Extracted from:

Modern Perl, Fourth Edition

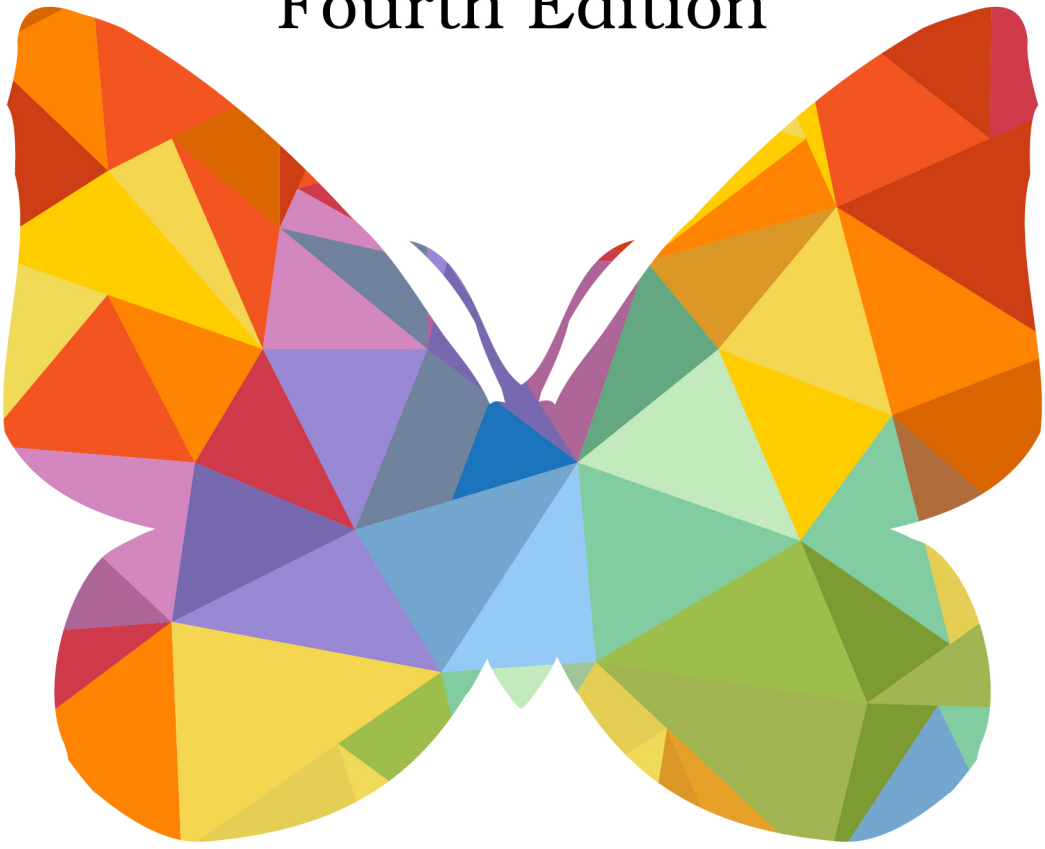
The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Modern Perl

Fourth Edition



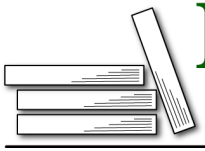
chromatic

The Classic Reference, Updated for Perl 5.22

Modern Perl, Fourth Edition

chromatic

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *https://pragprog.com*.

The team that produced this book includes:

Michael Swaine (editor)
Potomac Indexing, LLC (index)
Linda Recktenwald (copyedit)
Dave Thomas (layout)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-088-2
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—October 2015

Moose

Perl's default object system is minimal but flexible. Its syntax is a little clunky, and it exposes *how* an object system works. You can build great things on top of it, but it doesn't give you what many other languages do by default.

Moose is a complete object system for Perl. It's a complete distribution available from the CPAN—not a part of the core language but worth installing and using regardless. Moose offers both a simpler way to use an object system and advanced features of languages such as Smalltalk and Common Lisp.

Moose objects work with plain-vanilla Perl. Within your programs, you can mix and match objects written with Perl's default object system and Moose.

Moose Documentation

See `Moose::Manual` on the CPAN for comprehensive Moose documentation.

