NAME

perlboot - Beginner's Object-Oriented Tutorial

DESCRIPTION

If you're not familiar with objects from other languages, some of the other Perl object documentation may be a little daunting, such as *perlobj*, a basic reference in using objects, and *perltoot*, which introduces readers to the peculiarities of Perl's object system in a tutorial way.

So, let's take a different approach, presuming no prior object experience. It helps if you know about subroutines (*perlsub*), references (*perlref* et. seq.), and packages (*perlmod*), so become familiar with those first if you haven't already.

If we could talk to the animals...

Let's let the animals talk for a moment:

```
sub Cow::speak {
  print "a Cow goes moooo!\n";
}
sub Horse::speak {
  print "a Horse goes neigh!\n";
}
sub Sheep::speak {
  print "a Sheep goes baaaah!\n";
}

Cow::speak;
Horse::speak;
Sheep::speak;
```

This results in:

```
a Cow goes moooo!
a Horse goes neigh!
a Sheep goes baaaah!
```

Nothing spectacular here. Simple subroutines, albeit from separate packages, and called using the full package name. So let's create an entire pasture:

```
# Cow::speak, Horse::speak, Sheep::speak as before
@pasture = qw(Cow Cow Horse Sheep Sheep);
foreach $animal (@pasture) {
  &{$animal."::speak"};
}
```

This results in:

```
a Cow goes moooo!
a Cow goes moooo!
a Horse goes neigh!
a Sheep goes baaaah!
a Sheep goes baaaah!
```

Wow. That symbolic coderef de-referencing there is pretty nasty. We're counting on `no strict refs` mode, certainly not recommended for larger programs. And why was that necessary? Because the name of the package seems to be inseparable from the name of the subroutine we want to invoke within that package.

Or is it?

Introducing the method invocation arrow

For now, let's say that `Class->method` invokes subroutine `method` in package `Class`. (Here, "Class" is used in its "category" meaning, not its "scholastic" meaning.) That's not completely accurate, but we'll do this one step at a time. Now let's use it like so:

```
# Cow::speak, Horse::speak, Sheep::speak as before
Cow->speak;
Horse->speak;
Sheep->speak;
```

And once again, this results in:

```
a Cow goes moooo!
a Horse goes neigh!
a Sheep goes baaaah!
```

That's not fun yet. Same number of characters, all constant, no variables. But yet, the parts are separable now. Watch:

```
$a = "Cow";
$a->speak; # invokes Cow->speak
```

Ahh! Now that the package name has been parted from the subroutine name, we can use a variable package name. And this time, we've got something that works even when `use strict refs` is enabled.

Invoking a barnyard

Let's take that new arrow invocation and put it back in the barnyard example:

```
sub Cow::speak {
  print "a Cow goes moooo!\n";
}
sub Horse::speak {
  print "a Horse goes neigh!\n";
}
sub Sheep::speak {
  print "a Sheep goes baaaah!\n";
}

@pasture = qw(Cow Cow Horse Sheep Sheep);
foreach $animal (@pasture) {
  $animal->speak;
}
```

There! Now we have the animals all talking, and safely at that, without the use of symbolic coderefs.

But look at all that common code. Each of the `speak` routines has a similar structure: a `print` operator and a string that contains common text, except for two of the words. It'd be nice if we could factor out the commonality, in case we decide later to change it all to `says` instead of `goes`.

And we actually have a way of doing that without much fuss, but we have to hear a bit more about what the method invocation arrow is actually doing for us.

The extra parameter of method invocation

The invocation of:

```
Class->method(@args)
```

attempts to invoke subroutine `Class::method` as:

```
Class::method("Class", @args);
```

(If the subroutine can't be found, "inheritance" kicks in, but we'll get to that later.) This means that we get the class name as the first parameter (the only parameter, if no arguments are given). So we can rewrite the `Sheep` speaking subroutine as:

```
sub Sheep::speak {
  my $class = shift;
  print "a $class goes baaaah!\n";
}
```

And the other two animals come out similarly:

```
sub Cow::speak {
  my $class = shift;
  print "a $class goes moooo!\n";
}
sub Horse::speak {
  my $class = shift;
  print "a $class goes neigh!\n";
}
```

In each case, `$class` will get the value appropriate for that subroutine. But once again, we have a lot of similar structure. Can we factor that out even further? Yes, by calling another method in the same class.

