Objects

Writing large programs requires more discipline than writing small programs, due to the difficulty of managing all of the details of your program simultaneously. Abstraction (finding and exploiting similarities and near-similarities) and encapsulation (grouping specific details together and accessing them where they belong) are essential to managing this complexity.

Functions help, but functions by themselves aren't sufficient for the largest programs. Object orientation is a popular technique for grouping functions together into classes of related behaviors.

Perl 5's default object system is minimal. It's very flexible--you can build almost any other object system you want on top of it--but it provides little assistance for performing the most common tasks simply and easily.

Moose

Moose is a more complete object system for Perl 5. It builds on the existing Perl 5 system, and provides simpler defaults, better integration, and advanced features from several other languages, including Smalltalk, Common Lisp, and Perl 6. It's still worth learning the default Perl 5 object system, if only to write very simple programs where Moose is inappropriate or to maintain legacy code, but Moose is currently the best way to write object-oriented code in modern Perl 5.

Object orientation, or *object oriented programming*, is a way of managing programs by categorizing their components into discrete, unique entities. These are *objects*. In Moose terms, each object is an instance of a *class*, which serves as a template to describe any data the object contains as well as its specific behaviors.

Classes

A class in Perl 5 is part of a package, which provides a namespace in which to store class data.

```
{
    package Cat;

    use Moose;
}
```

Classes can be anonymous. See `perldoc Moose::Meta::Class`.

This `Cat` class appears to do nothing, but Moose does a lot of work to define the class and register it with Perl. With that done, you can create objects (or *instances*) of the `Cat` class:

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

The arrow syntax should look familiar. Just as an arrow dereferences a reference, an arrow calls a method on an object or class.

Methods

A *method* is a function associated with a class. It resembles a fully-qualified function call in a superficial sense, but it differs in two important ways. First, a method call always has an *invocant* on which to perform the method. In the case of creating the two `Cat` objects, the name of the class (`Cat`) is the invocant:

```
my $fuzzy = B<Cat>->new();
```

Second, a method call always involves a *dispatch* strategy. The dispatch strategy describes how the object system decides *which* method to call. This may seem obvious when there's only a `Cat`, but method dispatch is fundamental to the design of object systems.

The invocant of a method in Perl 5 is its first argument. For example, the `Cat` class could have a `meow()` method:

```
{
    package Cat;

    use Moose;

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

Now all `Cat` instances can wake you up in the morning because they haven't eaten yet:

```
my $alarm = Cat->new();
$alarm->meow();
$alarm->meow();
$alarm->meow();
```

By convention, invocants in Perl methods are lexical variables named `$self`, but this is merely pervasive convention. The example implementation of `meow()` does not use the invocant, so it's irrelevant once method dispatch has completed. In that sense, `meow()` is like `new()`; you can safely use the name of the class (`Cat`) as its invocant. This is a *class method*:

```
Cat->meow() for 1 .. 3;
```

Attributes

Every object in Perl 5 is unique. Objects can contain *attributes*, or private data associated with each object. You may also hear this described as *instance data* or *state*.

To define object attributes, describe them as part of the class:

```
{
    package Cat;

    use Moose;

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

In English, that line of code means "`Cat` objects have a `name` attribute. It's readable but not writable,

and it's a string." That single line of code creates an accessor method (`name()`) and allows you to pass a `name` parameter to the constructor:

```
use Cat;

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

Attributes do not *need* to have types, in which case Moose will skip all of the verification and validation for you:

```
{
    package Cat;

    use Moose;

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

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

This can be more flexible, but it can also lead to strange errors if someone tries to provide invalid data for an attribute. The balance between flexibility and correctness depends on your local coding standards and the type of errors you want to catch.

The Moose documentation uses parentheses to separate an attribute name from its characteristics:

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

Perl parses both that form and the form used in this book the same way. You *could* achieve the same effect by writing either:

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

... but in this case, extra punctuation adds clarity. The approach of the Moose documentation is most useful when dealing with multiple characteristics:

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

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

... but for the sake of simplicity of introduction, this book prefers to use less punctuation. Perl gives you the flexibility to choose whichever approach makes the intent of your code most clear.