Classes

A Moose object is a concrete instance of a *class*, which is a template describing data and behavior specific to the object. A class generally belongs to a package (Packages on page ?), which provides its name:

```
package Cat {
    use Moose;
}
```

This Cat class *appears* to do nothing, but that's all Moose needs to make a class. You can create objects (or *instances*) of the Cat class with this syntax:

```
my $brad = Cat->new;
my $jack = Cat->new;
```

In the same way that this arrow operator dereferences a reference, it calls a method on Cat.

Methods

A *method* is a function associated with a class. In the same way that a function belongs to a namespace, a method belongs to a class.

When you call a method, you do so with an *invocant*. When you call new() on Cat, the name of the class, Cat, is new()'s invocant. Think of this as sending a message to a class: "do whatever new() does." In this case, calling the new() method—sending the new message—returns a new object of the Cat class.

When you call a method on an *object*, that object is the invocant:

```
my $choco = Cat->new;
$choco->sleep_on_keyboard;
```

A method's first argument is its invocant ($self, by convention). Suppose a Cat can meow():

```
package Cat {
    use Moose;

    sub meow {
        my $self = shift;
        say 'Meow!';
    }
}
```

Now any Cat instance can wake you for its early morning feeding:

```
# the cat always meows three times at 6 am
my $fuzzy_alarm = Cat->new;
$fuzzy_alarm->meow for 1 .. 3;
```

Every object can have its own distinct data. Methods that read or write the data of their invocants are *instance methods*; they depend on the presence of an appropriate invocant to work correctly. Methods (such as meow()) that don't access instance data are *class methods*. You may invoke class methods on classes and class and instance methods on instances, but you cannot invoke instance methods on classes.

Class methods are effectively namespaced global functions. Without access to instance data, they have few advantages over namespaced functions. Most OO code uses instance methods to read and write instance data.

Constructors, which *create* instances, are class methods. When you declare a Moose class, Moose provides a default constructor named new().

Attributes

Every Perl object is unique. Objects can contain private data associated with each unique object—often called *attributes*, *instance data*, or object *state*. You define an attribute by declaring it as part of the class:

```
package Cat {
    use Moose;

    has 'name', is => 'ro', isa => 'Str';
}
```

Moose exports the has() function for you to use to declare an attribute. In English, this code reads "Cat objects have a name attribute. It's read-only, and it's a string." The first argument, 'name', is the attribute's name. The is => 'ro' pair of arguments declares that this attribute is read only, so you cannot modify the attribute's value after you've set it. Finally, the isa => 'Str' pair declares that the value of this attribute can only be a String.

From this code Moose creates an *accessor* method named name() and allows you to pass a name parameter to Cat's constructor:

```
for my $name (qw( Tuxie Petunia Daisy )) {
    my $cat = Cat->new( name => $name );
    say "Created a cat for ", $cat->name;
}
```

Moose's uses parentheses to separate attribute names and characteristics:

```
has 'name' => ( is => 'ro', isa => 'Str' );
```

This is equivalent to the following:

```
has( 'name', 'is', 'ro', 'isa', 'Str' );
```

Moose's approach works nicely for complex declarations:

```
has 'name' => (
    is         => 'ro',
    isa        => 'Str',

    # advanced Moose options; perldoc Moose
    init_arg   => undef,
    lazy_build => 1,
);
```

But this book prefers a low-punctuation approach for simple declarations. Choose the style that offers you the most clarity.

When an attribute declaration has a type, Moose will attempt to validate all values assigned to that attribute. Sometimes this strictness is invaluable. While Moose will complain if you try to set name to a value that isn't a string, attributes don't *require* types. In that case, anything goes:

```
package Cat {
    use Moose;

    has 'name', is => 'ro', isa => 'Str';
    has 'age',  is => 'ro';
}

my $invalid = Cat->new( name => 'bizarre', age  => 'purple' );
```

If you mark an attribute as readable *and* writable (with is => rw), Moose will create a *mutator* method that can change that attribute's value:

```
package Cat {
    use Moose;

    has 'name', is => 'ro', isa => 'Str';
    has 'age',  is => 'ro', isa => 'Int';
    has 'diet', is => 'rw';
}

my $fat = Cat->new( name => 'Fatty',
                    age  => 8,
                    diet => 'Sea Treats' );

say $fat->name, ' eats ', $fat->diet;

$fat->diet( 'Low Sodium Kitty Lo Mein' );
say $fat->name, ' now eats ', $fat->diet;
```

An ro accessor used as a mutator will throw the exception Cannot assign a value to a read-only accessor at ....

