## NAME

perlobj - Perl objects

## DESCRIPTION

First you need to understand what references are in Perl. See *perlref* for that. Second, if you still find the following reference work too complicated, a tutorial on object-oriented programming in Perl can be found in *perltoot* and *perltooc*.

If you're still with us, then here are three very simple definitions that you should find reassuring.

1. An object is simply a reference that happens to know which class it belongs to.
2. A class is simply a package that happens to provide methods to deal with object references.
3. A method is simply a subroutine that expects an object reference (or a package name, for class methods) as the first argument.

We'll cover these points now in more depth.

### An Object is Simply a Reference

Unlike say C++, Perl doesn't provide any special syntax for constructors. 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. Here is a typical constructor:

```
    package Critter;
    sub new { bless {} }
```

That word `new` isn't special. You could have written a construct this way, too:

```
    package Critter;
    sub spawn { bless {} }
```

This might even be preferable, because the C++ programmers won't be tricked into thinking that `new` works in Perl as it does in C++. It doesn't. We recommend that you name your constructors whatever makes sense in the context of the problem you're solving. For example, constructors in the Tk extension to Perl are named after the widgets they create.

One thing that's different about Perl constructors compared with those in C++ is that in Perl, they have to allocate their own memory. (The other things is that they don't automatically call overridden base-class constructors.) The `{}` allocates an anonymous hash containing no key/value pairs, and returns it The bless() takes that reference and tells the object it references that it's now a Critter, and returns the reference. This is for convenience, because the referenced object itself knows that it has been blessed, and the reference to it could have been returned directly, like this:

```
    sub new {
 my $self = {};
 bless $self;
 return $self;
    }
```

You often see such a thing in more complicated constructors that wish to call methods in the class as part of the construction:

```
    sub new {
 my $self = {};
 bless $self;
 $self->initialize();
 return $self;
```

```
    }
```

If you care about inheritance (and you should; see *"Modules: Creation, Use, and Abuse" in perlmodlib* ), then you want to use the two-arg form of bless so that your constructors may be inherited:

```
    sub new {
my $class = shift;
my $self = {};
bless $self, $class;
$self->initialize();
return $self;
    }
```

Or if you expect people to call not just `CLASS->new()` but also `$obj->new()`, then use something like the following. (Note that using this to call new() on an instance does not automatically perform any copying. If you want a shallow or deep copy of an object, you'll have to specifically allow for that.) The initialize() method used will be of whatever $class we blessed the object into:

```
    sub new {
my $this = shift;
my $class = ref($this) || $this;
my $self = {};
bless $self, $class;
$self->initialize();
return $self;
    }
```

Within the class package, the methods will typically deal with the reference as an ordinary reference. Outside the class package, the reference is generally treated as an opaque value that may be accessed only through the class's methods.

Although a constructor can in theory re-bless a referenced object currently belonging to another class, this is almost certainly going to get you into trouble. The new class is responsible for all cleanup later. The previous blessing is forgotten, as an object may belong to only one class at a time. (Although of course it's free to inherit methods from many classes.) If you find yourself having to do this, the parent class is probably misbehaving, though.

A clarification: Perl objects are blessed. References are not. Objects know which package they belong to. References do not. The bless() function uses the reference to find the object. Consider the following example:

```
    $a = {};
    $b = $a;
    bless $a, BLAH;
    print "\$b is a ", ref($b), "\n";
```

This reports $b as being a BLAH, so obviously bless() operated on the object and not on the reference.

### A Class is Simply a Package

Unlike say C++, Perl doesn't provide any special syntax for class definitions. You use a package as a class by putting method definitions into the class.

There is a special array within each package called @ISA, which says where else to look for a method if you can't find it in the current package. This is how Perl implements inheritance. Each element of the @ISA array is just the name of another package that happens to be a class package. The classes are searched for missing methods in depth-first, left-to-right order by default (see *mro* for

alternative search order and other in-depth information). The classes accessible through @ISA are known as base classes of the current class.

All classes implicitly inherit from class `UNIVERSAL` as their last base class. Several commonly used methods are automatically supplied in the UNIVERSAL class; see *Default UNIVERSAL methods* for more details.

If a missing method is found in a base class, it is cached in the current class for efficiency. Changing @ISA or defining new subroutines invalidates the cache and causes Perl to do the lookup again.

