NAME

perlobj - Perl object reference

DESCRIPTION

This document provides a reference for Perl's object orientation features. If you're looking for an introduction to object-oriented programming in Perl, please see *perlootut*.

In order to understand Perl objects, you first need to understand references in Perl. See *perlref* for details.

This document describes all of Perl's object-oriented (OO) features from the ground up. If you're just looking to write some object-oriented code of your own, you are probably better served by using one of the object systems from CPAN described in *perlootut*.

If you're looking to write your own object system, or you need to maintain code which implements objects from scratch then this document will help you understand exactly how Perl does object orientation.

There are a few basic principles which define object oriented Perl:

1. An object is simply a data structure that knows to which class it belongs.
2. A class is simply a package. A class provides methods that expect to operate on objects.
3. A method is simply a subroutine that expects a reference to an object (or a package name, for class methods) as the first argument.

Let's look at each of these principles in depth.

An Object is Simply a Data Structure

Unlike many other languages which support object orientation, Perl does not provide any special syntax for constructing an object. Objects are merely Perl data structures (hashes, arrays, scalars, filehandles, etc.) that have been explicitly associated with a particular class.

That explicit association is created by the built-in `bless` function, which is typically used within the *constructor* subroutine of the class.

Here is a simple constructor:

```
package File;

sub new {
    my $class = shift;

    return bless {}, $class;
}
```

The name `new` isn't special. We could name our constructor something else:

```
package File;

sub load {
    my $class = shift;

    return bless {}, $class;
}
```

The modern convention for OO modules is to always use `new` as the name for the constructor, but

there is no requirement to do so. Any subroutine that blesses a data structure into a class is a valid constructor in Perl.

In the previous examples, the `{}` code creates a reference to an empty anonymous hash. The `bless` function then takes that reference and associates the hash with the class in `$class`. In the simplest case, the `$class` variable will end up containing the string "File".

We can also use a variable to store a reference to the data structure that is being blessed as our object:

```
sub new {
    my $class = shift;

    my $self = {};
    bless $self, $class;

    return $self;
}
```

Once we've blessed the hash referred to by `$self` we can start calling methods on it. This is useful if you want to put object initialization in its own separate method:

```
sub new {
    my $class = shift;

    my $self = {};
    bless $self, $class;

    $self->_initialize();

    return $self;
}
```

Since the object is also a hash, you can treat it as one, using it to store data associated with the object. Typically, code inside the class can treat the hash as an accessible data structure, while code outside the class should always treat the object as opaque. This is called **encapsulation**. Encapsulation means that the user of an object does not have to know how it is implemented. The user simply calls documented methods on the object.

Note, however, that (unlike most other OO languages) Perl does not ensure or enforce encapsulation in any way. If you want objects to actually *be* opaque you need to arrange for that yourself. This can be done in a varierty of ways, including using *Inside-Out objects* or modules from CPAN.

Objects Are Blessed; Variables Are Not

When we bless something, we are not blessing the variable which contains a reference to that thing, nor are we blessing the reference that the variable stores; we are blessing the thing that the variable refers to (sometimes known as the *referent*). This is best demonstrated with this code:

```
use Scalar::Util 'blessed';

my $foo = {};
my $bar = $foo;

bless $foo, 'Class';
print blessed( $bar );      # prints "Class"
```

```
$bar = "some other value";
print blessed( $bar );      # prints undef
```

When we call `bless` on a variable, we are actually blessing the underlying data structure that the variable refers to. We are not blessing the reference itself, nor the variable that contains that reference. That's why the second call to `blessed( $bar )` returns false. At that point `$bar` is no longer storing a reference to an object.

You will sometimes see older books or documentation mention "blessing a reference" or describe an object as a "blessed reference", but this is incorrect. It isn't the reference that is blessed as an object; it's the thing the reference refers to (i.e. the referent).