If you mark an attribute as readable *and* writable (with `is => rw`), Moose will create a *mutator* method--a method you can use to change the value of an attribute:

```
{
    package Cat;

    use Moose;

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

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

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

Trying to use a `ro` accessor as a mutator will throw an exception:

Cannot assign a value to a read-only accessor at ....

When should you use `ro` and when `rw`? It's a matter of design, convenience, and purity. One school of thought (*immutability*) suggests making all instance data `ro` and passing all relevant data into the constructor. 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, rather than relying on someone to update the age of all cats manually.

This approach helps to consolidate all validation code and helps to ensure that all created objects have valid data. These are design goals worth considering, though Moose does not enforce any particular philosophy in this area.

Now that individual objects can have their own instance data, the value of object orientation may be more obvious. An object is a bookmark for the data it contains as well as the behavior appropriate to that data. An object is a collection of named data and behaviors. A class is the description of the data and behaviors that instances of that class possess.

Encapsulation

Moose allows you to declare *which* attributes class instances possess as well as how to treat those attributes. The examples shown so far do not describe *how* to store those attributes. This information is available if you really need it, but the declarative approach can actually improve your programs. In this way, Moose encourages *encapsulation*, or hiding the internal details of an object from external uses of that object.

Consider a change to the age of a `Cat`; instead of requesting that directly from the constructor, calculate the age of a `Cat` based on the year of its birth:

```
package Cat;

use Moose;

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

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

    B<< return $year - $self->birth_year(); >>
B<}>
```

While the syntax for *creating* `Cat` objects has changed, the syntax for *using* `Cat` objects has not. The `age()` method does the same thing it has always done, at least as far as all code outside of the `Cat` class understands. *How* it does that has changed, but that is a detail internal to the `Cat` class--encapsulated within that class itself.

The old syntax for *creating* `Cat` objects could remain in place; customize the generated `Cat` constructor to allow passing an `age` parameter and calculate `birth_year` appropriately from that. See `perldoc Moose::Manual::Attributes`.

This new approach to calculating `Cat` ages has another advantage; you can use *default attribute values* to reduce the code necessary to create a `Cat` object:

```
package Cat;

use Moose;

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

The `default` keyword on an attribute takes a function reference which returns the default value for that attribute when constructing a new object. If the constructor does not receive an appropriate value for that attribute, the object gets that default value instead. Now you can create a kitten:

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

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

You can also use a simple value, such as a number or string, as a default value. Use a function reference when you need to calculate something unique for each object, including a hash or array reference.

Polymorphism

A program which deals with one type of data and one type of behavior on that data benefits little from the use of objects. A well-designed object-oriented program should be capable of managing many types of data. When well designed classes encapsulate specific details of objects into the appropriate places, something curious happens to the rest of the program: it has the opportunity to become *less* specific.

In other words, moving the specifics of 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.

This is clearer with an example. Consider a function which describes 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();
    }
```

It's obvious (in context) that you can pass a `Cat` object to this function and get sane results. It's less obvious that you can pass other types of objects and get sane results. This is an important object orientation property called *polymorphism*, where you can substitute an object of one class for an object of another class if they provide the same external interface in the same way.

Any object of any class which provides the `name()`, `age()`, and `diet()` accessors will work with this function. The function is sufficiently generic that any object which respects this interface is a valid parameter.

Some languages and environments require a formal relationship between two classes before allowing a program to substitute instances of one class for another. Perl 5 provides ways to enforce these checks, but it does not require them. Its default ad-hoc system lets you treat any two instances with methods of the same name as equivalent enough. Some people call this *duck typing*, in the theory that any object which can `quack()` is sufficiently duck-like that you can treat it as a duck.

The benefit of the genericity in `show_vital_stats()` is that neither the specific *type* nor the implementation of the object provided matters. Any invocant is valid if it supports three methods, `name()`, `age()`, and `diet()` which take no arguments and each return something which can concatenate in a string context. You may have a hundred different classes in your code, none of which have any obvious relationship between each other, but if they conform to this expectation of behavior, they will work with this method.

This is an improvement over writing specific functions to extract and display this information for even a fraction of those hundred classes. This genericity requires less code, and using a well-defined interface as the mechanism to access this information means that any of those hundred classes can calculate that information in any way possible. The details of those calculations is where it matters most: in the bodies of the methods in the classes themselves.

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 `$rodney` is 8 but `$lucky` is 3. A `Cheese` object may have an `age()` method that lets you control how long to stow `$cheddar` so that it becomes sharp; in other words, `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. In other words, you need to understand its type.

Roles

A *role* is a named collection of behavior and state. A class is like a role, with the vital difference that you can instantiate a class, but not a role. While a class is primarily a mechanism for organizing behaviors and state into a template for objects, a role is primarily a mechanism for organizing behaviors and state into a named collection.

In short, a role is something a class does.

The difference between some sort of `Animal`--with a `name()`, an `age()`, and a preferred `diet()` --and `Cheese`--which can `age()` in storage--may be that the `Animal` performs a `LivingBeing` role, while the `Cheese` performs a `Storable` role.

While you *could* check that every object passed into `show_vital_stats()` is an instance of `Animal`, you lose some genericity that way. Instead, check that the object *performs* the `LivingBeing` role:

