NAME

perlootut − Object−Oriented Programming in Perl Tutorial

DATE

This document was created in February, 2011, and the last major revision was in February, 2013.

If you are reading this in the future then it's possible that the state of the art has changed. We recommend you start by reading the perlootut document in the latest stable release of Perl, rather than this version.

DESCRIPTION

This document provides an introduction to object-oriented programming in Perl. It begins with a brief overview of the concepts behind object oriented design. Then it introduces several different OO systems from CPAN <http://search.cpan.org> which build on top of what Perl provides.

By default, Perl's built-in OO system is very minimal, leaving you to do most of the work. This minimalism made a lot of sense in 1994, but in the years since Perl 5.0 we've seen a number of common patterns emerge in Perl OO. Fortunately, Perl's flexibility has allowed a rich ecosystem of Perl OO systems to flourish.

If you want to know how Perl OO works under the hood, the perlobj document explains the nitty gritty details.

This document assumes that you already understand the basics of Perl syntax, variable types, operators, and subroutine calls. If you don't understand these concepts yet, please read perlintro first. You should also read the perlsyn, perlop, and perlsub documents.

OBJECT-ORIENTED FUNDAMENTALS

Most object systems share a number of common concepts. You've probably heard terms like ''class'', ''object, ''method'', and ''attribute" before. Understanding the concepts will make it much easier to read and write object-oriented code. If you're already familiar with these terms, you should still skim this section, since it explains each concept in terms of Perl's OO implementation.

Perl's OO system is class-based. Class-based OO is fairly common. It's used by Java, C++, C#, Python, Ruby, and many other languages. There are other object orientation paradigms as well. JavaScript is the most popular language to use another paradigm. JavaScript's OO system is prototype-based.

Object

An **object** is a data structure that bundles together data and subroutines which operate on that data. An object's data is called **attributes**, and its subroutines are called **methods**. An object can be thought of as a noun (a person, a web service, a computer).

An object represents a single discrete thing. For example, an object might represent a file. The attributes for a file object might include its path, content, and last modification time. If we created an object to represent */etc/hostname* on a machine named ''foo.example.com'', that object's path would be ''/etc/hostname'', its content would be ''foo\n'', and it's last modification time would be 1304974868 seconds since the beginning of the epoch.

The methods associated with a file might include `rename()` and `write()`.

In Perl most objects are hashes, but the OO systems we recommend keep you from having to worry about this. In practice, it's best to consider an object's internal data structure opaque.

Class

A **class** defines the behavior of a category of objects. A class is a name for a category (like ''File''), and a class also defines the behavior of objects in that category.

All objects belong to a specific class. For example, our */etc/hostname* object belongs to the `File` class. When we want to create a specific object, we start with its class, and **construct** or **instantiate** an object. A specific object is often referred to as an **instance** of a class.

In Perl, any package can be a class. The difference between a package which is a class and one which isn't is based on how the package is used. Here's our ''class declaration'' for the `File` class:

```
package File;
```

In Perl, there is no special keyword for constructing an object. However, most OO modules on CPAN use a method named `new()` to construct a new object:

```
my $hostname = File->new(
    path          => '/etc/hostname',
    content       => "foo\n",
    last_mod_time => 1304974868,
);
```

(Don't worry about that `->` operator, it will be explained later.)

Blessing

As we said earlier, most Perl objects are hashes, but an object can be an instance of any Perl data type (scalar, array, etc.). Turning a plain data structure into an object is done by **blessing** that data structure using Perl's `bless` function.

While we strongly suggest you don't build your objects from scratch, you should know the term **bless**. A **blessed** data structure (aka "a referent") is an object. We sometimes say that an object has been "blessed into a class".

Once a referent has been blessed, the `blessed` function from the Scalar::Util core module can tell us its class name. This subroutine returns an object's class when passed an object, and false otherwise.

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

print blessed($hash);       # undef
print blessed($hostname);   # File
```

Constructor

A **constructor** creates a new object. In Perl, a class's constructor is just another method, unlike some other languages, which provide syntax for constructors. Most Perl classes use `new` as the name for their constructor:

```
my $file = File->new(...);
```

Methods

You already learned that a **method** is a subroutine that operates on an object. You can think of a method as the things that an object can *do*. If an object is a noun, then methods are its verbs (save, print, open).

In Perl, methods are simply subroutines that live in a class's package. Methods are always written to receive the object as their first argument:

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

    print "This file is at ", $self->path, "\n";
}

$file->print_info;
# The file is at /etc/hostname
```

What makes a method special is *how it's called*. The arrow operator (`->`) tells Perl that we are calling a method.