A Class is Simply a Package

Perl does not provide any special syntax for class definitions. A package is simply a namespace containing variables and subroutines. The only difference is that in a class, the subroutines may expect a reference to an object or the name of a class as the first argument. This is purely a matter of convention, so a class may contain both methods and subroutines which *don't* operate on an object or class.

Each package contains a special array called `@ISA`. The `@ISA` array contains a list of that class's parent classes, if any. This array is examined when Perl does method resolution, which we will cover later.

It is possible to manually set `@ISA`, and you may see this in older Perl code. Much older code also uses the *base* pragma. For new code, we recommend that you use the *parent* pragma to declare your parents. This pragma will take care of setting `@ISA`. It will also load the parent classes and make sure that the package doesn't inherit from itself.

However the parent classes are set, the package's `@ISA` variable will contain a list of those parents. This is simply a list of scalars, each of which is a string that corresponds to a package name.

All classes inherit from the *UNIVERSAL* class implicitly. The *UNIVERSAL* class is implemented by the Perl core, and provides several default methods, such as `isa()`, `can()`, and `VERSION()`. The `UNIVERSAL` class will *never* appear in a package's `@ISA` variable.

Perl *only* provides method inheritance as a built-in feature. Attribute inheritance is left up the class to implement. See the *Writing Accessors* section for details.

A Method is Simply a Subroutine

Perl does not provide any special syntax for defining a method. A method is simply a regular subroutine, and is declared with `sub`. What makes a method special is that it expects to receive either an object or a class name as its first argument.

Perl *does* provide special syntax for method invocation, the `->` operator. We will cover this in more detail later.

Most methods you write will expect to operate on objects:

```
sub save {
    my $self = shift;

    open my $fh, '>', $self->path() or die $!;
    print {$fh} $self->data()       or die $!;
    close $fh                       or die $!;
}
```

Method Invocation

Calling a method on an object is written as `$object->method`.

The left hand side of the method invocation (or arrow) operator is the object (or class name), and the right hand side is the method name.

```
my $pod = File->new( 'perlobj.pod', $data );
$pod->save();
```

The `->` syntax is also used when dereferencing a reference. It looks like the same operator, but these are two different operations.

When you call a method, the thing on the left side of the arrow is passed as the first argument to the method. That means when we call `Critter->new()`, the `new()` method receives the string `"Critter"` as its first argument. When we call `$fred->speak()`, the `$fred` variable is passed as the first argument to `speak()`.

Just as with any Perl subroutine, all of the arguments passed in `@_` are aliases to the original argument. This includes the object itself. If you assign directly to `$_[0]` you will change the contents of the variable that holds the reference to the object. We recommend that you don't do this unless you know exactly what you're doing.

Perl knows what package the method is in by looking at the left side of the arrow. If the left hand side is a package name, it looks for the method in that package. If the left hand side is an object, then Perl looks for the method in the package that the object has been blessed into.

If the left hand side is neither a package name nor an object, then the method call will cause an error, but see the section on *Method Call Variations* for more nuances.

Inheritance

We already talked about the special `@ISA` array and the *parent* pragma.

When a class inherits from another class, any methods defined in the parent class are available to the child class. If you attempt to call a method on an object that isn't defined in its own class, Perl will also look for that method in any parent classes it may have.

```
package File::MP3;
use parent 'File';    # sets @File::MP3::ISA = ('File');

my $mp3 = File::MP3->new( 'Andvari.mp3', $data );
$mp3->save();
```

Since we didn't define a `save()` method in the `File::MP3` class, Perl will look at the `File::MP3` class's parent classes to find the `save()` method. If Perl cannot find a `save()` method anywhere in the inheritance hierarchy, it will die.

In this case, it finds a `save()` method in the `File` class. Note that the object passed to `save()` in this case is still a `File::MP3` object, even though the method is found in the `File` class.