```
{
    package LivingBeing;

    use Moose::Role;

    requires qw( name age diet );
}
```

Anything which performs this role must supply the `name()`, `age()`, and `diet()` methods. This does not happen automatically; the `Cat` class must explicitly mark 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 => (localtime)[5] + 1900;

B<with 'LivingBeing';>

sub age { ... }
```

That single line has two functions. First, it tells Moose to note that the class performs the named role. Second, it composes the role into the class. This process checks that the class *somehow* provides all of the required methods and all of the required attributes without potential collisions.

The `Cat` class provides `name()` and `diet()` methods as accessors to named attributes. It also declares its own `age()` method.

The `with` keyword used to apply roles to a class must occur *after* attribute declaration so that composition can identify any generated accessor methods.

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

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

This design approach may seem like extra bookkeeping, but it separates the *capabilities* of classes and objects from the *implementation* of those classes and objects. The special behavior of the `Cat` class, where it stores the birth year of the animal and calculates the age directly, could itself be a role:

```
{
    package CalculateAgeFromBirthYear;
```

```
        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();
        }
    }
```

Moving this code out of the `Cat` class into a separate role makes it available to other classes. Now `Cat` can perform both roles:

```
    package Cat;

    use Moose;

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

    B<with 'LivingBeing', 'CalculateAgeFromBirthYear';>
```

The implementation of the `age()` method supplied by the `CalculateAgeFromBirthYear` satisfies the requirement of the `LivingBeing` role, and the composition succeeds. Checking that objects perform the `LivingBeing` role remains unchanged, regardless of *how* objects perform this role. A class could choose to provide its own `age()` method or obtain it from another role; that doesn't matter. All that matters is that it contains one. This is *allomorphism*.

Roles and DOES()

Applying a role to a class means that the class and its instances will return true when you call the `DOES()` method on them:

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

Inheritance

Another feature of Perl 5's object system is *inheritance*, where one class specializes another. This establishes a relationship between the two classes, where the child inherits attributes and behavior of the parent. As with two classes which provide the same role, you may substitute a child class for its parent. In one sense, a subclass provides the role implied by the existence of its parent class.

Recent experiments in role-based systems in Perl 5 demonstrate that you can replace almost every use of inheritance in a system with roles. The decision to use either one is largely a matter of familiarity. Roles provide composition-time safety, better type checking, better-factored and less coupled code, and finger-grained control over names and behaviors, but inheritance is more familiar to users of other languages. The design question is whether one class truly *extends* another or

whether it provides additional (or, at least, *different*) behavior.