When we make a method call, Perl arranges for the method's **invocant** to be passed as the first argument. **Invocant** is a fancy name for the thing on the left side of the arrow. The invocant can either be a class name or an object. We can also pass additional arguments to the method:

```
sub print_info {
    my $self   = shift;
    my $prefix = shift // "This file is at ";

    print $prefix, ", ", $self->path, "\n";
}

$file->print_info("The file is located at ");
# The file is located at /etc/hostname
```

Attributes

Each class can define its **attributes**. When we instantiate an object, we assign values to those attributes. For example, every `File` object has a path. Attributes are sometimes called **properties**.

Perl has no special syntax for attributes. Under the hood, attributes are often stored as keys in the object's underlying hash, but don't worry about this.

We recommend that you only access attributes via **accessor** methods. These are methods that can get or set the value of each attribute. We saw this earlier in the `print_info()` example, which calls `$self->path`.

You might also see the terms **getter** and **setter**. These are two types of accessors. A getter gets the attribute's value, while a setter sets it. Another term for a setter is **mutator**

Attributes are typically defined as read-only or read-write. Read-only attributes can only be set when the object is first created, while read-write attributes can be altered at any time.

The value of an attribute may itself be another object. For example, instead of returning its last mod time as a number, the `File` class could return a DateTime object representing that value.

It's possible to have a class that does not expose any publicly settable attributes. Not every class has attributes and methods.

Polymorphism

Polymorphism is a fancy way of saying that objects from two different classes share an API. For example, we could have `File` and `WebPage` classes which both have a `print_content()` method. This method might produce different output for each class, but they share a common interface.

While the two classes may differ in many ways, when it comes to the `print_content()` method, they are the same. This means that we can try to call the `print_content()` method on an object of either class, and **we don't have to know what class the object belongs to!**

Polymorphism is one of the key concepts of object-oriented design.

Inheritance

Inheritance lets you create a specialized version of an existing class. Inheritance lets the new class reuse the methods and attributes of another class.

For example, we could create an `File::MP3` class which **inherits** from `File`. An `File::MP3` **is-a** *more specific* type of `File`. All mp3 files are files, but not all files are mp3 files.

We often refer to inheritance relationships as **parent-child** or `superclass/subclass` relationships. Sometimes we say that the child has an **is-a** relationship with its parent class.

`File` is a **superclass** of `File::MP3` and `File::MP3` is a **subclass** of `File`.

```
package File::MP3;

use parent 'File';
```

The parent module is one of several ways that Perl lets you define inheritance relationships.

Perl allows multiple inheritance, which means that a class can inherit from multiple parents. While this is possible, we strongly recommend against it. Generally, you can use **roles** to do everything you can do with multiple inheritance, but in a cleaner way.

Note that there's nothing wrong with defining multiple subclasses of a given class. This is both common and safe. For example, we might define `File::MP3::FixedBitrate` and `File::MP3::VariableBitrate` classes to distinguish between different types of mp3 file.

Overriding methods and method resolution

Inheritance allows two classes to share code. By default, every method in the parent class is also available in the child. The child can explicitly **override** a parent's method to provide its own implementation. For example, if we have an `File::MP3` object, it has the `print_info()` method from `File`:

```
my $cage = File::MP3->new(
    path          => 'mp3s/My-Body-Is-a-Cage.mp3',
    content       => $mp3_data,
    last_mod_time => 1304974868,
    title         => 'My Body Is a Cage',
);

$cage->print_info;
# The file is at mp3s/My-Body-Is-a-Cage.mp3
```

If we wanted to include the mp3's title in the greeting, we could override the method:

```
package File::MP3;

use parent 'File';

sub print_info {
    my $self = shift;

    print "This file is at ", $self->path, "\n";
    print "Its title is ", $self->title, "\n";
}

$cage->print_info;
# The file is at mp3s/My-Body-Is-a-Cage.mp3
# Its title is My Body Is a Cage
```

The process of determining what method should be used is called **method resolution**. What Perl does is look at the object's class first (`File::MP3` in this case). If that class defines the method, then that class's version of the method is called. If not, Perl looks at each parent class in turn. For `File::MP3` its only parent is `File`. If `File::MP3` does not define the method, but `File` does, then Perl calls the method in `File`.

