Every large program has several levels of design. At the bottom, you have specific details about the problem you're solving. At the top levels, you have to organize the code so it makes sense. Our only hope to manage this complexity is to exploit *abstraction* (treating similar things similarly) and *encapsulation* (grouping related details together).

Functions alone are insufficient for large problems. Several techniques group functions into units of related behaviors; you've already seen higher-order functions. Another popular technique is *object orientation* (OO), or *object oriented programming* (OOP), where programs work with *objects*--discrete, unique entities with their own identities.

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 as well as 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.

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*), 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. Create objects (or *instances*) of the `Cat` class with the 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 $choco = B<Cat>->new;
B<$choco>->sleep_on_keyboard;
```

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

```
package Cat {
    use Moose;

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

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 which 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()`) which do not 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*. Define an attribute by declaring it as part of the class:

```
package Cat {
    use Moose;

    B<< 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 is a string." The first argument, `'name'`, is the attribute's name. The `is => 'ro'` pair of arguments declares that this attribute is `r`ead `o`nly, 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 `Str`ing.

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 documentation uses parentheses to separate attribute names and characteristics:

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

This is equivalent to:

```
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,
);
```

... while this book prefers a low-punctuation approach for simple declarations. Choose the style which 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 which isn't a string, attributes do not *require* types. In that case, anything goes:

```
package Cat {
    use Moose;

    has 'name', is => 'ro', isa => 'Str';
    B<< 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 which can change that attribute's value:

```
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;
```

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*). 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';
    B<< has 'birth_year',  is => 'ro', isa => 'Int'; >>

    B<sub age {>
        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. Outside of `Cat`, `age()` behaves as it always has. *How* it works is a detail hidden inside the `Cat` class.

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`.

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';

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

The `default` keyword on an attribute uses a function reference (or a literal string or number) which 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.

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 which 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 which can concatenate in a string context. Your code may have a hundred different classes with no obvious relationship between 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.

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 which 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 stow `$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. (Many of the ideas come from Smalltalk traits http://scg.unibe.ch/research/traits.) While 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` which has an age and a `Cheese` which can age, one difference may be that `Animal` does the `LivingBeing` role, while the `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 which does 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 };
    B<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 which did not provide those methods, Moose would throw an exception.

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.

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' );
```

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';

        B<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()` which 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' );
```

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.

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

Consider a `LightSource` class which 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.

Inheritance and Attributes

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

```
    package SuperCandle {
        use Moose;

        B<extends 'LightSource'>;

        has 'B<+>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`).

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 which 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*) 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';

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

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

            B<super()>;
        };

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

            carp "Can't extinguish unlit LightSource!" 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 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. While you *can* build everything you get with Moose yourself (*blessed_references*), or cobble it together with a series of CPAN distributions, Moose is worth using. It is 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;

    B<role> LivingBeing { requires qw( name age diet ) }
    B<role> CalculateAge::From::BirthYear {
        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 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:

```
    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';
    }
```

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.

Blessed References

Perl's core object system is deliberately minimal. It has only three rules:

* A class is a package.

* A method is a function.

* A (blessed) reference is an object.

You can build anything else out of those three rules. This minimalism can be impractical for larger projects--in particular, the possibilities for greater abstraction through metaprogramming (*code_generation*) are awkward and limited. Moose (*moose*) is a better choice for modern programs larger than a couple of hundred lines, although plenty of legacy code uses Perl's default OO.

You've seen the first two rules already. The `bless` builtin associates the name of a class with a reference. That reference is now a valid invocant. Perl will perform method dispatch on it.

A constructor is a method which creates and blesses a reference. By convention, constructors are named `new()`. Constructors are also almost always *class methods*.

`bless` takes two operands, a reference and a class name, and evaluates to the reference. The reference may be any valid reference, empty or not. The class does not have to exist yet. You may even use `bless` outside of a constructor or a class, but you violate encapsulation to expose the details of object construction outside of a constructor. A constructor can be as simple as:

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

By design, this constructor receives the class name as the method's invocant. You may also hard-code the name of a class at the expense of flexibility. A parametric constructor--one which relies on the invocant to determine the class name--allows reuse through inheritance, delegation, or exporting.

The type of reference used is relevant only to how the object stores its own *instance data*. It has no other effect on the resulting object. 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 $func_obj   = bless \&some_func, $class;
```