Consider a `LightSource` class which provides two public attributes (`candle_power` and `enabled`) 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);
    }
}
```

The `_writer` option to the `enabled` attribute creates a private accessor usable within the class to set the value.

Inheritance and Attributes

Subclassing `LightSource` makes it possible to define a super candle which behaves the same way as `LightSource` but provides a hundred times the amount of light:

```
{
    package LightSource::SuperCandle;

    use Moose;

    B<extends 'LightSource'>;

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

The `extends` syntax takes a list of class names to use as parents of the current class. The + at the start of the `candle_power` attribute name indicates that the current class extends the declaration of the attribute. In this case, the super candle overrides the default value of the light source, so any new `SuperCandle` created has a light value of 100 candles. The other attribute and both methods are available on `SuperCandle` instances; when you invoke `light` or `extinguish` on such an instance, Perl will look in `LightSource::SuperCandle` for the method, then in the list of parents of the class. Ultimately it finds them in `LightSource`.

Attribute inheritance works in a similar fashion, except that the act of *constructing* the instance makes all of the appropriate attributes available in the proper fashion (see `perldoc Class::MOP`).

Method dispatch order is easy to understand in the case of single-parent inheritance. When a class has multiple parents (*multiple inheritance*), dispatch is less obvious. By default, Perl 5 provides a depth-first strategy of method resolution. It searches the class of the *first* named parent and all of its parents recursively before searching the classes of the subsequent named parents. This behavior is often confusing; avoid using multiple inheritance until you understand it and have exhausted all other alternatives. See `perldoc mro` for more details about method resolution and dispatch strategies.

Inheritance and Methods

You may override methods in subclasses. Imagine a light that you cannot extinguish:

```
{
    package LightSource::Glowstick;

    use Moose;

    extends 'LightSource';

    sub extinguish {};
}
```

All calls to the `extinguish` method for objects of this class will do nothing. Perl's method dispatch system will find this method and will not look for any methods of this name in any of the parent classes.

Sometimes an overridden method needs behavior from its parent as well. The `override` command tells Moose that the subclass deliberately overrides the named method. The `super()` function is available to dispatch from the overriding method to the overridden method:

```
{
    package LightSource::Cranky;

    use Carp;
    use Moose;

    extends 'LightSource';

    B<override> light => sub
    {
        my $self = shift;

        Carp::carp( "Can't light a lit light source!" )
            if $self->enabled;

        B<super()>;
    };

    B<override> extinguish => sub
    {
        my $self = shift;
```

```
            Carp::carp( "Can't extinguish an unlit light source!" )
                unless $self->enabled;

            B<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 5 method resolution order.

You can achieve the same behavior by using Moose method modifiers. See `perldoc Moose::Manual::MethodModifiers`.

Inheritance and isa()

Inheriting from a parent class means that the child class and all of its instances will return true when you call the `isa()` method on them:

```
say 'Looks like a LightSource' if     $sconce->isa( 'LightSource' );
say 'Monkeys do not glow'      unless $chimpy->isa( 'LightSource' );
```

Sub-types, Type coercion, lazy yet immutable data

To me this is one of the most exciting areas of development of Perl. sub-typing, along with type coercion rules (both in `Moose::Util::TypeConstraints`), type libraries (`MooseX::Types`) and read-only objects that you can still construct lazily allow you to efficiently layer assertions in a declarative fashion, into very easy to debug code. (Read-only objects will also later win when it comes to threading, STM, etc of course as people port their designs to Perl 6, but no need to scare readers with such crazy talk).

Moose and Perl 5 OO

Moose provides many features you'd otherwise have to build for yourself with the default object orientation of Perl 5. While you *can* build everything you get with Moose yourself (see *blessed_references*), or cobble it together with a series of CPAN distributions, Moose is a coherent package which just works, includes good documentation, is part of many successful projects, and is under active development by an attentive and talented community.

By default, with Moose objects you do not have to worry about constructors and destructors, accessors, and encapsulation. Moose objects can extend and work with objects from the vanilla Perl 5 system. You also get *metaprogramming*--a way of accessing the implementation of the system through the system itself--and the concomitant extensibility. If you've ever wondered which methods are available on a class or an object or which attributes an object supports, this metaprogramming information is available with Moose:

```
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 5. This is valid Perl 5 code:

```
use MooseX::Declare;

B<role> LivingBeing { requires qw( name age diet ) }

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

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

B<class Cat with LivingBeing with CalculateAgeFromBirthYear>
{
    has 'name', is => 'ro', isa => 'Str';
    has 'diet', is => 'rw';
}
```

The `MooseX::Declare` extension from the CPAN uses a clever module called `Devel::Declare` to add new syntax to Perl 5, specifically for Moose. The `class`, `role`, and `method` keywords reduce the amount of boilerplate necessary to write good OO code in Perl 5. Note specifically the declarative nature of this example, as well as the now unnecessary `my $self = shift;` line at the start of the `age` method.

