12

Object Oriented Perl

WHAT YOU WILL LEARN IN THIS CHAPTER:

- Understanding what an object is
- Learning the three rules of Perl's OO system
- Creating a class
- How to subclass a class
- Overloading classes
- Learning OO traps for the unwary

WROX.COM CODE DOWNLOAD FOR THIS CHAPTER

The wrox.com code downloads for this chapter are found at `http://www.wrox.com/remtitle.cgi?isbn=1118013847` on the Download Code tab. The code for this chapter is divided into the following major examples:

- `example_12_1_shopper.pl`
- `example_12_2_episode.pl`
- `lib/Shopper/Personal.pm`
- `lib/TV/Episode.pm`
- `lib/TV/Episode/Broadcast.pm`
- `lib/TV/Episode/OnDemand.pm`
- `lib/TV/Episode/Version.pm`
- `listing_12_1_episode.pl`

Chapter 10 mentioned that knowledge of the `sort`, `map`, and `grep` functions is sort of a litmus test that some programmers use to know if a Perl developer is at least at an intermediate level. Knowledge of object-oriented programming (often referred to as OOP, or just OO) is your first step toward being an advanced Perl developer. Many languages support OO programming, and learning about it in Perl will help you in many other languages.

Two chapters discuss OOP. This chapter describes Perl's built-in OO tools. They're minimal, but this minimalism gives you a lot of freedom. You need to understand how Perl's built-in OO works because much of the Perl software in the wild is written with this.

The next chapter, Chapter 13, covers `Moose` which is an incredibly powerful object system built on top of Perl's OO tools. It's so powerful that it's rapidly becoming Perl's de facto OO system for many developers and companies and has had a large influence over the development of Perl.

WHAT ARE OBJECTS? THE ÆVAR THE PERSONAL SHOPPER

Many books have been written about OOP and even among experts, there is often disagreement about what OOP is. Many programmers have tried to explain OOP and leave the programmer confused. A case in point is the classic "An object is a data structure with behaviors attached to it." Although that's correct, that's also an awful description and tells you almost nothing you need to know, so instead of giving you a textbook definition, we're going to tell you a story.

You're an awfully busy person and have little free time but plenty of disposable income, so you've decided to hire a personal shopper. His name is Ævar (any resemblance to reviewers of this book, living or dead, is purely coincidental) and he's friendly, flamboyant, and most of all, cheap.

Because Ævar is new to both your city and the job, you have to tell him carefully how much money he can spend, exactly what quality of products you want, and where to buy them. You may even have to tell him which route to drive to pick up the goods and how to invoice you.

That, in essence, is procedural code and that's what you've been doing up to now. You've been carefully telling the computer every step of the way what to do.

After a few months of explaining every little detail, Ævar gets upset and says, "*þegiðu maður, ég veit alveg hvað ég er að gera*" (Icelandic for "Shut up dude; I know what I'm doing"). And he does. He knows what you like and where to get it. He's become an expert. In OO terms, you might now be doing this:

```
my $aevar = Shopper::Personal->new({
    name   => 'Ævar',
    budget => 100
});
$aevar->buy(@list_of_things_to_buy);
my $invoice = $aevar->get_invoice;
```

You're no longer telling Ævar every little step he needs to take to get your shopping done. He's an expert, and he has all the knowledge needed to do your shopping for you and present you with the bill.

And that's all objects are: experts about a problem you need solved. They have all the knowledge you need to get a task done, and you don't *tell* them how to do something, you merely *ask* them to do something.

THREE RULES OF PERL OO

We've already said that Perl has a minimalist OO system. This is both good and bad. It's bad because if you're familiar with OO from another language, you may be frustrated with the differences in Perl or its lack of native facilities to handle things you take for granted. However, it's good because it's easy to learn and extend.

There are three simple rules to know about Perl's OO system.

- A class is a package.
- An object is a reference that knows its class.
- A method is a subroutine.

When you understand and memorize those three rules, you'll know most of what there is to know about basic OO programming in Perl.

Class Is a Package

In OO programming, we often speak of classes. A *class* is a blueprint for something you want to create. Just as you can use a blueprint of a house to make several houses, each painted in different colors, you can use a `Shopper::Personal` class to create several personal shoppers, each with different buying habits. The `Shopper::Personal` class is not the object, but it's the blueprint you can use to create one.