Calling a second method to simplify things

Let's call out from `speak` to a helper method called `sound`. This method provides the constant text for the sound itself.

```
{ package Cow;
  sub sound { "moooo" }
  sub speak {
    my $class = shift;
    print "a $class goes ", $class->sound, "!\n";
  }
}
```

Now, when we call `Cow->speak`, we get a `$class` of `Cow` in `speak`. This in turn selects the `Cow->sound` method, which returns `moooo`. But how different would this be for the `Horse`?

```
{ package Horse;
  sub sound { "neigh" }
  sub speak {
    my $class = shift;
    print "a $class goes ", $class->sound, "!\n";
  }
}
```

Only the name of the package and the specific sound change. So can we somehow share the definition for `speak` between the Cow and the Horse? Yes, with inheritance!

Inheriting the windpipes

We'll define a common subroutine package called `Animal`, with the definition for `speak`:

```
{ package Animal;
  sub speak {
  my $class = shift;
  print "a $class goes ", $class->sound, "!\n";
  }
}
```

Then, for each animal, we say it "inherits" from `Animal`, along with the animal-specific sound:

```
{ package Cow;
  @ISA = qw(Animal);
  sub sound { "moooo" }
}
```

Note the added `@ISA` array (pronounced "is a"). We'll get to that in a minute.

But what happens when we invoke `Cow->speak` now?

First, Perl constructs the argument list. In this case, it's just `Cow`. Then Perl looks for `Cow::speak`. But that's not there, so Perl checks for the inheritance array `@Cow::ISA`. It's there, and contains the single name `Animal`.

Perl next checks for `speak` inside `Animal` instead, as in `Animal::speak`. And that's found, so Perl invokes that subroutine with the already frozen argument list.

Inside the `Animal::speak` subroutine, `$class` becomes `Cow` (the first argument). So when we get to the step of invoking `$class->sound`, it'll be looking for `Cow->sound`, which gets it on the first try without looking at `@ISA`. Success!

A few notes about @ISA

This magical `@ISA` variable has declared that `Cow` "is a" `Animal`. Note that it's an array, not a simple single value, because on rare occasions, it makes sense to have more than one parent class searched for the missing methods.

If `Animal` also had an `@ISA`, then we'd check there too. The search is recursive, depth-first, left-to-right in each `@ISA` by default (see *mro* for alternatives). Typically, each `@ISA` has only one element (multiple elements means multiple inheritance and multiple headaches), so we get a nice tree of inheritance.

When we turn on `use strict`, we'll get complaints on `@ISA`, since it's not a variable containing an explicit package name, nor is it a lexical ("my") variable. We can't make it a lexical variable though (it has to belong to the package to be found by the inheritance mechanism), so there's a couple of straightforward ways to handle that.

The easiest is to just spell the package name out:

```
@Cow::ISA = qw(Animal);
```

Or declare it as package global variable:

```
package Cow;
our @ISA = qw(Animal);
```

Or allow it as an implicitly named package variable:

```
package Cow;
use vars qw(@ISA);
@ISA = qw(Animal);
```

If the `Animal` class comes from another (object-oriented) module, then just employ `use base` to specify that `Animal` should serve as the basis for the `Cow` class:

```
package Cow;
use base qw(Animal);
```

Now that's pretty darn simple!

Overriding the methods

Let's add a mouse, which can barely be heard:

```
# Animal package from before
{ package Mouse;
  @ISA = qw(Animal);
  sub sound { "squeak" }
  sub speak {
    my $class = shift;
    print "a $class goes ", $class->sound, "!\n";
    print "[but you can barely hear it!]\n";
  }
}

Mouse->speak;
```

which results in:

```
a Mouse goes squeak!
[but you can barely hear it!]
```

Here, `Mouse` has its own speaking routine, so `Mouse->speak` doesn't immediately invoke `Animal->speak`. This is known as "overriding". In fact, we don't even need to say that a `Mouse` is an `Animal` at all, because all of the methods needed for `speak` are completely defined for `Mouse`; this is known as "duck typing": "If it walks like a duck and quacks like a duck, I would call it a duck" (James Whitcomb). However, it would probably be beneficial to allow a closer examination to conclude that a `Mouse` is indeed an `Animal`, so it is actually better to define `Mouse` with `Animal` as its base (that is, it is better to "derive `Mouse` from `Animal`").