We can override a parent's method in a child class. When we do so, we can still call the parent class's method with the SUPER pseudo-class.

```
sub save {
    my $self = shift;

    say 'Prepare to rock';
    $self->SUPER::save();
}
```

The SUPER modifier can *only* be used for method calls. You can't use it for regular subroutine calls or class methods:

```
SUPER::save($thing);     # FAIL: looks for save() sub in package SUPER

SUPER->save($thing);     # FAIL: looks for save() method in class
                         #       SUPER

$thing->SUPER::save();   # Okay: looks for save() method in parent
                         #       classes
```

How SUPER is Resolved

The SUPER pseudo-class is resolved from the package where the call is made. It is *not* resolved based on the object's class. This is important, because it lets methods at different levels within a deep inheritance hierarchy each correctly call their respective parent methods.

```
package A;

sub new {
    return bless {}, shift;
}

sub speak {
    my $self = shift;

    $self->SUPER::speak();

    say 'A';
}

package B;

use parent 'A';

sub speak {
    my $self = shift;

    $self->SUPER::speak();

    say 'B';
}

package C;

use parent 'B';

sub speak {
    my $self = shift;

    $self->SUPER::speak();
```

```
    say 'C';
}

my $c = C->new();
$c->speak();
```

In this example, we will get the following output:

```
A
B
C
```

This demonstrates how SUPER is resolved. Even though the object is blessed into the C class, the `speak()` method in the B class can still call `SUPER::speak()` and expect it to correctly look in the parent class of B (i.e the class the method call is in), not in the parent class of C (i.e. the class the object belongs to).

There are rare cases where this package-based resolution can be a problem. If you copy a subroutine from one package to another, SUPER resolution will be done based on the original package.

Multiple Inheritance

Multiple inheritance often indicates a design problem, but Perl always gives you enough rope to hang yourself with if you ask for it.

To declare multiple parents, you simply need to pass multiple class names to `use parent`:

```
package MultiChild;

use parent 'Parent1', 'Parent2';
```

Method Resolution Order

Method resolution order only matters in the case of multiple inheritance. In the case of single inheritance, Perl simply looks up the inheritance chain to find a method:

```
Grandparent
  |
Parent
  |
Child
```

If we call a method on a `Child` object and that method is not defined in the `Child` class, Perl will look for that method in the `Parent` class and then, if necessary, in the `Grandparent` class.

If Perl cannot find the method in any of these classes, it will die with an error message.

When a class has multiple parents, the method lookup order becomes more complicated.

By default, Perl does a depth-first left-to-right search for a method. That means it starts with the first parent in the @ISA array, and then searches all of its parents, grandparents, etc. If it fails to find the method, it then goes to the next parent in the original class's @ISA array and searches from there.

```
           SharedGreatGrandParent
           /                    \
PaternalGrandparent       MaternalGrandparent
           \                    /
            Father         Mother
                  \       /
```

```
                    Child
```

So given the diagram above, Perl will search `Child`, `Father`, `PaternalGrandparent`, `SharedGreatGrandParent`, `Mother`, and finally `MaternalGrandparent`. This may be a problem because now we're looking in `SharedGreatGrandParent` *before* we've checked all its derived classes (i.e. before we tried `Mother` and `MaternalGrandparent`).

It is possible to ask for a different method resolution order with the *mro* pragma.

```
package Child;

use mro 'c3';
use parent 'Father', 'Mother';
```

This pragma lets you switch to the "C3" resolution order. In simple terms, "C3" order ensures that shared parent classes are never searched before child classes, so Perl will now search: `Child`, `Father`, `PaternalGrandparent`, `Mother` `MaternalGrandparent`, and finally `SharedGreatGrandParent`. Note however that this is not "breadth-first" searching: All the `Father` ancestors (except the common ancestor) are searched before any of the `Mother` ancestors are considered.

The C3 order also lets you call methods in sibling classes with the `next` pseudo-class. See the *mro* documentation for more details on this feature.