Certainly worth noting that the `Devel::Declare` infrastructure is essentially core in Perl 5.12, though I am unaware of `MooseX::` modules like `MooseX::Method::Signatures` being ported to that yet.

One drawback of this approach is that you must be able to install CPAN modules (or a custom Perl 5 distribution such as Strawberry or Chocolate Perl which may include them for you), but in comparison to Perl 5's built-in object orientation, the advantage in cleanliness and simplicity of Moose should be obvious.

See `perldoc Moose::Manual` for more information on using Moose.

Blessed References

Perl 5's default object system is deliberately minimal. Three simple rules combine to form the simple--though effective--basic object system:

* A class is a package.

* A method is a function.

* A (blessed) reference is an object.

You've already seen the first two rules with Moose. The third rule is new. The `bless` keyword associates the name of a class with a reference, such that any method invocation performed on that reference uses the associated class for resolution. That sounds more complicated than it is.

Though these rules explain Perl 5's underlying object system, they are somewhat more minimal in practice than may be practical, especially for larger projects. In particular, they offer few facilities for metaprogramming (using APIs to manipulate the program itself).

Moose (*moose*) is a better choice for serious, modern Perl programs larger than a couple of hundred lines, but you will likely encounter bare-bones Perl 5 OO in existing code.

The default Perl 5 object constructor is a method which creates and blesses a reference. By convention, constructors have the name `new()`, but this is not a requirement. Constructors are also almost always *class methods*:

```
sub new
{
    my $class = shift;
    bless {}, $class;
}
```

`bless` takes two arguments, the reference to associate with a class and the name of a class. You may use `bless` outside of a constructor or a class--though abstraction recommends the use of the method. The class name does not have to exist yet.

By design, this constructor receives the class name as the method's invocant. It's possible, but inadvisable, to hard-code the name of a class directly. The parametric constructor allows reuse of the method through inheritance, delegation, or exporting.

The type of reference makes no difference when invoking methods on the object. It only governs how the object stores *instance data*--the object's own information. Hash references are most common, but you can bless any type of reference:

```
my $array_obj  = bless [], $class;
my $scalar_obj = bless \$scalar, $class;
my $sub_obj    = bless \&some_sub, $class;
```

Where classes built with Moose define their own object attributes declaratively, Perl 5's default OO is lax. A class representing basketball players which stores jersey number and position might use a constructor like:

```
package Player;
```

```
sub new
{
    my ($class, %attrs) = @_;

    bless \%attrs, $class;
}
```

... and create players with:

```
my $joel = Player->new(
    number   => 10,
    position => 'center',
);

my $jerryd = Player->new(
    number   => 4,
    position => 'guard',
);
```

Within the body of the class, methods can access hash elements directly:

```
sub format
{
    my $self = shift;
    return '#' . $self->{number} . ' plays ' . $self->{position};
}
```

Yet so can any code outside of the class. This violates encapsulation--in particular, it means that you can never change the object's internal representation without breaking external code or performing hacks--so it's safer to provide accessor methods:

```
sub number   { return shift->{number}   }
sub position { return shift->{position} }
```

Even with two attributes, Moose is much more appealing in terms of code you don't have to write.

Moose's default behavior of accessor generation encourages you to do the right thing with regard to encapsulation as well as genericity.

Method Lookup and Inheritance

Besides instance data, the other part of objects is method dispatch. Given an object (a blessed reference), a method call of the form:

```
my $number = $joel->number();
```

... looks up the name of the class associated with the blessed reference `$joel`. In this case, the class is `Player`. Next, Perl looks for a function named `number` in the `Player` package. If the `Player` class inherits from another class, Perl looks in the parent class (and so on and so on) until it finds a `number` method. If one exists, Perl calls it with `$joel` as an invocant.