If neither the current class, its named base classes, nor the UNIVERSAL class contains the requested method, these three places are searched all over again, this time looking for a method named AUTOLOAD(). If an AUTOLOAD is found, this method is called on behalf of the missing method, setting the package global $AUTOLOAD to be the fully qualified name of the method that was intended to be called.

If none of that works, Perl finally gives up and complains.

If you want to stop the AUTOLOAD inheritance say simply

```
sub AUTOLOAD;
```

and the call will die using the name of the sub being called.

Perl classes do method inheritance only. Data inheritance is left up to the class itself. By and large, this is not a problem in Perl, because most classes model the attributes of their object using an anonymous hash, which serves as its own little namespace to be carved up by the various classes that might want to do something with the object. The only problem with this is that you can't sure that you aren't using a piece of the hash that isn't already used. A reasonable workaround is to prepend your fieldname in the hash with the package name.

```
    sub bump {
my $self = shift;
$self->{ __PACKAGE__ . ".count"}++;
    }
```

## A Method is Simply a Subroutine

Unlike say C++, Perl doesn't provide any special syntax for method definition. (It does provide a little syntax for method invocation though. More on that later.) A method expects its first argument to be the object (reference) or package (string) it is being invoked on. There are two ways of calling methods, which we'll call class methods and instance methods.

A class method expects a class name as the first argument. It provides functionality for the class as a whole, not for any individual object belonging to the class. Constructors are often class methods, but see *perltoot* and *perltooc* for alternatives. Many class methods simply ignore their first argument, because they already know what package they're in and don't care what package they were invoked via. (These aren't necessarily the same, because class methods follow the inheritance tree just like ordinary instance methods.) Another typical use for class methods is to look up an object by name:

```
    sub find {
my ($class, $name) = @_;
$objtable{$name};
    }
```

An instance method expects an object reference as its first argument. Typically it shifts the first argument into a "self" or "this" variable, and then uses that as an ordinary reference.

```
    sub display {
```

```
my $self = shift;
my @keys = @_ ? @_ : sort keys %$self;
foreach $key (@keys) {
    print "\t$key => $self->{$key}\n";
}
   }
```

## Method Invocation

For various historical and other reasons, Perl offers two equivalent ways to write a method call. The simpler and more common way is to use the arrow notation:

```
my $fred = Critter->find("Fred");
$fred->display("Height", "Weight");
```

You should already be familiar with the use of the `->` operator with references. In fact, since `$fred` above is a reference to an object, you could think of the method call as just another form of dereferencing.

Whatever is on the left side of the arrow, whether a reference or a class name, is passed to the method subroutine as its first argument. So the above code is mostly equivalent to:

```
my $fred = Critter::find("Critter", "Fred");
Critter::display($fred, "Height", "Weight");
```

How does Perl know which package the subroutine is in? By looking at the left side of the arrow, which must be either a package name or a reference to an object, i.e. something that has been blessed to a package. Either way, that's the package where Perl starts looking. If that package has no subroutine with that name, Perl starts looking for it in any base classes of that package, and so on.

If you need to, you *can* force Perl to start looking in some other package:

```
my $barney = MyCritter->Critter::find("Barney");
$barney->Critter::display("Height", "Weight");
```

Here `MyCritter` is presumably a subclass of `Critter` that defines its own versions of find() and display(). We haven't specified what those methods do, but that doesn't matter above since we've forced Perl to start looking for the subroutines in `Critter`.

As a special case of the above, you may use the `SUPER` pseudo-class to tell Perl to start looking for the method in the packages named in the current class's `@ISA` list.

```
package MyCritter;
use base 'Critter';    # sets @MyCritter::ISA = ('Critter');

sub display {
    my ($self, @args) = @_;
    $self->SUPER::display("Name", @args);
}
```

It is important to note that `SUPER` refers to the superclass(es) of the *current package* and not to the superclass(es) of the object. Also, the `SUPER` pseudo-class can only currently be used as a modifier to a method name, but not in any of the other ways that class names are normally used, eg:

```
something->SUPER::method(...); # OK
SUPER::method(...);   # WRONG
SUPER->method(...);   # WRONG
```

Instead of a class name or an object reference, you can also use any expression that returns either of those on the left side of the arrow. So the following statement is valid:

```
Critter->find("Fred")->display("Height", "Weight");
```

and so is the following:

```
my $fred = (reverse "rettirC")->find(reverse "derF");
```

The right side of the arrow typically is the method name, but a simple scalar variable containing either the method name or a subroutine reference can also be used.