Method Resolution Caching

When Perl searches for a method, it caches the lookup so that future calls to the method do not need to search for it again. Changing a class's parent class or adding subroutines to a class will invalidate the cache for that class.

The *mro* pragma provides some functions for manipulating the method cache directly.

Writing Constructors

As we mentioned earlier, Perl provides no special constructor syntax. This means that a class must implement its own constructor. A constructor is simply a class method that returns a reference to a new object.

The constructor can also accept additional parameters that define the object. Let's write a real constructor for the `File` class we used earlier:

```
package File;

sub new {
    my $class = shift;
    my ( $path, $data ) = @_;

    my $self = bless {
        path => $path,
        data => $data,
    }, $class;

    return $self;
}
```

As you can see, we've stored the path and file data in the object itself. Remember, under the hood, this object is still just a hash. Later, we'll write accessors to manipulate this data.

For our File::MP3 class, we can check to make sure that the path we're given ends with ".mp3":

```
package File::MP3;

sub new {
    my $class = shift;
    my ( $path, $data ) = @_;

    die "You cannot create a File::MP3 without an mp3 extension\n"
        unless $path =~ /\.mp3\z/;

    return $class->SUPER::new(@_);
}
```

This constructor lets its parent class do the actual object construction.

Attributes

An attribute is a piece of data belonging to a particular object. Unlike most object-oriented languages, Perl provides no special syntax or support for declaring and manipulating attributes.

Attributes are often stored in the object itself. For example, if the object is an anonymous hash, we can store the attribute values in the hash using the attribute name as the key.

While it's possible to refer directly to these hash keys outside of the class, it's considered a best practice to wrap all access to the attribute with accessor methods.

This has several advantages. Accessors make it easier to change the implementation of an object later while still preserving the original API.

An accessor lets you add additional code around attribute access. For example, you could apply a default to an attribute that wasn't set in the constructor, or you could validate that a new value for the attribute is acceptable.

Finally, using accessors makes inheritance much simpler. Subclasses can use the accessors rather than having to know how a parent class is implemented internally.

Writing Accessors

As with constructors, Perl provides no special accessor declaration syntax, so classes must provide explicitly written accessor methods. There are two common types of accessors, read-only and read-write.

A simple read-only accessor simply gets the value of a single attribute:

```
sub path {
    my $self = shift;

    return $self->{path};
}
```

A read-write accessor will allow the caller to set the value as well as get it:

```
sub path {
    my $self = shift;

    if (@_) {
        $self->{path} = shift;
    }
```

```
    return $self->{path};
}
```

An Aside About Smarter and Safer Code

Our constructor and accessors are not very smart. They don't check that a `$path` is defined, nor do they check that a `$path` is a valid filesystem path.

Doing these checks by hand can quickly become tedious. Writing a bunch of accessors by hand is also incredibly tedious. There are a lot of modules on CPAN that can help you write safer and more concise code, including the modules we recommend in *perlootut*.

Method Call Variations

Perl supports several other ways to call methods besides the `$object->method()` usage we've seen so far.

Method Names as Strings

Perl lets you use a scalar variable containing a string as a method name:

```
my $file = File->new( $path, $data );

my $method = 'save';
$file->$method();
```

This works exactly like calling `$file->save()`. This can be very useful for writing dynamic code. For example, it allows you to pass a method name to be called as a parameter to another method.

Class Names as Strings

Perl also lets you use a scalar containing a string as a class name:

```
my $class = 'File';

my $file = $class->new( $path, $data );
```

Again, this allows for very dynamic code.

Subroutine References as Methods

You can also use a subroutine reference as a method:

```
my $sub = sub {
    my $self = shift;

    $self->save();
};

$file->$sub();
```

This is exactly equivalent to writing `$sub->($file)`. You may see this idiom in the wild combined with a call to `can`:

```
if ( my $meth = $object->can('foo') ) {
    $object->$meth();
}
```

Deferencing Method Call

Perl also lets you use a dereferenced scalar reference in a method call. That's a mouthful, so let's look at some code:

```
$file->${ \'save' };
$file->${ returns_scalar_ref() };
$file->${ \( returns_scalar() ) };
$file->${ returns_sub_ref() };
```

This works if the dereference produces a string *or* a subroutine reference.