Moose classes define object attributes declaratively, but Perl'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 $damian = Player->new( number  => 0, position => 'guard'  );
```

The class's methods can access object attributes as hash elements directly:

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

... but so can any other code, so any change to the object's internal representation may break other code. Accessor methods are safer:

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

... and now you're starting to write yourself what Moose gives you for free. Better yet, Moose encourages people to use accessors instead of direct attribute access by generating the accessors itself. You won't see them in your code. Goodbye, temptation.

Method Lookup and Inheritance

Given 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 `$joel`--in this case, `Player`. Next, Perl looks for a function named `number()` in `Player`. (Remember that Perl makes no distinction between functions in a namespace and methods.) If no such function exists and if `Player` extends a parent class, Perl looks in the parent class (and so on and so on) until it finds a `number()`. If Perl finds `number()`, it calls that method with `$joel` as an invocant. You've seen this before with Moose; it works the same way here.

The `namespace::autoclean` CPAN module can help avoid unintentional collisions between imported functions and methods.

Moose provides `extends` to track inheritance relationships, but Perl uses a package global variable named `@ISA`. The method dispatcher looks in each class's `@ISA` to find the names of its parent classes. If `InjuredPlayer` extends `Player`, you might write:

```
    package InjuredPlayer {
        @InjuredPlayer::ISA = 'Player';
    }
```

The `parent` pragma (*pragmas*) is cleaner:

```
    package InjuredPlayer {
        use parent 'Player';
    }
```

Moose has its own metamodel which stores extended inheritance information. This allows Moose to provide additional metaprogramming opportunities.

You may inherit from multiple parent classes:

```
    package InjuredPlayer; {
        use parent qw( Player Hospital::Patient );
    }
```

... though the caveats about multiple inheritance and method dispatch complexity apply. Consider instead roles (*roles*) or Moose method modifiers.

AUTOLOAD

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

In the case of multiple inheritance, `AUTOLOAD()` can be very difficult to understand.

Method Overriding and SUPER

As with Moose, you may override methods in basic Perl OO. Unlike Moose, Perl provides no mechanism for indicating your *intent* to override a parent's method. Worse yet, any function you predeclare, declare, or import into the child class may silently override a method in the parent class. Even if you forget to use Moose's `override` system, at least it exists. Basic Perl OO offers no such protection.

To override a parent method in a child class, declare a method of the same name. 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 an overridden method of the appropriate name. You can provide your own arguments to the overridden method, but most code reuses `@_`. Be careful to `shift` off the invocant if you do.

`SUPER::` has a confusing misfeature: it dispatches to the parent of the package into which the overridden method was *compiled*. If you've imported this method from another package, Perl will happily dispatch to the *wrong* parent. The desire for backwards compatibility has kept this misfeature in place. The `SUPER` module from the CPAN offers a workaround. Moose's `super()` does not suffer the same problem.

Strategies for Coping with Blessed References

Blessed references may seem simultaneously minimal and confusing. Moose is much easier to use, so use it whenever possible. If you do find yourself maintaining code which uses blessed references, or if you can't convince your team to use Moose in full yet, you can work around some of the problems of blessed references with a few rules of thumb:

* 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.

* Follow Perl standards, such as naming constructors `new()` and using `$self` as the invocant name.

* Use accessor methods pervasively, even within methods in your class. A module such as `Class::Accessor` helps to avoid repetitive boilerplate.

* Avoid `AUTOLOAD()` where possible. If you *must* use it, use function forward declarations (*functions*)

to avoid ambiguity.

* 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.

* Use helper modules such as `Role::Tiny` to allow better use and reuse.

Reflection

Reflection (or *introspection*) is the process of asking a program about itself as it runs. By treating code as data you can manage code in the same way that you manage data. That sounds like a truism, but it's an important insight into modern programming. It's also a principle behind code generation (*code_generation*).

Moose's `Class::MOP` (*class_mop*) simplifies many reflection tasks for object systems. Several other Perl idioms help you inspect and manipulate running programs.

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 by looking in the `%INC` hash. When Perl loads code with `use` or `require`, it stores an entry in `%INC` where the key is the file path 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'} = '.../lib/site_perl/5.22.1/Modern/Perl.pm';
```

The details of the path will vary depending on your installation. To test that Perl has successfully loaded a module, convert the name of the module into the canonical file form and test for that key's existence within `%INC`:

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

As with `@INC`, any code anywhere may manipulate `%INC`. Some modules (such as `Test::MockObject` or `Test::MockModule`) manipulate `%INC` for good reasons. Depending on your paranoia level, you may check the path and the expected contents of the package yourself.

The `Class::Load` CPAN module's `is_class_loaded()` function does all of this for you without making you manipulate `%INC`.

Checking that a Package Exists

To check that a package exists somewhere in your program--if some code somewhere has executed a `package` directive with a given name--check that the package inherits from `UNIVERSAL`. Anything which extends `UNIVERSAL` must somehow provide the `can()` method (whether by inheriting it from `UNIVERSAL` or overriding it). If no such package exists, Perl will throw an exception about an invalid invocant, so wrap this call in an `eval` block:

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

An alternate approach is to grovel through Perl's symbol tables. You're on your own here.