Moreover, this duplication of code could become a maintenance headache (though code-reuse is not actually a good reason for inheritance; good design practices dictate that a derived class should be usable wherever its base class is usable, which might not be the outcome if code-reuse is the sole criterion for inheritance. Just remember that a `Mouse` should always act like an `Animal`).

So, let's make `Mouse` an `Animal`!

The obvious solution is to invoke `Animal::speak` directly:

```
# Animal package from before
{ package Mouse;
  @ISA = qw(Animal);
  sub sound { "squeak" }
  sub speak {
```

```
    my $class = shift;
    Animal::speak($class);
    print "[but you can barely hear it!]\n";
  }
}
```

Note that we're using `Animal::speak`. If we were to invoke `Animal->speak` instead, the first parameter to `Animal::speak` would automatically be `"Animal"` rather than `"Mouse"`, so that the call to `$class->sound` in `Animal::speak` would become `Animal->sound` rather than `Mouse->sound`.

Also, without the method arrow `->`, it becomes necessary to specify the first parameter to `Animal::speak` ourselves, which is why `$class` is explicitly passed: `Animal::speak($class)`.

However, invoking `Animal::speak` directly is a mess: Firstly, it assumes that the `speak` method is a member of the `Animal` class; what if `Animal` actually inherits `speak` from its own base? Because we are no longer using `->` to access `speak`, the special method look up mechanism wouldn't be used, so `speak` wouldn't even be found!

The second problem is more subtle: `Animal` is now hardwired into the subroutine selection. Let's assume that `Animal::speak` does exist. What happens when, at a later time, someone expands the class hierarchy by having `Mouse` inherit from `Mus` instead of `Animal`. Unless the invocation of `Animal::speak` is also changed to an invocation of `Mus::speak`, centuries worth of taxonomical classification could be obliterated!

What we have here is a fragile or leaky abstraction; it is the beginning of a maintenance nightmare. What we need is the ability to search for the right method wih as few assumptions as possible.

Starting the search from a different place

A *better* solution is to tell Perl where in the inheritance chain to begin searching for `speak`. This can be achieved with a modified version of the method arrow `->`:

```
ClassName->FirstPlaceToLook::method
```

So, the improved `Mouse` class is:

```
# same Animal as before
{ package Mouse;
  # same @ISA, &sound as before
  sub speak {
    my $class = shift;
    $class->Animal::speak;
    print "[but you can barely hear it!]\n";
  }
}
```

Using this syntax, we start with `Animal` to find `speak`, and then use all of `Animal`'s inheritance chain if it is not found immediately. As usual, the first parameter to `speak` would be `$class`, so we no longer need to pass `$class` explicitly to `speak`.

But what about the second problem? We're still hardwiring `Animal` into the method lookup.

The SUPER way of doing things

If `Animal` is replaced with the special placeholder SUPER in that invocation, then the contents of `Mouse`'s `@ISA` are used for the search, beginning with `$ISA[0]`. So, all of the problems can be fixed as follows:

```
# same Animal as before
```

```
{ package Mouse;
  # same @ISA, &sound as before
  sub speak {
    my $class = shift;
    $class->SUPER::speak;
    print "[but you can barely hear it!]\n";
  }
}
```

In general, `SUPER::speak` means look in the current package's `@ISA` for a class that implements `speak`, and invoke the first one found. The placeholder is called `SUPER`, because many other languages refer to base classes as "*super*classes", and Perl likes to be eclectic.

Note that a call such as

```
$class->SUPER::method;
```

does *not* look in the `@ISA` of `$class` unless `$class` happens to be the current package.

Let's review...