Method Calls on Filehandles

Under the hood, Perl filehandles are instances of the `IO::Handle` or `IO::File` class. Once you have an open filehandle, you can call methods on it. Additionally, you can call methods on the `STDIN`, `STDOUT`, and `STDERR` filehandles.

```
open my $fh, '>', 'path/to/file';
$fh->autoflush();
$fh->print('content');

STDOUT->autoflush();
```

Invoking Class Methods

Because Perl allows you to use barewords for package names and subroutine names, it sometimes interprets a bareword's meaning incorrectly. For example, the construct `Class->new()` can be interpreted as either `'Class'->new()` or `Class()->new()`. In English, that second interpretation reads as "call a subroutine named Class(), then call new() as a method on the return value of Class()". If there is a subroutine named `Class()` in the current namespace, Perl will always interpret `Class->new()` as the second alternative: a call to `new()` on the object returned by a call to `Class()`

You can force Perl to use the first interpretation (i.e. as a method call on the class named "Class") in two ways. First, you can append a `::` to the class name:

```
Class::->new()
```

Perl will always interpret this as a method call.

Alternatively, you can quote the class name:

```
'Class'->new()
```

Of course, if the class name is in a scalar Perl will do the right thing as well:

```
my $class = 'Class';
$class->new();
```

Indirect Object Syntax

Outside of the file handle case, use of this syntax is discouraged, as it can confuse the Perl interpreter. See below for more details.

Perl suports another method invocation syntax called "indirect object" notation. This syntax is called "indirect" because the method comes before the object it is being invoked on.

This syntax can be used with any class or object method:

```
my $file = new File $path, $data;
save $file;
```

We recommend that you avoid this syntax, for several reasons.

First, it can be confusing to read. In the above example, it's not clear if `save` is a method provided by the `File` class or simply a subroutine that expects a file object as its first argument.

When used with class methods, the problem is even worse. Because Perl allows subroutine names to be written as barewords, Perl has to guess whether the bareword after the method is a class name or subroutine name. In other words, Perl can resolve the syntax as either `File->new( $path, $data )` **or** `new( File( $path, $data ) )`.

To parse this code, Perl uses a heuristic based on what package names it has seen, what subroutines exist in the current package, what barewords it has previously seen, and other input. Needless to say, heuristics can produce very surprising results!

Older documentation (and some CPAN modules) encouraged this syntax, particularly for constructors, so you may still find it in the wild. However, we encourage you to avoid using it in new code.

You can force Perl to interpret the bareword as a class name by appending "::" to it, like we saw earlier:

```
my $file = new File:: $path, $data;
```

bless, blessed, and ref

As we saw earlier, an object is simply a data structure that has been blessed into a class via the `bless` function. The `bless` function can take either one or two arguments:

```
my $object = bless {}, $class;
my $object = bless {};
```

In the first form, the anonymous hash is being blessed into the class in `$class`. In the second form, the anonymous hash is blessed into the current package.

The second form is strongly discouraged, because it breaks the ability of a subclass to reuse the parent's constructor, but you may still run across it in existing code.

If you want to know whether a particular scalar refers to an object, you can use the `blessed` function exported by *Scalar::Util*, which is shipped with the Perl core.

```
use Scalar::Util 'blessed';

if ( defined blessed($thing) ) { ... }
```

If `$thing` refers to an object, then this function returns the name of the package the object has been blessed into. If `$thing` doesn't contain a reference to a blessed object, the `blessed` function returns `undef`.