Moose classes store their inheritance information in a metamodel. Each class of a blessed reference stores information about its parents in a package global variable named `@ISA`. The method dispatcher looks in a class's `@ISA` to find the names of parent classes in which to search for the appropriate method. Thus, an `InjuredPlayer` class might contain `Player` in its `@ISA`. You could write this relationship as:

```
package InjuredPlayer;

@InjuredPlayer::ISA = 'Player';
```

Many existing Perl 5 projects do this, but it's easier and simpler to use the `parent` pragma instead:

```
package InjuredPlayer;

use parent 'Player';
```

Perl 5.10 added `parent` to supersede the `base` pragma added in Perl 5.004_4. If you can't use Moose, use `parent`.

You may inherit from multiple parent classes:

```
package InjuredPlayer;

use parent qw( Player Hospital::Patient );
```

Perl 5 has traditionally preferred a depth-first search of parents when resolving method dispatch. That is to say, if `InjuredPlayer` inherits from both `Player` and `Hospital::Patient`, a method call on an `InjuredPlayer` instance will dispatch first to `InjuredPlayer`, then `Player`, then any of `Player`'s parents before dispatching in `Hospital::Patient`.

Perl 5.10 added a pragma called `mro` which allows you to use a different method resolution scheme called C3. While the specific details can get complex in the case of complex multiple inheritance hierarchies, the important difference is that method resolution will visit all children of a parent before visiting the parent.

While other techniques (such as *roles* and Moose method modifiers) allow you to avoid multiple inheritance, the `mro` pragma can help avoid surprising behavior with method dispatch. Enable it in your class with:

```
package InjuredPlayer;

use mro 'c3';
```

Unless you're writing a complex framework with multiple interoperable plugins, you likely never need to use this.

AUTOLOAD

If there is no applicable method in the invocant's class or any of its superclasses, Perl 5 will next look for an `AUTOLOAD` function in every class according to the selected method resolution order. Perl will invoke any `AUTOLOAD` it finds to provide or decline the desired method. See *autoload* for more details.

As you might expect, this can get quite complex in the face of multiple inheritance and multiple potential `AUTOLOAD` targets.

Method Overriding and SUPER

You may override methods in the default Perl 5 OO system as well as in Moose. Unfortunately, core Perl 5 provides no mechanism for indicating your *intent* to override a parent's method. Worse yet, any function you pre-declare, declare, or import into the child class may override a method in the parent class simply by existing and having the same name. While you may forget to use the `override` system of Moose, you have no such protection (even optional) in the default Perl 5 OO system.

To override a method in a child class, declare a method of the same name as the method in the parent. Within an overridden method, call the parent method with the `SUPER::` dispatch hint:

```
sub overridden
{
    my $self = shift;
    warn "Called overridden() in child!";
    return $self->SUPER::overridden( @_ );
}
```

The `SUPER::` prefix to the method name tells the method dispatcher to dispatch to the named method in a *parent* implementation. You may pass any arguments to it you like, but it's safest to reuse `@_`.

Beware that this dispatcher relies on the package into which the overridden method was originally compiled when redispatching to a parent method. This is a long-standing misfeature retained for the sake of backwards compatibility. If you export methods into other classes or compose roles into classes manually, you may run afoul of this feature. The `SUPER` module on the CPAN can work around this for you. Moose handles it nicely as well.

Strategies for Coping with Blessed References

Avoid `AUTOLOAD` where possible. If you *must* use it, use forward declarations of your functions (*functions*) to help Perl know which `AUTOLOAD` will provide the method implementation.

Use accessor methods rather than accessing instance data directly through the reference. This applies even within the bodies of methods within the class itself. Generating these yourself can be tedious; if you can't use Moose, consider using a module such as `Class::Accessor` to avoid repetitive boilerplate.

Expect that someone, somewhere will eventually need to subclass (or delegate to or reimplement the interface of) your classes. Make it easier for them by not assuming details of the internals of your code, by using the two-argument form of `bless`, and by breaking your classes into the smallest responsible units of code.

Do not mix functions and methods in the same class.