So far, we've seen the method arrow syntax:

```
Class->method(@args);
```

or the equivalent:

```
$a = "Class";
$a->method(@args);
```

which constructs an argument list of:

```
("Class", @args)
```

and attempts to invoke:

```
Class::method("Class", @args);
```

However, if `Class::method` is not found, then `@Class::ISA` is examined (recursively) to locate a class (a package) that does indeed contain `method`, and that subroutine is invoked instead.

Using this simple syntax, we have class methods, (multiple) inheritance, overriding, and extending. Using just what we've seen so far, we've been able to factor out common code (though that's never a good reason for inheritance!), and provide a nice way to reuse implementations with variations.

Now, what about data?

A horse is a horse, of course of course, or is it?

Let's start with the code for the `Animal` class and the `Horse` class:

```
{ package Animal;
  sub speak {
    my $class = shift;
    print "a $class goes ", $class->sound, "!\n";
  }
}
{ package Horse;
  @ISA = qw(Animal);
```

```
  sub sound { "neigh" }
}
```

This lets us invoke `Horse->speak` to ripple upward to `Animal::speak`, calling back to `Horse::sound` to get the specific sound, and the output of:

```
a Horse goes neigh!
```

But all of our Horse objects would have to be absolutely identical. If we add a subroutine, all horses automatically share it. That's great for making horses the same, but how do we capture the distinctions of an individual horse? For example, suppose we want to give our first horse a name. There's got to be a way to keep its name separate from the other horses.

That is to say, we want particular instances of `Horse` to have different names.

In Perl, any reference can be an "instance", so let's start with the simplest reference that can hold a horse's name: a scalar reference.

```
my $name = "Mr. Ed";
my $horse = \$name;
```

So, now `$horse` is a reference to what will be the instance-specific data (the name). The final step is to turn this reference into a real instance of a `Horse` by using the special operator `bless`:

```
bless $horse, Horse;
```

This operator stores information about the package named `Horse` into the thing pointed at by the reference. At this point, we say `$horse` is an instance of `Horse`. That is, it's a specific horse. The reference is otherwise unchanged, and can still be used with traditional dereferencing operators.

Invoking an instance method

The method arrow can be used on instances, as well as classes (the names of packages). So, let's get the sound that `$horse` makes:

```
my $noise = $horse->sound("some", "unnecessary", "args");
```

To invoke `sound`, Perl first notes that `$horse` is a blessed reference (and thus an instance). It then constructs an argument list, as per usual.

Now for the fun part: Perl takes the class in which the instance was blessed, in this case `Horse`, and uses that class to locate the subroutine. In this case, `Horse::sound` is found directly (without using inheritance). In the end, it is as though our initial line were written as follows:

```
my $noise = Horse::sound($horse, "some", "unnecessary", "args");
```

Note that the first parameter here is still the instance, not the name of the class as before. We'll get `neigh` as the return value, and that'll end up as the `$noise` variable above.

If Horse::sound had not been found, we'd be wandering up the `@Horse::ISA` array, trying to find the method in one of the superclasses. The only difference between a class method and an instance method is whether the first parameter is an instance (a blessed reference) or a class name (a string).

Accessing the instance data

Because we get the instance as the first parameter, we can now access the instance-specific data. In this case, let's add a way to get at the name:

```
{ package Horse;
```

```
  @ISA = qw(Animal);
  sub sound { "neigh" }
  sub name {
    my $self = shift;
    $$self;
  }
}
```

Inside `Horse::name`, the `@_` array contains:

```
($horse, "some", "unnecessary", "args")
```

so the `shift` stores `$horse` into `$self`. Then, `$self` gets de-referenced with `$$self` as normal, yielding `"Mr. Ed"`.

It's traditional to `shift` the first parameter into a variable named `$self` for instance methods and into a variable named `$class` for class methods.