> ***NOTE*** *Given that Perl has been heavily influenced by linguistics, it might also be fair to describe a class as a noun and an instance as a proper noun. It's the difference between the generic idea of a "city" (a noun) and "Paris" (a proper noun).*

> ***NOTE*** *Perl's OO is based on classes. However, this is not the only way to do OO programming. For example, JavaScript uses a prototype-based object system. There's actually some disagreement about many aspects of OO programming, but most of the OO world today (outside of JavaScript, ActionScript, and a few other languages) have settled on class-based OO programming.*

For the `Shopper::Personal` snippet, you can have this:

```
my $aevar = Shopper::Personal->new({
    name   => 'Ævar',
    budget => 100
});
```

You'll note the `Shopper::Personal->new` bit. `Shopper::Personal` is the class name. It looks like a package name because it is! In Perl a class is a package and it's declared the same way. There is no special syntax for declaring a class. In `Shopper/Personal.pm`, declaring the class might start with this:

```
package Shopper::Personal;
use strict;
use warnings;

sub new {
    # more code here
}
```

Pretty simple, eh? Sure, there's more to the code, but a class is nothing special in Perl.

An Object Is a Reference That Knows Its Class

When you create an object, you create a reference that knows what class it belongs to. You can do that by blessing the reference into the class using the `bless` builtin. The syntax looks like this:

```
OBJECT = bless REFERENCE, CLASSNAME;
```

The `bless` builtin tells a reference that it belongs to a class. When the object is used, it knows where its methods are.

When you created your `Shopper::Personal` object and passed in a hash reference:

```
my $aevar = Shopper::Personal->new( {
    name   => 'Ævar',
    budget => 100
} );
```

The code to create it may have looked like this:

```
package Shopper::Personal;
use strict;
use warnings;
sub new {
    my ( $class, $arg_for ) = @_;
    return bless {
        name   => $arg_for->{name},
        budget => $arg_for->{budget},
    }, $class;
}
```

In this code, the `$arg_for` hashref has now been blessed into the `Shopper::Personal` personal class. When you or anyone else uses the object and call methods on it, the blessed reference knows where it is, in this case the `Shopper::Personal` class.

WARNING *Some OO tutorials show you this:*

```
sub new {
    my ( $class, $arg_for ) = @_;
    return bless {
        name   => $arg_for->{name},
        budget => $arg_for->{budget},
    }; # assume current package, bad form
}
```

You blessed the reference but did not say what class to bless it in. When this happens, Perl blesses the object into the current package. This is considered to be bad form because if you later need to inherit from this class (explained later), you may want to reuse the `new()` *constructor, but you can't because it blesses the reference into the current class.*

This is called the one-argument bless *and its use is heavily discouraged.*

For some other languages that allow OO programming, `new` is actually a keyword used to construct objects. In Perl this is not the case. The `new()` method is just another method. You could easily have called the constructor `hire()` and written `Shopper::Personal->hire()`. However, unless you have good reason to do so, the best thing to do is name your constructors `new()` to avoid confusion.

When you see this:

```
my $aevar = Shopper::Personal->new( {
    name   => 'Ævar',
    budget => 100
} );
```

The `Shopper::Personal->new` bit is important. When you use the dereferencing operator, `->`, with a class name on the left and a method name on the right (remember that a method is a subroutine in the class), the method receives the class name as the first argument in `@_` with the other arguments added to `@_` as normal.

NOTE *The first argument to a method is either a class name or an object. Because it's what is responsible for invoking the method, it's referred to as the* invocant.

So the `new()` method, in the preceding example, can have the following arguments:

```
@_ = ( 'Shopper::Personal', { name => 'Ævar', budget => 100 } );
```

So look at the constructor again:

```
sub new {
    my ( $class, $arg_for ) = @_;
    return bless {
        name   => $arg_for->{name},
        budget => $arg_for->{budget},
    }, $class;
}
```

You can see that `$class` contains `Shopper::Personal` and `$arg_for` contains the hash reference.