Note that `blessed($thing)` will also return false if `$thing` has been blessed into a class named "0". This is a possible, but quite pathological. Don't create a class named "0" unless you know what you're doing.

Similarly, Perl's built-in `ref` function treats a reference to a blessed object specially. If you call `ref($thing)` and `$thing` holds a reference to an object, it will return the name of the class that the object has been blessed into.

If you simply want to check that a variable contains an object reference, we recommend that you use `defined blessed($object)`, since `ref` returns true values for all references, not just objects.

The UNIVERSAL Class

All classes automatically inherit from the *UNIVERSAL* class, which is built-in to the Perl core. This class provides a number of methods, all of which can be called on either a class or an object. You can also choose to override some of these methods in your class. If you do so, we recommend that you follow the built-in semantics described below.

isa($class)

The `isa` method returns *true* if the object is a member of the class in `$class`, or a member of a subclass of `$class`.

If you override this method, it should never throw an exception.

DOES($role)

The `DOES` method returns *true* if its object claims to perform the role `$role`. By default, this is equivalent to `isa`. This method is provided for use by object system extensions that implement roles, like `Moose` and `Role::Tiny`.

You can also override `DOES` directly in your own classes. If you override this method, it should never throw an exception.

can($method)

The `can` method checks to see if the class or object it was called on has a method named `$method`. This checks for the method in the class and all of its parents. If the method exists, then a reference to the subroutine is returned. If it does not then `undef` is returned.

If your class responds to method calls via `AUTOLOAD`, you may want to overload `can` to return a subroutine reference for methods which your `AUTOLOAD` method handles.

If you override this method, it should never throw an exception.

VERSION($need)

The `VERSION` method 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 $VERSION variable in the package) is greater than or equal to `$need`; it will die if this is not the case. This method is called automatically by the `VERSION` form of `use`.

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

We recommend that you use this method to access another package's version, rather than looking directly at `$Package::VERSION`. The package you are looking at could have overridden the `VERSION` method.

We also recommend using this method to check whether a module has a sufficient version. The internal implementation uses the *version* module to make sure that different types of version numbers are compared correctly.

AUTOLOAD

If you call a method that doesn't exist in a class, Perl will throw an error. However, if that class or any of its parent classes defines an `AUTOLOAD` method, that `AUTOLOAD` method is called instead.

`AUTOLOAD` is called as a regular method, and the caller will not know the difference. Whatever value your `AUTOLOAD` method returns is returned to the caller.

The fully qualified method name that was called is available in the `$AUTOLOAD` package global for your class. Since this is a global, if you want to refer to do it without a package name prefix under

`strict 'vars'`, you need to declare it.

```
  # XXX - this is a terrible way to implement accessors, but it makes
  # for a simple example.
  our $AUTOLOAD;
  sub AUTOLOAD {
      my $self = shift;

      # Remove qualifier from original method name...
      my $called =  $AUTOLOAD =~ s/.*::///r;

      # Is there an attribute of that name?
      die "No such attribute: $called"
          unless exists $self->{$called};

      # If so, return it...
      return $self->{$called};
  }

  sub DESTROY { } # see below
```

Without the `our $AUTOLOAD` declaration, this code will not compile under the *strict* pragma.

As the comment says, this is not a good way to implement accessors. It's slow and too clever by far. However, you may see this as a way to provide accessors in older Perl code. See *perlootut* for recommendations on OO coding in Perl.

If your class does have an AUTOLOAD method, we strongly recommend that you override `can` in your class as well. Your overridden `can` method should return a subroutine reference for any method that your AUTOLOAD responds to.

Destructors

When the last reference to an object goes away, the object is destroyed. If you only have one reference to an object stored in a lexical scalar, the object is destroyed when that scalar goes out of scope. If you store the object in a package global, that object may not go out of scope until the program exits.

If you want to do something when the object is destroyed, you can define a DESTROY method in your class. This method will always be called by Perl at the appropriate time, unless the method is empty.