If `File` inherited from `DataSource`, which inherited from `Thing`, then Perl would keep looking "up the chain" if necessary.

It is possible to explicitly call a parent method from a child:

```
package File::MP3;

use parent 'File';

sub print_info {
    my $self = shift;

    $self->SUPER::print_info();
    print "Its title is ", $self->title, "\n";
}
```

The `SUPER::` bit tells Perl to look for the `print_info()` in the `File::MP3` class's inheritance chain.

When it finds the parent class that implements this method, the method is called.

We mentioned multiple inheritance earlier. The main problem with multiple inheritance is that it greatly complicates method resolution. See perlobj(1) for more details.

Encapsulation

Encapsulation is the idea that an object is opaque. When another developer uses your class, they don't need to know *how* it is implemented, they just need to know *what* it does.

Encapsulation is important for several reasons. First, it allows you to separate the public API from the private implementation. This means you can change that implementation without breaking the API.

Second, when classes are well encapsulated, they become easier to subclass. Ideally, a subclass uses the same APIs to access object data that its parent class uses. In reality, subclassing sometimes involves violating encapsulation, but a good API can minimize the need to do this.

We mentioned earlier that most Perl objects are implemented as hashes under the hood. The principle of encapsulation tells us that we should not rely on this. Instead, we should use accessor methods to access the data in that hash. The object systems that we recommend below all automate the generation of accessor methods. If you use one of them, you should never have to access the object as a hash directly.

Composition

In object-oriented code, we often find that one object references another object. This is called **composition**, or a **has-a** relationship.

Earlier, we mentioned that the `File` class's `last_mod_time` accessor could return a DateTime object. This is a perfect example of composition. We could go even further, and make the `path` and `content` accessors return objects as well. The `File` class would then be **composed** of several other objects.

Roles

Roles are something that a class *does*, rather than something that it *is*. Roles are relatively new to Perl, but have become rather popular. Roles are **applied** to classes. Sometimes we say that classes **consume** roles.

Roles are an alternative to inheritance for providing polymorphism. Let's assume we have two classes, `Radio` and `Computer`. Both of these things have on/off switches. We want to model that in our class definitions.

We could have both classes inherit from a common parent, like `Machine`, but not all machines have on/off switches. We could create a parent class called `HasOnOffSwitch`, but that is very artificial. Radios and computers are not specializations of this parent. This parent is really a rather ridiculous creation.

This is where roles come in. It makes a lot of sense to create a `HasOnOffSwitch` role and apply it to both classes. This role would define a known API like providing `turn_on()` and `turn_off()` methods.

Perl does not have any built-in way to express roles. In the past, people just bit the bullet and used multiple inheritance. Nowadays, there are several good choices on CPAN for using roles.

When to Use OO

Object Orientation is not the best solution to every problem. In *Perl Best Practices* (copyright 2004, Published by O'Reilly Media, Inc.), Damian Conway provides a list of criteria to use when deciding if OO is the right fit for your problem:

- The system being designed is large, or is likely to become large.
- The data can be aggregated into obvious structures, especially if there's a large amount of data in each aggregate.
- The various types of data aggregate form a natural hierarchy that facilitates the use of inheritance and polymorphism.
- You have a piece of data on which many different operations are applied.
- You need to perform the same general operations on related types of data, but with slight variations depending on the specific type of data the operations are applied to.

- It's likely you'll have to add new data types later.
- The typical interactions between pieces of data are best represented by operators.
- The implementation of individual components of the system is likely to change over time.
- The system design is already object-oriented.
- Large numbers of other programmers will be using your code modules.

PERL OO SYSTEMS

As we mentioned before, Perl's built-in OO system is very minimal, but also quite flexible. Over the years, many people have developed systems which build on top of Perl's built-in system to provide more features and convenience.

We strongly recommend that you use one of these systems. Even the most minimal of them eliminates a lot of repetitive boilerplate. There's really no good reason to write your classes from scratch in Perl.

If you are interested in the guts underlying these systems, check out perlobj.