You don't actually need to pass a hash reference. You could pass a list:

```
my $aevar = Shopper::Personal->new( 'Ævar', 100 );
```

And then the `new()` constructor might look something like this:

```
sub new {
    my ( $class, $name, $budget ) = @_;
    return bless {
        name   => $name,
        budget => $budget,
    }, $class;
}
```

> ***NOTE*** *You can use* `bless` *with any kind of reference. Here, you bless an array reference:*
>
> ```
> sub new {
>     my ( $class, $name, $budget ) = @_;
>     return bless [ $name, $budget ], $class;
> }
>
> # these methods will make more sense in the next section
> sub name {
>     my $self = shift;
>     return $self->[0];
> }
>
> sub budget {
>     my $self = shift;
>     return $self->[1];
> }
> ```
>
> *However, as you get more experience with OO programming, blessing a hash reference is much easier to work with than blessing other types of references, particularly if the class may be subclassed.*

A Method Is a Subroutine

Moving along, you see this:

```
$aevar->buy(@list_of_things_to_buy);
my $invoice = $aevar->get_invoice;
```

Here you call two methods, `buy()` and `get_invoice()` against the `$aevar` object. When this happens, `$aevar` is passed as the first argument in `@_` with the other arguments following. Before looking at those methods, look at the `name` and `budget` attributes passed to the constructor.

```
my $aevar = Shopper::Personal->new( {
    name   => 'Ævar',
    budget => 100
} );

print $aevar->get_name;
print $aevar->get_budget;
```

Now expand the `Shopper::Personal` class just a bit to provide those methods.

```
package Shopper::Personal;

use strict;
use warnings;

sub new {
    my ( $class, $arg_for ) = @_;
    return bless {
        name   => $arg_for->{name},
        budget => $arg_for->{budget},
    }, $class;
}

sub get_name {
    my $self = shift;
    return $self->{name};
}

sub get_budget {
    my $self = shift;
    return $self->{budget};
}

1;
```

> ***NOTE*** *By now, some of you are wondering why the constructor is blessing a hash reference without checking the validity of those arguments:*
>
> ```
> sub new {
>     my ( $class, $arg_for ) = @_;
>     return bless {
>         name   => $arg_for->{name},
>         budget => $arg_for->{budget},
>     }, $class;
> }
> ```
>
> *What if some of the keys are misspelled or the values contain invalid values? The* `new()` *constructor here is actually not good practice, but it has the advantage of being simple enough to not get in the way of explaining the basics of OOP in Perl.*

When you call a method using a class name:

```
my $shopper = Shopper::Personal->new($args);
```

The class name is passed as the first argument to `@_`. Naturally, when you call a method using the instance:

```
my $budget = $shopper->get_budget();
```

The `$shopper` instance gets passed as the first argument to `@_`. Thus, for the `get_budget()` method:

```
sub get_budget {
    my $self = shift @_;
    return $self->{budget};
}
```

You can refer to the object as `$self` (this is by convention, but other popular names are `$this` and `$object`) and because it's the first argument to `get_budget()`, you can shift it off `@_`. Because `$self` is a blessed hash reference it "knows" that it wants the `get_budget()` method from the `Shopper::Personal` class. Therefore, you can fetch the budget attribute with normal dereferencing syntax:

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

> ***WARNING*** *When you read the data in a blessed object by directly accessing the reference, this is called* reaching inside *the object. In general, the only time this should be done is for the getters and setters and even then, only inside the class. Otherwise, use the proper methods to get the data.*
>
> ```
> my $budget = $shopper->budget;      # Right.
> my $budget = $shopper->{budget};    # WRONG, WRONG, WRONG!
> ```
>
> *DO NOT REACH INSIDE THE OBJECT IF YOU DO NOT HAVE TO. I cannot emphasize this strongly enough; even though many developers seem to think it's Okay. The reason is simple: When you use a method call to get the value, you do not know or care how the data is "gotten." The maintainer of the object class is free to change the internals of the object at any time so long as they keep the interface the same. By reaching inside the object, you're relying on behavior that is not and should not be guaranteed. Many a programmer (including your author) has learned this the hard way.*
>
> *Let me repeat that: DO NOT REACH INSIDE THE OBJECT IF YOU DO NOT HAVE TO. It's important.*

By now you can see that if you want to change the budget value, it's fairly trivial:

```
sub set_budget {
    my ( $self, $new_budget ) = @_;
    $self->{budget} = $new_budget;
}
```

In fact, many objects in Perl overload the `budget()` method to be both a setter and a getter (or mutator/accessor, if you prefer big words).

```
sub budget {
    my $self = shift;
    if (@_) { # we have more than one argument
        $self->{budget} = shift;
    }
    return $self->{budget};
}
```

That allows you to do this:

```
my $budget = $aevar->budget;   # get the existing budget
$aevar->budget($new_budget);   # set a new budget
```

Some developers prefer to keep the `get_` and `set_` behaviors separate, such as:

```
my $budget = $aevar->budget;       # get the existing budget
$aevar->set_budget($new_budget);   # set a new budget
```

Others prefer to have the `budget()` method used for both the getter and the setter. It's a matter of personal choice, but whichever style you choose, stick with it to avoid confusing later developers.