## Indirect Object Syntax

The other way to invoke a method is by using the so-called "indirect object" notation. This syntax was available in Perl 4 long before objects were introduced, and is still used with filehandles like this:

```
print STDERR "help!!!\n";
```

The same syntax can be used to call either object or class methods.

```
my $fred = find Critter "Fred";
display $fred "Height", "Weight";
```

Notice that there is no comma between the object or class name and the parameters. This is how Perl can tell you want an indirect method call instead of an ordinary subroutine call.

But what if there are no arguments? In that case, Perl must guess what you want. Even worse, it must make that guess *at compile time*. Usually Perl gets it right, but when it doesn't you get a function call compiled as a method, or vice versa. This can introduce subtle bugs that are hard to detect.

For example, a call to a method `new` in indirect notation -- as C++ programmers are wont to make -- can be miscompiled into a subroutine call if there's already a `new` function in scope. You'd end up calling the current package's `new` as a subroutine, rather than the desired class's method. The compiler tries to cheat by remembering bareword `requires`, but the grief when it messes up just isn't worth the years of debugging it will take you to track down such subtle bugs.

There is another problem with this syntax: the indirect object is limited to a name, a scalar variable, or a block, because it would have to do too much lookahead otherwise, just like any other postfix dereference in the language. (These are the same quirky rules as are used for the filehandle slot in functions like `print` and `printf`.) This can lead to horribly confusing precedence problems, as in these next two lines:

```
move $obj->{FIELD};             # probably wrong!
move $ary[$i];                  # probably wrong!
```

Those actually parse as the very surprising:

```
$obj->move->{FIELD};            # Well, lookee here
$ary->move([$i]);               # Didn't expect this one, eh?
```

Rather than what you might have expected:

```
$obj->{FIELD}->move();          # You should be so lucky.
$ary[$i]->move;                 # Yeah, sure.
```

To get the correct behavior with indirect object syntax, you would have to use a block around the indirect object:

```
move {$obj->{FIELD}};
move {$ary[$i]};
```

Even then, you still have the same potential problem if there happens to be a function named `move` in the current package. **The `->` notation suffers from neither of these disturbing ambiguities, so we recommend you use it exclusively.** However, you may still end up having to read code using the indirect object notation, so it's important to be familiar with it.

## Default UNIVERSAL methods

The `UNIVERSAL` package automatically contains the following methods that are inherited by all other classes:

isa(CLASS)

`isa` returns *true* if its object is blessed into a subclass of `CLASS`

You can also call `UNIVERSAL::isa` as a subroutine with two arguments. Of course, this will do the wrong thing if someone has overridden `isa` in a class, so don't do it.

If you need to determine whether you've received a valid invocant, use the `blessed` function from *Scalar::Util*:

```
if (blessed($ref) && $ref->isa( 'Some::Class')) {
    # ...
}
```

`blessed` returns the name of the package the argument has been blessed into, or `undef`.

can(METHOD)

`can` checks to see if its object has a method called `METHOD`, if it does then a reference to the sub is returned, if it does not then *undef* is returned.

`UNIVERSAL::can` can also be called as a subroutine with two arguments. It'll always return *undef* if its first argument isn't an object or a class name. The same caveats for calling `UNIVERSAL::isa` directly apply here, too.

VERSION( [NEED] )

`VERSION` returns the version number of the class (package). If the NEED argument is given then it will check that the current version (as defined by the $VERSION variable in the given package) not less than NEED; it will die if this is not the case. This method is normally called as a class method. This method is called automatically by the `VERSION` form of `use`.

```
use A 1.2 qw(some imported subs);
# implies:
A->VERSION(1.2);
```

**NOTE:** `can` directly uses Perl's internal code for method lookup, and `isa` uses a very similar method and cache-ing strategy. This may cause strange effects if the Perl code dynamically changes @ISA in any package.

You may add other methods to the UNIVERSAL class via Perl or XS code. You do not need to `use` `UNIVERSAL` to make these methods available to your program (and you should not do so).

## Destructors

When the last reference to an object goes away, the object is automatically destroyed. (This may even be after you exit, if you've stored references in global variables.) If you want to capture control just before the object is freed, you may define a DESTROY method in your class. It will automatically be called at the appropriate moment, and you can do any extra cleanup you need to do. Perl passes a reference to the object under destruction as the first (and only) argument. Beware that the reference is a read-only value, and cannot be modified by manipulating `$_[0]` within the destructor. The object

itself (i.e. the thingy the reference points to, namely `${$_[0]}`, `@{$_[0]}`, `%{$_[0]}` etc.) is not similarly constrained.