Using ro or rw is a matter of design, convenience, and purity. Moose enforces no single philosophy here. Some people suggest making all instance data ro such that you must pass instance data into the constructor (Immutability on page ?). In the Cat example, age() might still be an accessor, but the constructor could take the *year* of the cat's birth and calculate the age itself based on the current year. This approach consolidates validation code and ensures that all objects have valid data after creation.

This illustrates a subtle but important principle of object orientation. An object contains related data and can perform behaviors with and on that data. A class describes that data and those behaviors. You can have multiple independent objects with separate instance data and treat all of those objects the same way; they will behave differently depending on their instance data.

Encapsulation

Moose allows you to declare *which* attributes class instances possess (a cat has a name) as well as the attributes of those attributes (you can name a cat once and thereafter its name cannot change). Moose itself decides how to *store* those attributes—you access them through accessors. This is *encapsulation*: hiding the internal details of an object from external users of that object.

Consider the aforementioned idea to change how Cats manage their ages by passing in the year of the cat's birth and calculating the age as needed:

```
package Cat {
    use Moose;

    has 'name',        is => 'ro', isa => 'Str';
    has 'diet',        is => 'rw';
    has 'birth_year',  is => 'ro', isa => 'Int';

    sub age {
        my $self = shift;
        my $year = (localtime)[5] + 1900;

        return $year - $self->birth_year;
    }
}
```

While the syntax for *creating* Cat objects has changed, the syntax for *using* Cat objects has not. Outside of Cat, age() behaves as it always has. *How* it works is a detail hidden inside the Cat class.

This change offers another advantage; a *default attribute value* will let users construct a new Cat object *without* providing a birth year:

```
package Cat {
    use Moose;

    has 'name', is => 'ro', isa => 'Str';
    has 'diet', is => 'rw', isa => 'Str';

    has 'birth_year',
        is      => 'ro',
        isa     => 'Int',
        default => sub { (localtime)[5] + 1900 };
}
```

The default keyword on an attribute uses a function reference (or a literal string or number) that returns the default value for that attribute when constructing a new object. If the code creating an object passes no constructor value for that attribute, the object gets the default value:

```
my $kitten = Cat->new( name => 'Hugo' );
```

And that kitten will have an age of 0 until next year.

Compatibility and APIs

Retain the old syntax for *creating* Cat objects by customizing the generated Cat constructor to allow passing an age parameter. Calculate birth_year from that. See perldoc Moose::Manual::Attributes.

Polymorphism

The real power of object orientation goes beyond classes and encapsulation. A well-designed OO program can manage many types of data. When well-designed classes encapsulate specific details of objects into the appropriate places, something curious happens: the code often becomes *less* specific.

Moving the details of what the program knows about individual Cats (the attributes) and what the program knows that Cats can do (the methods) into the Cat class means that code that deals with Cat instances can happily ignore *how* Cat does what it does.

Consider a function that displays details of an object:

```
sub show_vital_stats {
    my $object = shift;
    say 'My name is ', $object->name;
    say 'I am ',       $object->age;
    say 'I eat ',      $object->diet;
}
```

This function obviously works if you pass it a Cat object. It will also do the right thing for *any* object with the appropriate three accessors, no matter *how* that object provides those accessors and no matter *what kind* of object it is—Cat, Caterpillar, or Catbird—or even if the class uses Moose at all. show_vital_stats() cares that an invocant is valid only in that it supports three methods, name(), age(), and diet(), which take no arguments and each return something that can concatenate in a string context. Your code may have a hundred different classes with no obvious relationship among them, but they will all work with this function if they support the behavior it expects.

This property is *polymorphism*: you can substitute an object of one class for an object of another class if they provide the same external interface.