One strong recommendation in favor of separate getters and setters is the case in which some getters do not have corresponding setters because that data is read-only:

```
my $customer = Customer->find($customer_id);
print $customer->name;

$customer->name($new_name);
print $customer->id;

$customer->id($new_id);   # boom! this is read-only
```

In this example, the `id()` method of a `Customer` object is assumed to be read-only, but you can't tell this directly from the API methods. However, if you prefixed all setters with `set_` and there was no `set_id()` method, the run-time error `Can't locate object method "set_id" via package "Customer"` is a good clue that you cannot set the ID to a new value. What's worse, the minimalist getters that many developers write can obscure the problem:

```
sub id {
    my ($self) = @_;
    return $self->{id};
}
```

As you can see, if you tried to set a new ID with this method, it would fail, but it would do so silently. This could be hard to debug. Failures should be loud, painful, and clear.

Getting back to `Shopper::Personal`, you have the following code:

```
package Shopper::Personal;

use strict;
```

```
use warnings;

sub new {
    my ( $class, $arg_for ) = @_;
    return bless {
        name   => $arg_for->{name},
        budget => $arg_for->{budget},
    }, $class;
}

sub get_name {
    my $self = shift;
    return $self->{name};
}

sub get_budget {
    my $self = shift;
    return $self->{budget};
}

1;
```

But what does the `buy()` method look like? Well, it might look something like this:

```
sub buy {
    my ( $self, @list_of_things_to_buy ) = @_;

    my $remaining_budget = $self->get_budget;
    my $name             = $self->get_name;

    foreach my $item (@list_of_things_to_buy) {
        my $cost = $self->_find_cost_of($item);

        if ( not defined $cost ) {
            carp("$name doesn't know how to buy '$item'");
        }
        elsif ( $cost > $remaining_budget ) {
            carp("$name doesn't have enough money buy '$item'");
        }
        else {
            $remaining_budget -= $cost;
            $self->_buy_item($item);
        }
    }
}
```

You can see that this method is calling out to other methods, some of which start with an underscore (`_find_cost_of()`, and `_buy_item()`), indicating that they are private methods that should not be used outside of this package.

For each item, you have three possibilities:

- Ævar can't find the item.
- Ævar can't afford the item.
- The item is purchased.

Oh, and you can use the `carp()` subroutine, so don't forget to include the `use Carp 'carp';` line at the top of the code.

When you call a method against a class, such as this:

```
my $aevar = Shopper::Personal->new($hashref);
```

This method is called a class method because it can be safely called with the class name instead of an instance of the class. In this case, the constructor is returning `$aevar`, an instance of the `Shopper::Personal` class. Later, when you call a method against `$aevar`:

```
my $invoice = $aevar->get_invoice;
```

`get_invoice()` is called an instance method because you must have an instance of the object to safely call that method. When you try to call a method and you get an error message like this:

```
Can't use string ("Shopper::Personal") as a HASH ref ...
```

It's probably because you accidentally called an instance method as a class method:

```
Shopper::Personal->buy(@list_of_things_to_buy);
```

When you should have called it on an instance:

```
$aevar->buy(@list_of_things_to_buy);
```

New OO programmers are often confused by this, but think about blueprints again. If you use a blueprint (class) to build several houses (instances), you could see how many bedrooms each house (instance) has by reading the blueprint (class). However, you probably wouldn't know what furniture each house (instance) has.

TRY IT OUT Your First Class

Because you wrote a lot of the code for the `Shopper::Personal` class, finish writing the entire class. All the code in this Try It Out is found in the code file `lib/Shopper/Personal.pm` and `example_12_1_shopper.pl`.