Checking that a Class Exists

Because Perl makes no strong distinction between packages and classes, the best you can do without Moose is to check that a package of the expected class name exists. You *can* check that the package `can()` provide `new()`, but there is no guarantee that any `new()` found is either a method or a constructor.

Checking a Module Version Number

Modules do not have to provide version numbers, but every package inherits the `VERSION()` method from the universal parent class `UNIVERSAL` (*universal*):

```
my $version = $module->VERSION;
```

`VERSION()` returns the given module's version number, if defined. Otherwise it returns `undef`. If the module does not exist, the method will likewise return `undef`.

Checking that a Function Exists

To check whether a function exists in a package, call `can()` as a class method on the package name:

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

Perl will throw an exception unless `$pkg` is 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 has not predeclared the function or overridden `can()` correctly. This is a bug in the other package.

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 );
```

As with checking for the existence of a package, you *can* root around in symbol tables yourself, if you have the patience for it.

Checking that a Method Exists

There is no foolproof way for reflection to distinguish between a function or a method.

Rooting Around in Symbol Tables

A *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 an internal data structure which can contain a scalar, an array, a hash, a filehandle, and a function--any or all at once.

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 core have modified the details of what typeglobs store and when and why.

See the "Symbol Tables" section in `perldoc perlmod` for more details, then consider the other techniques explained earlier instead. If you really need to manipulate symbol tables and typeglobs, use the `Package::Stash` CPAN module.

Advanced OO Perl

Creating and using objects in Perl with Moose (*moose*) is easy. *Designing* good programs is not. It's as easy to overdesign a program as it is to underdesign it. 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 to reuse code and to exploit polymorphism. The result is a deep class hierarchy with responsibilities scattered all over the place. Maintaining this code is difficult--who knows where to add or edit behavior? What happens when code in one place conflicts with code declared elsewhere?

Inheritance is only one of many tools for OO programmers. It's not always the right tool. It's often the wrong tool. A `Car` may extend `Vehicle::Wheeled` (an *is-a relationship*), but `Car` may better *contain* several `Wheel` objects as instance attributes (a *has-a relationship*).

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

Single Responsibility Principle

When you design your object system, consider the responsibilities of each entity. 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. Separating these entities in terms of their responsibilities 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, or one `Employee` may have the CFO and the COO `Jobs`.)

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

Don't Repeat Yourself

Complexity and duplication complicate development and maintenance. The *DRY* principle (Don't Repeat Yourself) is a reminder to seek out and to eliminate duplication within the system. Duplication exists in data as well as in code. Instead of repeating configuration information, user data, and other important artifacts of your system, create a single, canonical representation of that information from which you can generate the other artifacts.

This principle helps you to find the optimal representation of your system and its data and reduces the possibility that duplicate information will get out of sync.

Liskov Substitution Principle

The Liskov substitution principle suggests that you should be able to substitute a specialization of a class or a role for the original without violating the original's API. In other words, an object should be as or more general with regard to what it expects and at least as specific about what it produces as the object it replaces.

Imagine two classes, `Dessert` and its child class `PecanPie`. If the classes follow the Liskov substitution principle, you can replace every use of `Dessert` objects with `PecanPie` objects in the test suite, and everything should pass. See Reg Braithwaite's "IS-STRICTLY-EQUIVALENT-TO-A" http://weblog.raganwald.com/2008/04/is-strictly-equivalent-to.html. for more details.

Subtypes and Coercions

Moose allows you to declare and use types and extend them through subtypes to form ever more specialized descriptions of what your data represents and how it behaves. These type annotations help verify that the function and method parameters are correct--or can be coerced into the proper data types.

For example, you may wish to allow people to provide dates to a `Ledger` entry as strings while representing them as `DateTime` instances internally. You can do this by creating a Date type and adding a coercion from string types. See `Moose::Util::TypeConstraints` and `MooseX::Types` for more information.

Immutability

With a well-designed object, you tell it *what to do*, not *how to do it*. If you find yourself accessing object instance data (even through accessor methods) outside of the object itself, you may have too much access to an object's internals.

OO novices often treat objects as if they were bundles of records which use methods to get and set internal values. This simple technique leads to the unfortunate temptation to spread the object's responsibilities throughout the entire system.

You can prevent inappropriate access by making your objects immutable. Provide the necessary data to their constructors, then disallow any modifications of this information from outside the class. Expose no methods to mutate instance data--make all of your public accessors read-only and use internal attribute writers sparingly. Once you've constructed such an object, you know it's always in a valid state. You can never modify its data to put it in an invalid state.

This takes tremendous discipline, but the resulting systems are robust, testable, and maintainable. Some designs go as far as to prohibit the modification of instance data *within* the class itself.