Duck Typing

Some languages and environments require you to imply or declare a formal relationship between two classes before allowing a program to substitute instances for each other. Perl makes no such requirement. You may treat any two instances with methods of the same name as equivalent. Some people call this *duck typing*, arguing that any object that can quack() is sufficiently duck-like that you can treat it as a duck.

Without object polymorphism, enumerating a zoo's worth of animals would be tedious. Similarly, you may already start to see how calculating the age of an ocelot or octopus should be the same as calculating the age of a Cat. Hold that thought.

Of course, the mere existence of a method called name() or age() does not by itself imply the behavior of that object. A Dog object may have an age(), which is an accessor such that you can discover $rodney is 13 but $lucky is 8. A Cheese object may have an age() method that lets you control how long to store $cheddar to sharpen it. age() may be an accessor in one class but not in another:

```
# how old is the cat?
my $years = $zeppie->age;

# store the cheese in the warehouse for six months
$cheese->age;
```

Sometimes it's useful to know *what* an object does and what that *means*.

Roles

A *role* is a named collection of behavior and state.[1] Whereas a class organizes behaviors and state into a template for objects, a role organizes a named collection of behaviors and state. You can instantiate a class but not a role. A role is something a class *does*.

Given an Animal that has an age and a Cheese that can age, one difference may be that Animal does the LivingBeing role, while Cheese does the Storable role:

```
package LivingBeing {
    use Moose::Role;
    requires qw( name age diet );
}
```

The requires keyword provided by Moose::Role allows you to list methods that this role requires of its composing classes. Anything that performs this role must supply the name(), age(), and diet() methods. The Cat class must declare that it performs the role:

```
package Cat {
    use Moose;

    has 'name', is => 'ro', isa => 'Str';
    has 'diet', is => 'rw', isa => 'Str';

    has 'birth_year',
        is      => 'ro',
        isa     => 'Int',
        default => sub { (localtime)[5] + 1900 };

    with 'LivingBeing';

    sub age { ... }
}
```

The with line causes Moose to *compose* the LivingBeing role into the Cat class. Composition ensures all of the attributes and methods of the role are part of the class. LivingBeing requires any composing class to provide methods named name(), age(), and diet(). Cat satisfies these constraints. If LivingBeing were composed into a class that didn't provide them, Moose would throw an exception.

Now all Cat instances will return a true value when queried if they provide the LivingBeing role. Cheese objects should not:

```
say 'Alive!' if $fluffy->DOES( 'LivingBeing' );
say 'Moldy!' if $cheese->DOES( 'LivingBeing' );
```

1. Many of the ideas come from Smalltalk traits: http://scg.unibe.ch/research/traits

This design technique separates the *capabilities* of classes and objects from the *implementation* of those classes and objects. As implied earlier, the birth year calculation behavior of the Cat class could itself be a role:

```
package CalculateAge::From::BirthYear {
    use Moose::Role;

    has 'birth_year',
        is      => 'ro',
        isa     => 'Int',
        default => sub { (localtime)[5] + 1900 };

    sub age {
        my $self = shift;
        my $year = (localtime)[5] + 1900;

        return $year - $self->birth_year;
    }
}
```

Extracting this role from Cat makes the useful behavior available to other classes. Now Cat can compose both roles:

```
package Cat {
    use Moose;

    has 'name', is => 'ro', isa => 'Str';
    has 'diet', is => 'rw';

    with 'LivingBeing', 'CalculateAge::From::BirthYear';
}
```

The age() method of CalculateAge::From::BirthYear satisfies the requirement of the LivingBeing role. Extracting the CalculateAge::From::BirthYear role has only changed the details of *how* Cat calculates an age. It's still a LivingBeing. Cat can choose to implement its own age or get it from somewhere else. All that matters is that it provides an age() that satisfies the LivingBeing constraint.

While polymorphism means that you can treat multiple objects with the same behavior in the same way, *allomorphism* means that an object may implement the same behavior in multiple ways. Pervasive allomorphism can reduce the size of your classes and increase the amount of code shared between them. It also allows you to name specific and discrete collections of behaviors—very useful for testing for capabilities instead of implementations.

