time in $self->{city} is $self->{localTime}\n"; } 1;
```

As you can see, we've got four variables: the name of the city, an indicator as to whether the city time is ahead of, or behind, Greenwich Mean Time, and the difference between the local time in that city and the standard GMT, in hours and minutes. Once these properties are made available to the package via the set_offset() method, the display() method takes over and performs some simple calculations to obtain the local time in that city.

And here's how you could use it:

```
#!/usr/bin/perl use Clock; $london = new Clock;
$london->set_offset("+", 0, 00, "London"); $london->display();
$bombay = new Clock; $bombay->set_offset("+", 5, 30,
```

```
"Bombay"); $bombay->display(); $sydney = new Clock;
$sydney->set_offset("+", 11, 00, "Sydney");
$sydney->display(); $us_ct = new Clock;
$us_ct->set_offset("-", 6, 00, "US/Central");
$us_ct->display(); And the output is: Local time in London is
8:10:1 Local time in Bombay is 13:40:1 Local time in Sydney is
19:10:1 Local time in US/Central is 2:10:59 It should be noted
here that it is also possible to directly adjust the offset
values without using the set_offset() method. For example, the
line $bombay->set_offset("+", 5, 30, "Bombay"); is technically
equivalent to $bombay->{offsetSign} = "+"; $bombay->{offsetH}
= 5; $bombay->{offsetM} = 30; $bombay->{city} = "Bombay";
```

Notice I said "technically". It is generally not advisable to do this, as it would violate the integrity of the object; the preferred method is always to use the methods exposed by the object to change object properties.

Just as an illustration − consider what would happen if the author of the Clock.pm module decided to change the variable name $offsetH to $OFFSETh − you would need to rework your test script as well, since you were directly referencing the variable $offsetH. If, on the other hand, you had used the set_offset() method, the changes to the variable name would be reflected in the set_offset() method by the author and your code would require no changes whatsoever.

By limiting yourself to exposed methods, you are provided with a level of protection which ensures that changes in the package code do not have repercussions on your code.

Alarm Bells

One of the main virtues of object−oriented programming is that it allows you to re−use existing objects, and add new capabilities to them − a feature referred to as "inheritance". By creating a new object which inherits the properties and methods of an existing object, developers can build on existing Perl packages, and reduce both development and testing time.

As an example, let's create a new package, AlarmClock, which inherits from the base class Clock.

```
package AlarmClock; # tell the new package to use the Clock.pm
module require Clock; # make methods available from "Clock"
@ISA=("Clock"); 1;
```

At this point, you should be able to do this

```
#!/usr/bin/perl require AlarmClock; $mumbai = new AlarmClock;
$mumbai−>set_offset(5,30, "Bombay"); $mumbai−>display;
```

and have the code work exactly as before, despite the fact that you are now using the AlarmClock package. This indicates that the package AlarmClock has successfully inherited the properties and methods of the package Clock. This is referred to as the "empty sub−class test" − essentially, a new class which functions exactly like the parent class, and can be used as a replacement for it (obviously, this is not very useful, and I'll be adding new methods to AlarmClock soon.)

The @ISA array that you see above is a special Perl array which indicates the class from which methods and properties are to be inherited − in this case, "Clock".

Now, how about adding a new method to the AlarmClock package?

```
package AlarmClock; # this tells it to use the Clock.pm module
require Clock; # this makes methods available @ISA="Clock"; #
simple subroutine to increase the value of the "hours"
variable by one sub adjustHoursByOne { my $self=shift;
$self−>{offsetH}++; } 1;
```

And let's add this to the test script as well:

```
#!/usr/bin/perl require AlarmClock; $mumbai = new AlarmClock;
$mumbai−>set_offset(5,30, "Bombay"); $mumbai−>display;
$mumbai−>adjustHoursByOne; $mumbai−>display;
```

And here's the output you'll probably see:

```
Local time in Bombay is 15:21:32 Can't locate object method
"adjustHoursByOne" via package "Clock"
```

You're probably thinking, "Duh? What's this all about?". Let me explain.

Bless() You...And You...And You Too!

In order for a child class to derive methods and properties correctly from the parent, the parent class' constructor needs to bless references into the child class correctly. If you take a look at the original constructor from Clock.pm, you'll see that it looks like this:

```
# constructor sub new { my $self = {}; # define variables to
store timezone offset from GMT, in hours and minutes, # and
city name $self->{offsetSign} = undef; $self->{offsetH} =
undef; $self->{offsetM} = undef; $self->{city} = undef;
bless($self); return $self; }
```

In this case, the bless() function blesses the reference $self directly into the same class...which creates a problem when the class is inherited. To solve this problem, you can bless the reference into any requesting class by using the extended form of bless(), or

```
bless($self, $class);
```

where $class is the name of the class $self is bless()ed into.

And where does $class come from? It's passed to the new() constructor at the time the object is created, and you can access it like this:

```
# constructor sub new { # get calling class name my $class =
shift; my $self = {}; # define variables to store timezone
offset from GMT, in hours and minutes, # and city name
$self->{offsetSign} = undef; $self->{offsetH} = undef;
$self->{offsetM} = undef; $self->{city} = undef; bless($self,
$class); return $self; }
```

And now, when you try the test script, you should see the following output:

```
Local time in Bombay is 13:51:40 Local time in Bombay is
14:51:40
```

Obviously, your child classes can have their own constructors, which you can use to define variables specific to the child class.

Manual Override

Now, while inheritance might not seem like a very big deal, it can blow up in your face at times. For example, look what happens if I add a new display() method to the AlarmClock package:

```
package AlarmClock; # lots of code sub display { print "This
clock is broken. Aargh!!\n"; } 1;
```

In this case, when I run the test script

```
#!/usr/bin/perl require AlarmClock; $mumbai = AlarmClock->new;
$mumbai->set_offset(5,30, "Bombay"); $mumbai->display;
```

the output reads

```
This clock is broken. Aargh!!
```

or, in other words, the display() method called belongs to AlarmClock, not Clock.

If I specifically want to run the display() method from the package Clock, I have to use an "override" like this:

```
#!/usr/bin/perl require AlarmClock; $mumbai = AlarmClock->new;
$mumbai->set_offset(5,30, "Bombay"); $mumbai->Clock::display;
```

By preceding the method with the package name, I can control which display() method is called.

Destroying Objects

And finally, destructors. In Perl, an object is automatically destroyed once the references to it are no longer in use, or when the Perl script completes execution. A destructor is a special function which allows you to execute commands immediately prior to the destruction of an object.

You do not usually need to define a destructor − but if you want to see what it looks like, take a look at this:

```
package AlarmClock; # lots of code sub DESTROY { print "I'm
outta here!\n"; } 1;
```

Note that a destructor must always be called DESTROY − yes, all caps!

And if you were to run the test script above again, your output would look like this:

```
Local time in Bombay is 15:33:12 I'm outta here!
```

Like all bad comedians, I know how to take a hint − I'm outta here too! Hopefully, this article made it a little easier for you to get your mind around the confusing morass of jargon that is Perl OOP. If you'd like to get deeper into the subject, I'd suggest that you take a look at the excellent "perltoot", "perlref" and "perlobj" man pages − or just write me and tell me you'd like to see more on the same topic. Ciao!