Moose

Moose bills itself as a ''postmodern object system for Perl 5''. Don't be scared, the ''postmodern'' label is a callback to Larry's description of Perl as ''the first postmodern computer language''.

`Moose` provides a complete, modern OO system. Its biggest influence is the Common Lisp Object System, but it also borrows ideas from Smalltalk and several other languages. `Moose` was created by Stevan Little, and draws heavily from his work on the Perl 6 OO design.

Here is our `File` class using `Moose`:

```
package File;
use Moose;

has path          => ( is => 'ro' );
has content       => ( is => 'ro' );
has last_mod_time => ( is => 'ro' );

sub print_info {
    my $self = shift;

    print "This file is at ", $self->path, "\n";
}
```

`Moose` provides a number of features:

- Declarative sugar

  `Moose` provides a layer of declarative ''sugar'' for defining classes. That sugar is just a set of exported functions that make declaring how your class works simpler and more palatable. This lets you describe *what* your class is, rather than having to tell Perl *how* to implement your class.

  The `has()` subroutine declares an attribute, and `Moose` automatically creates accessors for these attributes. It also takes care of creating a `new()` method for you. This constructor knows about the attributes you declared, so you can set them when creating a new `File`.

- Roles built-in

  `Moose` lets you define roles the same way you define classes:

  ```
  package HasOnOffSwitch;
  use Moose::Role;

  has is_on => (
      is  => 'rw',
      isa => 'Bool',
  ```

```
);

sub turn_on {
    my $self = shift;
    $self->is_on(1);
}

sub turn_off {
    my $self = shift;
    $self->is_on(0);
}
```

- A miniature type system

  In the example above, you can see that we passed `isa => 'Bool'` to `has()` when creating our `is_on` attribute. This tells `Moose` that this attribute must be a boolean value. If we try to set it to an invalid value, our code will throw an error.

- Full introspection and manipulation

  Perl's built-in introspection features are fairly minimal. `Moose` builds on top of them and creates a full introspection layer for your classes. This lets you ask questions like "what methods does the File class implement?" It also lets you modify your classes programmatically.

- Self-hosted and extensible

  `Moose` describes itself using its own introspection API. Besides being a cool trick, this means that you can extend `Moose` using `Moose` itself.

- Rich ecosystem

  There is a rich ecosystem of `Moose` extensions on CPAN under the MooseX <http://search.cpan.org/search?query=MooseX&mode=dist> namespace. In addition, many modules on CPAN already use `Moose`, providing you with lots of examples to learn from.

- Many more features

  `Moose` is a very powerful tool, and we can't cover all of its features here. We encourage you to learn more by reading the `Moose` documentation, starting with Moose::Manual <http://search.cpan.org/perldoc?Moose::Manual>.

Of course, `Moose` isn't perfect.

`Moose` can make your code slower to load. `Moose` itself is not small, and it does a *lot* of code generation when you define your class. This code generation means that your runtime code is as fast as it can be, but you pay for this when your modules are first loaded.

This load time hit can be a problem when startup speed is important, such as with a command-line script or a "plain vanilla" CGI script that must be loaded each time it is executed.

Before you panic, know that many people do use `Moose` for command-line tools and other startup-sensitive code. We encourage you to try `Moose` out first before worrying about startup speed.

`Moose` also has several dependencies on other modules. Most of these are small stand-alone modules, a number of which have been spun off from `Moose`. `Moose` itself, and some of its dependencies, require a compiler. If you need to install your software on a system without a compiler, or if having *any* dependencies is a problem, then `Moose` may not be right for you.

Moo

If you try `Moose` and find that one of these issues is preventing you from using `Moose`, we encourage you to consider Moo next. `Moo` implements a subset of `Moose`'s functionality in a simpler package. For most features that it does implement, the end-user API is *identical* to `Moose`, meaning you can switch from `Moo` to `Moose` quite easily.

`Moo` does not implement most of `Moose`'s introspection API, so it's often faster when loading your modules. Additionally, none of its dependencies require XS, so it can be installed on machines without a compiler.

One of `Moo`'s most compelling features is its interoperability with `Moose`. When someone tries to use `Moose`'s introspection API on a `Moo` class or role, it is transparently inflated into a `Moose` class or role. This makes it easier to incorporate `Moo`–using code into a `Moose` code base and vice versa.