Roles and DOES()

When you compose a role into a class, the class and its instances will return a true value when you call DOES() on them:

```
say 'This Cat is alive!' if $kitten->DOES( 'LivingBeing' );
```

Order Matters!

The with keyword used to apply roles to a class must occur *after* attribute declaration so that composition can identify any generated accessor methods. This is a side effect of the implementation of Moose and not an intrinsic feature of roles.

Inheritance

Perl's object system supports *inheritance*, which establishes a parent and child relationship between two classes such that a child specializes its parent. The child class behaves the same way as its parent—it has the same number and types of attributes and can use the same methods. It may have additional data and behavior, but you may substitute any instance of a child where code expects its parent. In one sense, a subclass provides the role implied by the existence of its parent class.

Consider a LightSource class that provides two public attributes (enabled and candle_power) and two methods (light and extinguish):

```
package LightSource {
    use Moose;

    has 'candle_power', is      => 'ro',
                        isa     => 'Int',
                        default => 1;

    has 'enabled', is      => 'ro',
                   isa     => 'Bool',
                   default => 0,
                   writer  => '_set_enabled';

    sub light {
        my $self = shift;
        $self->_set_enabled( 1 );
    }

    sub extinguish {
        my $self = shift;
        $self->_set_enabled( 0 );
    }
}
```

Note that enabled's writer option creates a private accessor usable within the class to set the value.

Roles versus Inheritance

Should you use roles or inheritance? Roles provide composition-time safety, better type checking, better factoring of code, and finer-grained control over names and behaviors, but inheritance is more familiar to experienced developers of other languages. Use inheritance when one class truly *extends* another. Use a role when a class needs additional behavior, especially when that behavior has a meaningful name.

Roles compare favorably to other design techniques such as mixins, multiple inheritance, and monkeypatching.[2]

Inheritance and Attributes

A subclass of LightSource could define an industrial-strength super candle with a hundred times the luminance:

```
package SuperCandle {
    use Moose;

    extends 'LightSource';

    has '+candle_power', default => 100;
}
```

extends takes a list of class names to use as parents of the current class. If that were the only line in this class, SuperCandle objects would behave in the same ways as LightSource objects. A SuperCandle instance would have both the candle_power and enabled attributes as well as the light() and extinguish() methods.

The + at the start of an attribute name (such as candle_power) indicates that the current class does something special with that attribute. Here the super candle overrides the default value of the light source, so any new SuperCandle created has a light value of 100 regular candles.

When you invoke light() or extinguish() on a SuperCandle object, Perl will look in the SuperCandle class for the method. If there's no method by that name in the child class, Perl will look at the parent class, then grandparent, and so on. In this case, those methods are in the LightSource class.

Attribute inheritance works similarly (see perldoc Class::MOP).

2. http://www.modernperlbooks.com/mt/2009/04/the-why-of-perl-roles.html

Method Dispatch Order

Perl's *dispatch* strategy controls how Perl selects the appropriate method to run for a method call. As you may have gathered from roles and polymorphism, much of OO's power comes from method dispatch.

Method dispatch order (or *method resolution order* or *MRO*) is obvious for single-parent classes. Look in the object's class, then its parent, and so on until you find the method—or run out of parents. Classes that inherit from multiple parents (*multiple inheritance*), such as a Hovercraft, which extends both Boat and Car, require trickier dispatch. Reasoning about multiple inheritance is complex, so avoid multiple inheritance when possible.

Perl uses a depth-first method resolution strategy. It searches the class of the *first* named parent and all of that parent's parents recursively before searching the classes of the current class's immediate parents. The mro pragma (Pragmas on page ?) provides alternate strategies, including the C3 MRO strategy, which searches a given class's immediate parents before searching any of their parents.

See perldoc mro for more details.

Inheritance and Methods

As with attributes, subclasses may override methods. Imagine a light that you cannot extinguish:

```
package Glowstick {
    use Moose;

    extends 'LightSource';

    sub extinguish {}
}
```

Calling extinguish() on a glowstick does nothing, even though LightSource's method does something. Method dispatch will find the subclass's method. You may not have meant to do this. When you do, use Moose's override to express your intention clearly.