Then, the following line:

```
print $horse->name, " says ", $horse->sound, "\n";
```

outputs:

```
Mr. Ed says neigh.
```

How to build a horse

Of course, if we constructed all of our horses by hand, we'd most likely make mistakes from time to time. We're also violating one of the properties of object-oriented programming, in that the "inside guts" of a Horse are visible. That's good if you're a veterinarian, but not if you just like to own horses. So, let's have the Horse class handle the details inside a class method:

```
{ package Horse;
  @ISA = qw(Animal);
  sub sound { "neigh" }
  sub name {
    my $self = shift;     # instance method, so use $self
    $$self;
  }
  sub named {
    my $class = shift;    # class method, so use $class
    my $name = shift;
    bless \$name, $class;
  }
}
```

Now with the new `named` method, we can build a horse as follows:

```
my $horse = Horse->named("Mr. Ed");
```

Notice we're back to a class method, so the two arguments to `Horse::named` are `Horse` and `Mr. Ed`. The `bless` operator not only blesses `\$name`, it also returns that reference.

This `Horse::named` method is called a "constructor".

We've called the constructor `named` here, so that it quickly denotes the constructor's argument as the name for this particular `Horse`. You can use different constructors with different names for different

ways of "giving birth" to the object (like maybe recording its pedigree or date of birth). However, you'll find that most people coming to Perl from more limited languages use a single constructor named `new`, with various ways of interpreting the arguments to `new`. Either style is fine, as long as you document your particular way of giving birth to an object. (And you *were* going to do that, right?)

Inheriting the constructor

But was there anything specific to `Horse` in that method? No. Therefore, it's also the same recipe for building anything else that inherited from `Animal`, so let's put `name` and `named` there:

```
{ package Animal;
  sub speak {
    my $class = shift;
    print "a $class goes ", $class->sound, "!\n";
  }
  sub name {
    my $self = shift;
    $$self;
  }
  sub named {
    my $class = shift;
    my $name = shift;
    bless \$name, $class;
  }
}
{ package Horse;
  @ISA = qw(Animal);
  sub sound { "neigh" }
}
```

Ahh, but what happens if we invoke `speak` on an instance?

```
my $horse = Horse->named("Mr. Ed");
$horse->speak;
```

We get a debugging value:

```
a Horse=SCALAR(0xaca42ac) goes neigh!
```

Why? Because the `Animal::speak` routine is expecting a classname as its first parameter, not an instance. When the instance is passed in, we'll end up using a blessed scalar reference as a string, and that shows up as we saw it just now.

Making a method work with either classes or instances

All we need is for a method to detect if it is being called on a class or called on an instance. The most straightforward way is with the `ref` operator. This returns a string (the classname) when used on a blessed reference, and an empty string when used on a string (like a classname). Let's modify the `name` method first to notice the change:

```
sub name {
  my $either = shift;
  ref $either ? $$either : "Any $either";
}
```

Here, the `?:` operator comes in handy to select either the dereference or a derived string. Now we can use this with either an instance or a class. Note that I've changed the first parameter holder to `$either` to show that this is intended:

```
my $horse = Horse->named("Mr. Ed");
print Horse->name, "\n"; # prints "Any Horse\n"
print $horse->name, "\n"; # prints "Mr Ed.\n"
```

and now we'll fix `speak` to use this:

```
sub speak {
  my $either = shift;
  print $either->name, " goes ", $either->sound, "\n";
}
```

And since `sound` already worked with either a class or an instance, we're done!

Adding parameters to a method

Let's train our animals to eat:

```
{ package Animal;
  sub named {
    my $class = shift;
    my $name = shift;
    bless \$name, $class;
  }
  sub name {
    my $either = shift;
    ref $either ? $$either : "Any $either";
  }
  sub speak {
    my $either = shift;
    print $either->name, " goes ", $either->sound, "\n";
  }
  sub eat {
    my $either = shift;
    my $food = shift;
    print $either->name, " eats $food.\n";
  }
}
{ package Horse;
  @ISA = qw(Animal);
  sub sound { "neigh" }
}
{ package Sheep;
  @ISA = qw(Animal);
  sub sound { "baaaah" }
}
```

And now try it out:

```
my $horse = Horse->named("Mr. Ed");
$horse->eat("hay");
Sheep->eat("grass");
```

which prints:

```
Mr. Ed eats hay.
Any Sheep eats grass.
```

An instance method with parameters gets invoked with the instance, and then the list of parameters. So that first invocation is like:

```
Animal::eat($horse, "hay");
```

More interesting instances

What if an instance needs more data? Most interesting instances are made of many items, each of which can in turn be a reference or even another object. The easiest way to store these is often in a hash. The keys of the hash serve as the names of parts of the object (often called "instance variables" or "member variables"), and the corresponding values are, well, the values.