For example, a `Moose` class can subclass a `Moo` class using `extends` or consume a `Moo` role using `with`.

The `Moose` authors hope that one day `Moo` can be made obsolete by improving `Moose` enough, but for now it provides a worthwhile alternative to `Moose`.

Class::Accessor

Class::Accessor is the polar opposite of `Moose`. It provides very few features, nor is it self-hosting.

It is, however, very simple, pure Perl, and it has no non-core dependencies. It also provides a ''Moose-like'' API on demand for the features it supports.

Even though it doesn't do much, it is still preferable to writing your own classes from scratch.

Here's our `File` class with `Class::Accessor`

```
package File;
use Class::Accessor 'antlers';

has path          => ( is => 'ro' );
has content       => ( is => 'ro' );
has last_mod_time => ( is => 'ro' );

sub print_info {
    my $self = shift;

    print "This file is at ", $self->path, "\n";
}
```

The `antlers` import flag tells `Class::Accessor` that you want to define your attributes using `Moose`–like syntax. The only parameter that you can pass to `has` is `is`. We recommend that you use this Moose-like syntax if you choose `Class::Accessor` since it means you will have a smoother upgrade path if you later decide to move to `Moose`.

Like `Moose`, `Class::Accessor` generates accessor methods and a constructor for your class.

Class::Tiny

Finally, we have Class::Tiny. This module truly lives up to its name. It has an incredibly minimal API and absolutely no dependencies on any recent Perl. Still, we think it's a lot easier to use than writing your own OO code from scratch.

Here's our `File` class once more:

```
package File;
use Class::Tiny qw( path content last_mod_time );

sub print_info {
    my $self = shift;

    print "This file is at ", $self->path, "\n";
}
```

That's it!

With `Class::Tiny` all accessors are read-write. It generates a constructor for you, as well as the accessors you define.

You can also use Class::Tiny::Antlers for `Moose`–like syntax.

Role::Tiny

As we mentioned before, roles provide an alternative to inheritance, but Perl does not have any built-in role support. If you choose to use Moose, it comes with a full-fledged role implementation. However, if you use one of our other recommended OO modules, you can still use roles with Role::Tiny

`Role::Tiny` provides some of the same features as Moose's role system, but in a much smaller package. Most notably, it doesn't support any sort of attribute declaration, so you have to do that by hand. Still, it's useful, and works well with `Class::Accessor` and `Class::Tiny`

OO System Summary

Here's a brief recap of the options we covered:

- Moose

  `Moose` is the maximal option. It has a lot of features, a big ecosystem, and a thriving user base. We also covered Moo briefly. `Moo` is `Moose` lite, and a reasonable alternative when Moose doesn't work for your application.

- Class::Accessor

  `Class::Accessor` does a lot less than `Moose`, and is a nice alternative if you find `Moose` overwhelming. It's been around a long time and is well battle-tested. It also has a minimal `Moose` compatibility mode which makes moving from `Class::Accessor` to `Moose` easy.

- Class::Tiny

  `Class::Tiny` is the absolute minimal option. It has no dependencies, and almost no syntax to learn. It's a good option for a super minimal environment and for throwing something together quickly without having to worry about details.

- Role::Tiny

  Use `Role::Tiny` with `Class::Accessor` or `Class::Tiny` if you find yourself considering multiple inheritance. If you go with `Moose`, it comes with its own role implementation.

Other OO Systems

There are literally dozens of other OO-related modules on CPAN besides those covered here, and you're likely to run across one or more of them if you work with other people's code.

In addition, plenty of code in the wild does all of its OO "by hand", using just the Perl built-in OO features. If you need to maintain such code, you should read perlobj(1) to understand exactly how Perl's built-in OO works.

CONCLUSION

As we said before, Perl's minimal OO system has led to a profusion of OO systems on CPAN. While you can still drop down to the bare metal and write your classes by hand, there's really no reason to do that with modern Perl.

For small systems, Class::Tiny and Class::Accessor both provide minimal object systems that take care of basic boilerplate for you.

For bigger projects, Moose provides a rich set of features that will let you focus on implementing your business logic. Moo provides a nice alternative to Moose when you want a lot of features but need faster compile time or to avoid XS.

We encourage you to play with and evaluate Moose, Moo, Class::Accessor, and Class::Tiny to see which OO system is right for you.