1. Make a directory path called `chapter12/lib/Shopper/`. Change to the `chapter12` directory. Type in the following class and save it as `lib/Shopper/Personal.pm`:

```
package Shopper::Personal;

use strict;
use warnings;
```

```
use Carp qw(croak carp);
use Scalar::Util 'looks_like_number';

our $VERSION = '0.01';

sub new {
    my ( $class, $arg_for ) = @_;
    my $self = bless {}, $class;
    $self->_initialize($arg_for);
    return $self;
}

sub _initialize {
    my ( $self, $arg_for ) = @_;
    my %arg_for = %$arg_for;     # make a shallow copy
    my $class = ref $self;
    $self->{purchased_items} = [];
    $self->{money_spent}     = 0;
    my $name = delete $arg_for{name};
    unless ( defined $name ) {
        croak("$class requires a name to be set");
    }
    $self->set_budget( delete $arg_for{budget} );
    $self->{attributes}{name} = $name;
    if ( my $remaining = join ', ', keys %arg_for ) {
        croak("Unknown keys to $class\::new: $remaining");
    }
}

sub get_name {
    my $self = shift;
    return $self->{attributes}{name};
}

sub set_budget {
    my ( $self, $budget ) = @_;
    unless ( looks_like_number($budget) && $budget > 0 ) {
        croak("Budget must be a number greater than zero");
    }
    $self->{attributes}{budget} = $budget;
}

sub get_budget {
    my $self = shift;
    return $self->{attributes}{budget};
}

sub buy {
    my ( $self, @list_of_things_to_buy ) = @_;
    my $remaining_budget = $self->get_budget;
    my $name             = $self->get_name;
    foreach my $item (@list_of_things_to_buy) {
        my $cost = $self->_find_cost_of($item);
        if ( not defined $cost ) {
```

```
            carp("$name doesn't know how to buy '$item'");
        }
        elsif ( $cost > $remaining_budget ) {
            carp("$name doesn't have enough money buy '$item'");
        }
        else {
            $remaining_budget -= $cost;
            $self->_buy_item($item);
        }
    }
}

sub get_invoice {
    my $self        = shift;
    my @items       = $self->_purchased_items;
    my $money_spent = $self->_money_spent;
    my $shopper     = $self->get_name;
    my $date        = localtime;
    unless (@items) {
        return "No items purchased";
    }
    my $invoice =<<"END_HEADER";
Date:    $date
Shopper: $shopper
Item       Cost
END_HEADER
    foreach my $item (@items) {
        $invoice .= sprintf "%-10s %0.2f\n", $item,
 $self->_find_cost_of($item);
    }
    $invoice .= "\nTotal + 10%: $money_spent\n";
    return $invoice;
}

sub _purchased_items { @{ shift->{purchased_items} } }

sub _money_spent {
    my $self = shift;
    # we assume personal shoppers add 10% to the price
    # to cover the cost of their services
    return $self->{money_spent} * 1.10;
}

sub _find_cost_of {
    my ( $class, $item ) = @_;
    my %price_of = (
        beer    => 1,
        coffee  => 3.5,
        ravioli => 1.5,
        ferrari => 225_000,
    );
    return $price_of{lc $item};
}
```

```
sub _buy_item {
    my ( $self, $item ) = @_;
    $self->{money_spent} += $self->_find_cost_of($item);
    push @{ $self->{purchased_items} }, $item;
}

1;
```

2. In your `chapter12` directory, save the following program as `example_12_1_shopper.pl`:

```
use strict;
use warnings;

use lib 'lib';
use Shopper::Personal;

my $shopper = Shopper::Personal->new({
    name   => 'aevar',
    budget => 10,
});

$shopper->buy(
    'beer',
    'Ferrari',
    ('coffee')  x 2,
    ('ravioli') x 2,
    'beer',
);
print $shopper->get_invoice;

my $next_shopper = Shopper::Personal->new({
    name  => 'bob',
    limit => 10,
});
```

When you finish, your current directory structure should look like this:

```
./
|  lib/
|  |  Shopper/
|  |  |--Personal.pm
|--example_12_1_shopper.pl
```

3. Run the program with `perl example_12_1_shopper.pl`. You should see output similar to the following (obviously your date will be different):

```
aevar doesn't have enough money buy 'Ferrari' at shopper.pl line 11
aevar doesn't have enough money buy 'ravioli' at shopper.pl line 11
aevar doesn't have enough money buy 'beer' at shopper.pl line 11
Date:    Sun Feb 26 16:15:29 2012
Shopper: aevar
Item       Cost
beer       1.00
```

```
coffee     3.50
coffee     3.50
ravioli    1.50
Total + 10%: 10.45
Budget must be a number greater than zero at shopper.pl line 21
```