Within an overridden method, Moose's super() allows you to call the overridden method:

```
package LightSource::Cranky {
    use Carp 'carp';
    use Moose;

    extends 'LightSource';
```

```
    override light => sub {
        my $self = shift;

        carp "Can't light a lit LightSource!" if $self->enabled;

        super();
    };

    override extinguish => sub {
        my $self = shift;

        carp "Can't extinguish unlit LightSource!" unless $self->enabled;

        super();
    };
}
```

This subclass adds a warning when trying to light or extinguish a light source that already has the current state. The super() function dispatches to the nearest parent's implementation of the current method, per the normal Perl method resolution order. (See perldoc Moose::Manual::MethodModifiers for more dispatch options.)

Inheritance and isa()

Perl's isa() method returns true if its invocant is or extends a named class. That invocant may be the name of a class or an instance of an object:

```
say 'Looks like a LightSource' if $sconce->isa( 'LightSource' );

say 'Hominidae do not glow' unless $chimpy->isa( 'LightSource' );
```

Moose and Perl OO

Moose provides many features beyond Perl's default OO system. Although you *can* build everything you get with Moose yourself (Blessed References on page ?) or cobble it together with a series of CPAN distributions, Moose is worth using. It's a coherent whole, with documentation, a mature and attentive development community, and a history of successful use in important projects.

Moose provides constructors, destructors, accessors, and encapsulation. You must do the work of declaring what you want, and you get safe and useful code in return. Moose objects can extend and work with objects from the vanilla Perl system.

While Moose is not a part of the Perl core, its popularity ensures that it's available on many OS distributions. Perl distributions such as Strawberry

Perl and ActivePerl also include it. Even though Moose is a CPAN module and not a core library, its cleanliness and simplicity make it essential to modern Perl programming.

Moose also allows *metaprogramming*—manipulating your objects through Moose itself. If you've ever wondered which methods are available on a class or an object or which attributes an object supports, this information is available:

```
my $metaclass = Monkey::Pants->meta;

say 'Monkey::Pants instances have the attributes:';
say $_->name for $metaclass->get_all_attributes;
say 'Monkey::Pants instances support the methods:';
say $_->fully_qualified_name for $metaclass->get_all_methods;
```

You can even see which classes extend a given class:

```
my $metaclass = Monkey->meta;

say 'Monkey is the superclass of:';
say $_ for $metaclass->subclasses;
```

See perldoc Class::MOP::Class for more information about metaclass operations and perldoc Class::MOP for Moose metaprogramming information.

Moose and its *meta-object protocol* (or MOP) offers the possibility of a better syntax for declaring and working with classes and objects in Perl. This is valid code:

```
use MooseX::Declare;

role LivingBeing { requires qw( name age diet ) }

role CalculateAge::From::BirthYear {
    has 'birth_year',
        is      => 'ro',
        isa     => 'Int',
        default => sub { (localtime)[5] + 1900 };

    method age {
        return (localtime)[5] + 1900 - $self->birth_year;
    }
}

class Cat with LivingBeing with CalculateAge::From::BirthYear {
    has 'name', is => 'ro', isa => 'Str';
    has 'diet', is => 'rw';
}
```

The MooseX::Declare CPAN distribution adds the class, role, and method keywords to reduce the amount of boilerplate necessary to write good object-oriented code in Perl. Note specifically the declarative nature of this example, as well as the lack of my $self = shift; in age().

Another good option is Moops, which allows you to write the following:

```
use Moops;

role LivingBeing {
    requires qw( name age diet );
}

role CalculateAge::From::BirthYear :ro {
    has 'birth_year',
        isa     => Int,
        default => sub { (localtime)[5] + 1900 };

    method age {
        return (localtime)[5] + 1900 - $self->birth_year;
    }
}

class Cat with LivingBeing with CalculateAge::From::BirthYear :ro {
    has 'name', isa => Str;
    has 'diet', is => 'rw';
}
```

The Svelte *Alces*

Moose isn't a small library, but it's powerful. The most popular alternative is Moo, a slimmer library that's almost completely compatible with Moose. Many projects migrate some or all code to Moo, where speed or memory use is an issue. Start with Moose; then see if Moo makes sense for you.