Use a single *.pm* file for each class, unless the class is a small, self-contained helper used from a single place.

Consider using Moose and `Any::Moose` instead of bare-bones Perl 5 OO; they can interact with vanilla classes and objects with ease, alleviate almost of the tedium of declaring classes, and provide more and better features.

Reflection

Reflection (or *introspection*) is the process of asking a program about itself as it runs. Even though you can write many useful programs without ever having to use reflection, techniques such as metaprogramming (*code_generation*) benefit from a deeper understanding of which entities are in the system.

`Class::MOP` (*class_mop*) simplifies many reflection tasks for object systems, but many useful

programs do not use objects pervasively, and many useful programs do not use `Class::MOP`. Several idioms (*idioms*) exist for performing reflection effectively in the absence of such a formal system.

Checking that a Package Exists

To check that a package exists somewhere in the system--that is, if some code somewhere has executed a `package` directive with a given name--check that the package inherits from `UNIVERSAL` by testing that the package can perform the `can()` method:

```
say "$pkg exists" if eval { $pkg->can( 'can' ) };
```

Although you *may* use packages with the names `0` and `''`... only if you define them symbolically, as these are *not* identifiers forbidden by the Perl 5 parser., the `can()` method will throw a method invocation exception if you use them as invocants. The `eval` block catches such an exception.

You *could* also grovel through the symbol table, but this approach is quicker and easier to understand.

Checking that a Class Exists

Because Perl 5 makes no strong distinction between packages and classes, the same technique for checking the existence of a package works for checking that a class exists. There is no generic way for determining if a package is a class. You *can* check that the package `can()` perform `new()`, but there is no guarantee that any `new()` found is a method, nor a constructor.

Checking that a Module Has Loaded

If you know the name of a module, you can check that Perl believes it has loaded that module from disk by looking in the `%INC` hash. This hash corresponds to `@INC`; when Perl 5 loads code with `use` or `require`, it stores an entry in `%INC` where the key is the pathified name of the module to load and the value is the full path on disk to that module. In other words, loading `Modern::Perl` effectively does:

```
$INC{'Modern/Perl.pm'} =>
    '/path/to/perl/lib/site_perl/5.12.1/Modern/Perl.pm';
```

The details of the path will vary depending on your installation, but for the purpose of testing that Perl has successfully loaded a module, you can convert the name of the module into the canonical file form and test for existence within `%INC`:

```
sub module_loaded
{
    (my $modname = shift) =~ s!::!/!g;
    return exists $INC{ $modname . '.pm' };
}
```

Nothing prevents other code from manipulating `%INC` itself. Depending on your paranoia level, you may check the path and the expected contents of the package yourself, but modules with good reasons for manipulating this variable (such as `Test::MockObject` or `Test::MockModule`) may do so. Code which manipulates `%INC` for poor reasons deserves replacing.

Checking the Version of a Module

There is no guarantee that a given module provides a version. Even so, all modules inherit from `UNIVERSAL` (*universal*), so they all have a `VERSION()` method available:

```
my $mod_ver = $module->VERSION();
```

If the given module does not override `VERSION()` or contain a package variable `$VERSION`, the method will return an undefined value. Likewise, if the module does not exist, the method call will fail.

Checking that a Function Exists

The simplest mechanism by which to determine if a function exists is to use the `can()` method on the package name:

```
say "$func() exists" if $pkg->can( $func );
```

Note that `$pkg` must be a valid invocant; wrap the method call in an `eval` block if you have any doubts about its validity. Beware that a function implemented in terms of `AUTOLOAD()` (*autoload*) may report the wrong answer if the function's package does not also override `can()` correctly. This is a bug in the other package.

You may use this technique to determine if a module's `import()` has imported a function into the current namespace:

```
say "$func() imported!" if __PACKAGE__->can( $func );
```

You may also root around in the symbol table and typeglobs to determine if a function exists, but this mechanism is simpler and easier to explain.

Checking that a Method Exists

There is no generic way to determine whether a given function is a function or a method. Some functions perform dual duty as both functions and methods; though this is overly complex and usually a mistake, it is an allowed feature.