Since DESTROY methods can be called at unpredictable times, it is important that you localise any global variables that the method may update. In particular, localise `$@` if you use `eval {}` and localise `$?` if you use `system` or backticks.

If you arrange to re-bless the reference before the destructor returns, perl will again call the DESTROY method for the re-blessed object after the current one returns. This can be used for clean delegation of object destruction, or for ensuring that destructors in the base classes of your choosing get called. Explicitly calling DESTROY is also possible, but is usually never needed.

Do not confuse the previous discussion with how objects *CONTAINED* in the current one are destroyed. Such objects will be freed and destroyed automatically when the current object is freed, provided no other references to them exist elsewhere.

## Summary

That's about all there is to it. Now you need just to go off and buy a book about object-oriented design methodology, and bang your forehead with it for the next six months or so.

## Two-Phased Garbage Collection

For most purposes, Perl uses a fast and simple, reference-based garbage collection system. That means there's an extra dereference going on at some level, so if you haven't built your Perl executable using your C compiler's `-O` flag, performance will suffer. If you *have* built Perl with `cc -O`, then this probably won't matter.

A more serious concern is that unreachable memory with a non-zero reference count will not normally get freed. Therefore, this is a bad idea:

```
    {
my $a;
$a = \$a;
    }
```

Even thought $a *should* go away, it can't. When building recursive data structures, you'll have to break the self-reference yourself explicitly if you don't care to leak. For example, here's a self-referential node such as one might use in a sophisticated tree structure:

```
    sub new_node {
my $class = shift;
my $node  = {};
$node->{LEFT} = $node->{RIGHT} = $node;
$node->{DATA} = [ @_ ];
return bless $node => $class;
    }
```

If you create nodes like that, they (currently) won't go away unless you break their self reference yourself. (In other words, this is not to be construed as a feature, and you shouldn't depend on it.)

Almost.

When an interpreter thread finally shuts down (usually when your program exits), then a rather costly but complete mark-and-sweep style of garbage collection is performed, and everything allocated by that thread gets destroyed. This is essential to support Perl as an embedded or a multithreadable language. For example, this program demonstrates Perl's two-phased garbage collection:

```
    #!/usr/bin/perl
    package Subtle;
```

```
   sub new {
my $test;
$test = \$test;
warn "CREATING " . \$test;
return bless \$test;
   }

   sub DESTROY {
my $self = shift;
warn "DESTROYING $self";
   }

   package main;

   warn "starting program";
   {
my $a = Subtle->new;
my $b = Subtle->new;
$$a = 0;  # break selfref
warn "leaving block";
   }

   warn "just exited block";
   warn "time to die...";
   exit;
```

When run as */foo/test*, the following output is produced:

```
starting program at /foo/test line 18.
CREATING SCALAR(0x8e5b8) at /foo/test line 7.
CREATING SCALAR(0x8e57c) at /foo/test line 7.
leaving block at /foo/test line 23.
DESTROYING Subtle=SCALAR(0x8e5b8) at /foo/test line 13.
just exited block at /foo/test line 26.
time to die... at /foo/test line 27.
DESTROYING Subtle=SCALAR(0x8e57c) during global destruction.
```

Notice that "global destruction" bit there? That's the thread garbage collector reaching the unreachable.

Objects are always destructed, even when regular refs aren't. Objects are destructed in a separate pass before ordinary refs just to prevent object destructors from using refs that have been themselves destructed. Plain refs are only garbage-collected if the destruct level is greater than 0. You can test the higher levels of global destruction by setting the PERL_DESTRUCT_LEVEL environment variable, presuming `-DDEBUGGING` was enabled during perl build time. See *"PERL_DESTRUCT_LEVEL" in perlhack* for more information.

A more complete garbage collection strategy will be implemented at a future date.

In the meantime, the best solution is to create a non-recursive container class that holds a pointer to the self-referential data structure. Define a DESTROY method for the containing object's class that manually breaks the circularities in the self-referential structure.

## SEE ALSO

A kinder, gentler tutorial on object-oriented programming in Perl can be found in *perltoot*, *perlboot* and *perltooc*. You should also check out *perlbot* for other object tricks, traps, and tips, as well as *perlmodlib* for some style guides on constructing both modules and classes.