But how do we turn the horse into a hash? Recall that an object was any blessed reference. We can just as easily make it a blessed hash reference as a blessed scalar reference, as long as everything that looks at the reference is changed accordingly.

Let's make a sheep that has a name and a color:

```
my $bad = bless { Name => "Evil", Color => "black" }, Sheep;
```

so `$bad->{Name}` has `Evil`, and `$bad->{Color}` has `black`. But we want to make `$bad->name` access the name, and that's now messed up because it's expecting a scalar reference. Not to worry, because that's pretty easy to fix up.

One solution is to override `Animal::name` and `Animal::named` by defining them anew in `Sheep`, but then any methods added later to `Animal` might still mess up, and we'd have to override all of those too. Therefore, it's never a good idea to define the data layout in a way that's different from the data layout of the base classes. In fact, it's a good idea to use blessed hash references in all cases. Also, this is why it's important to have constructors do the low-level work. So, let's redefine `Animal`:

```
## in Animal
sub name {
  my $either = shift;
  ref $either ? $either->{Name} : "Any $either";
}
sub named {
  my $class = shift;
  my $name = shift;
  my $self = { Name => $name };
  bless $self, $class;
}
```

Of course, we still need to override `named` in order to handle constructing a `Sheep` with a certain color:

```
## in Sheep
sub named {
  my ($class, $name) = @_;
  my $self = $class->SUPER::named(@_);
  $$self{Color} = $class->default_color;
  $self
}
```

(Note that `@_` contains the parameters to `named`.)

What's this `default_color`? Well, if `named` has only the name, we still need to set a color, so we'll have a class-specific default color. For a sheep, we might define it as white:

```
## in Sheep
sub default_color { "white" }
```

Now:

```
my $sheep = Sheep->named("Bad");
print $sheep->{Color}, "\n";
```

outputs:

```
white
```

Now, there's nothing particularly specific to `Sheep` when it comes to color, so let's remove `Sheep::named` and implement `Animal::named` to handle color instead:

```
## in Animal
sub named {
  my ($class, $name) = @_;
  my $self = { Name => $name, Color => $class->default_color };
  bless $self, $class;
}
```

And then to keep from having to define `default_color` for each additional class, we'll define a method that serves as the "default default" directly in `Animal`:

```
## in Animal
sub default_color { "brown" }
```

Of course, because `name` and `named` were the only methods that referenced the "structure" of the object, the rest of the methods can remain the same, so `speak` still works as before.

A horse of a different color

But having all our horses be brown would be boring. So let's add a method or two to get and set the color.

```
## in Animal
sub color {
  $_[0]->{Color}
}
sub set_color {
  $_[0]->{Color} = $_[1];
}
```

Note the alternate way of accessing the arguments: `$_[0]` is used in-place, rather than with a `shift`. (This saves us a bit of time for something that may be invoked frequently.) And now we can fix that color for Mr. Ed:

```
my $horse = Horse->named("Mr. Ed");
$horse->set_color("black-and-white");
print $horse->name, " is colored ", $horse->color, "\n";
```

which results in:

```
Mr. Ed is colored black-and-white
```

Summary

So, now we have class methods, constructors, instance methods, instance data, and even accessors. But that's still just the beginning of what Perl has to offer. We haven't even begun to talk about accessors that double as getters and setters, destructors, indirect object notation, overloading, "isa" and "can" tests, the `UNIVERSAL` class, and so on. That's for the rest of the Perl documentation to cover. Hopefully, this gets you started, though.

SEE ALSO

For more information, see *perlobj* (for all the gritty details about Perl objects, now that you've seen the basics), *perltoot* (the tutorial for those who already know objects), *perltooc* (dealing with class data), *perlbot* (for some more tricks), and books such as Damian Conway's excellent *Object Oriented Perl*.

Some modules which might prove interesting are Class::Accessor, Class::Class, Class::Contract, Class::Data::Inheritable, Class::MethodMaker and Tie::SecureHash

COPYRIGHT