Rooting Around in Symbol Tables

A Perl 5 symbol table is a special type of hash, where the keys are the names of package global symbols and the values are typeglobs. A *typeglob* is a core data structure which can contain any or all of a scalar, an array, a hash, a filehandle, and a function. Perl 5 uses typeglobs internally when it looks up these variables.

You can access a symbol table as a hash by appending double-colons to the name of the package. For example, the symbol table for the MonkeyGrinder package is available as `%MonkeyGrinder::`.

You *can* test the existence of specific symbol names within a symbol table with the `exists` operator (or manipulate the symbol table to *add* or *remove*) symbols, if you like). Yet be aware that certain changes to the Perl 5 core have modified what exists by default in each typeglob entry. In particular, earlier versions of Perl 5 have always provided a default scalar variable for every typeglob created, while modern versions of Perl 5 do not.

See the "Symbol Tables" section in `perldoc perlmod` for more details, then prefer the other techniques in this section for reflection.

Advanced OO Perl

Creating and using objects in Perl 5 with Moose (*moose*) is easy. *Designing* good object systems is not. Additional capabilities for abstraction also offer possibilities for obfuscation. Only practical experience can help you understand the most important design techniques... but several principles can guide you.

Favor Composition Over Inheritance

Novice OO designs often overuse inheritance. It's common to see class hierarchies which try to model all of the behavior for entities within the system in a single class. This adds a conceptual overhead to understanding the system, because you have to understand the hierarchy, and it adds technical weight to every class, because conflicting responsibilities and methods may get in the way of necessary behaviors or future modifications.

The encapsulation provided by classes offers better ways to organize code. You don't have to inherit

from superclasses to provide behavior to users of objects. A `Car` object does not have to inherit from a `Vehicle::Wheeled` object; it can contain several `Wheel` objects as instance attributes.

Decomposing complex classes into smaller, focused entities (whether classes or roles) improves encapsulation and reduces the possibility that any one class or role will grow to do too much. Smaller, simpler, and better encapsulated entities are easier to understand, test, and maintain.

Single Responsibility Principle

When you design your object system, model the problem in terms of responsibilities, or reasons why each specific entity may need to change. For example, an `Employee` object may represent specific information about a person's name, contact information, and other personal data, while a `Job` object may represent business responsibilities. A simple design might conflate the two into a single entity, but separating them allows the `Employee` class to consider only the problem of managing information specific to who the person is and the `Job` class to represent what the person does. (Two `Employees` may have a `Job`-sharing arrangement, for example.)

When each class has a single responsibility, you can improve the encapsulation of class-specific data and behaviors and reduce coupling between classes.

Don't Repeat Yourself

Complexity and duplication complicate development and maintenance activities. The DRY principle (Don't Repeat Yourself) is a reminder to seek out and to eliminate duplication within the system. Duplication exists in many forms, in data as well as in code. If you find yourself repeating configuration information, user data, and other artifacts within your system, instead find a canonical, single representation of that information, then generate all of the other artifacts from that representation.

This principle helps to reduce the possibility that important parts of your system can get unsynchronized, and helps you to find the optimal representation of the system and its data.

Liskov Substitution Principle

The Liskov substitution principle suggests that subtypes of a given type (specializations of a class or role or subclasses of a class) should be substitutable for the parent type without narrowing the types of data they receive or expanding the types of data they produce. In other words, they should be as general as or more general at what they expect and as specific as or more specific about what they produce.

Immutability

A common pattern among programmers new to object orientation is to treat objects as if they were bundles of records which use methods to get and set internal values. While this is simple to implement and easy to understand, it can lead to the unfortunate temptation to spread the behavioral responsibilities among individual classes throughout the system.

The most useful technique to working with objects effectively is to tell them what to do, not how to do it. If you find yourself accessing the instance data of objects (even through accessor methods), you may have too much access to the responsibilities of the class.

One approach to preventing this behavior is to consider objects as immutable. Pass in all of the relevant configuration data to their constructors, then disallow any modifications of this information from outside the class. Do not expose any methods to mutate instance data.

Some designs go as far as to prohibit the modification of instance data *within* the class itself, though this is much more difficult to achieve.