This is called just like any other method, with the object as the first argument. It does not receive any additional arguments. However, the `$_[0]` variable will be read-only in the destructor, so you cannot assign a value to it.

If your DESTROY method throws an error, this error will be ignored. It will not be sent to STDERR and it will not cause the program to die. However, if your destructor is running inside an `eval {}` block, then the error will change the value of `$@`.

Because DESTROY methods can be called at any time, you should localize any global variables you might update in your DESTROY. In particular, if you use `eval {}` you should localize `$@`, and if you use `system` or backticks, you should localize `$?`.

If you define an AUTOLOAD in your class, then Perl will call your AUTOLOAD to handle the DESTROY method. You can prevent this by defining an empty DESTROY, like we did in the autoloading example. You can also check the value of $AUTOLOAD and return without doing anything when called to handle DESTROY.

Global Destruction

The order in which objects are destroyed during the global destruction before the program exits is unpredictable. This means that any objects contained by your object may already have been destroyed. You should check that a contained object is defined before calling a method on it:

```
sub DESTROY {
    my $self = shift;

    $self->{handle}->close() if $self->{handle};
}
```

You can use the `${^GLOBAL_PHASE}` variable to detect if you are currently in the global destruction phase:

```
sub DESTROY {
    my $self = shift;

    return if ${^GLOBAL_PHASE} eq 'DESTRUCT';

    $self->{handle}->close();
}
```

Note that this variable was added in Perl 5.14.0. If you want to detect the global destruction phase on older versions of Perl, you can use the `Devel::GlobalDestruction` module on CPAN.

If your `DESTROY` method issues a warning during global destruction, the Perl interpreter will append the string " during global destruction" the warning.

During global destruction, Perl will always garbage collect objects before unblessed references. See *"PERL_DESTRUCT_LEVEL" in perlhacktips* for more information about global destruction.

Non-Hash Objects

All the examples so far have shown objects based on a blessed hash. However, it's possible to bless any type of data structure or referent, including scalars, globs, and subroutines. You may see this sort of thing when looking at code in the wild.

Here's an example of a module as a blessed scalar:

```
package Time;

use strict;
use warnings;

sub new {
    my $class = shift;

    my $time = time;
    return bless \$time, $class;
}

sub epoch {
    my $self = shift;
    return ${ $self };
}
```

```
my $time = Time->new();
print $time->epoch();
```

Inside-Out objects

In the past, the Perl community experimented with a technique called "inside-out objects". An inside-out object stores its data outside of the object's reference, indexed on a unique property of the object, such as its memory address, rather than in the object itself. This has the advantage of enforcing the encapsulation of object attributes, since their data is not stored in the object itself.

This technique was popular for a while (and was recommended in Damian Conway's *Perl Best Practices*), but never achieved universal adoption. The *Object::InsideOut* module on CPAN provides a comprehensive implementation of this technique, and you may see it or other inside-out modules in the wild.

Here is a simple example of the technique, using the *Hash::Util::FieldHash* core module. This module was added to the core to support inside-out object implementations.

```
package Time;

use strict;
use warnings;

use Hash::Util::FieldHash 'fieldhash';

fieldhash my %time_for;

sub new {
    my $class = shift;

    my $self = bless \( my $object ), $class;

    $time_for{$self} = time;

    return $self;
}

sub epoch {
    my $self = shift;

    return $time_for{$self};
}

my $time = Time->new;
print $time->epoch;
```

Pseudo-hashes

The pseudo-hash feature was an experimental feature introduced in earlier versions of Perl and removed in 5.10.0. A pseudo-hash is an array reference which can be accessed using named keys like a hash. You may run in to some code in the wild which uses it. See the *fields* pragma for more information.

SEE ALSO

A kinder, gentler tutorial on object-oriented programming in Perl can be found in *perlootut*. You should also check out *perlmodlib* for some style guides on constructing both modules and classes.