How It Works

Obviously this Try It Out is far more involved that much of what you've done before. Nothing new was introduced in the Try It Out, but I've reached into my bag of tricks and put together a lot of interesting things here. First, look at object construction. Start numbering lines of code for longer bits like this.

```
 1:  package Shopper::Personal;
 2:  use strict;
 3:  use warnings;
 4:  use Carp qw(croak carp);
 5:  use Scalar::Util 'looks_like_number';
 6:
 7:  our $VERSION = '0.01';
 8:
 9:  sub new {
10:      my ( $class, $arg_for ) = @_;
11:      my $self = bless {}, $class;
12:      $self->_initialize($arg_for);
13:      return $self;
14:  }
15:
16:  sub _initialize {
17:      my ( $self, $arg_for ) = @_;
18:      my %arg_for = %$arg_for;    # make a shallow copy
19:      my $class = ref $self;
20:
21:      $self->{purchased_items} = [];
22:      $self->{money_spent}     = 0;
23:
24:      my $name = delete $arg_for{name};
25:      unless ( defined $name ) {
26:          croak("$class requires a name to be set");
27:      }
28:
29:      $self->set_budget( delete $arg_for{budget} );
30:
31:      $self->{attributes}{name} = $name;
32:
33:      if ( my $remaining = join ', ', keys %arg_for ) {
34:          croak("Unknown keys to $class\::new: $remaining");
35:      }
36:  }
```

Lines 1 through 7 are standard boilerplate, making your code safer and importing `carp`, `croak`, and `looks_like_number`, three utility subroutines that your methods can find useful.

The constructor, `new()` (lines 9–14), now blesses only a hash ref and then immediately passes control to the `_initialize()` method. This allows the constructor to do only one thing. The `_initialize()`

method handles setting up the actual object state and making sure it's sane. In fact, `_initialize()` does three things:

1. Create entries in the hashref for storing important data (lines 21–29).
2. Make sure data supplied in the constructor is valid (lines 25–27).
3. Make sure no extra keys are supplied (lines 33–33).

It's important to look closely at lines 12 and 18:

```
12:      $self->_initialize($arg_for);
18:      my %arg_for = %$arg_for;    # make a shallow copy
```

Notice that you pass the hashref, `$arg_for` and then dereference it in the `%arg_for` variable. By doing this, you can ensure to provide a shallow copy of the hash to `_initialize()`. Otherwise, when you do this:

```
24:     my $name = delete $arg_for{name};
```

If it were a reference, you would have deleted `$arg_for->{name}` and that would have altered the value of the hash reference that's passed to the constructor! You don't want to do that.

So why delete the keys? You don't actually need to, but if you delete all allowed keys, it makes it easy for lines 33–35 to see that there are extra keys left over and `croak()` with a list of said keys. Note that there are many other ways to verify passing the correct arguments. This is merely one of them.

If this seems like a bit of extra work, that's because it is. Many OO authors assume that people will just read the documentation and use the class correctly. Unfortunately, without tight validation of your arguments, it's easy to get this wrong and have unexpected side effects. In Chapter 13, when discussing `Moose`, you'll see how much easier classes like this are to write.

Next, you have three methods for your attributes:

```
38:  sub get_name {
39:      my $self = shift;
40:      return $self->{attributes}{name};
41:  }
42:
43:  sub set_budget {
44:      my ( $self, $budget ) = @_;
45:      unless ( looks_like_number($budget) && $budget > 0 ) {
46:          croak("Budget must be a number greater than zero");
47:      }
48:      $self->{attributes}{budget} = $budget;
49:  }
50:
51:  sub get_budget {
52:      my $self = shift;
53:      return $self->{attributes}{budget};
54:  }
```

The `get_name()` and `get_budget()` methods are straightforward, but the `set_budget()` takes a bit more work because line 45 checks to make sure that the budget is actually a number greater than zero.

Note how the `_initialize()` method takes advantage of the `set_budget()` method on line 29. That makes it easier to avoid duplicating logic.

Next, you have the `buy()` method:

```
56:  sub buy {
57:      my ( $self, @list_of_things_to_buy ) = @_;
58:      my $remaining_budget = $self->get_budget;
59:      my $name             = $self->get_name;
60:
61:      foreach my $item (@list_of_things_to_buy) {
62:          my $cost = $self->_find_cost_of($item);
63:
64:          if ( not defined $cost ) {
65:              carp("$name doesn't know how to buy '$item'");
66:          }
67:          elsif ( $cost > $remaining_budget ) {
68:              carp("$name doesn't have enough money buy '$item'");
69:          }
70:          else {
71:              $remaining_budget -= $cost;
72:              $self->_buy_item($item);
73:          }
74:      }
75:  }
```

This method iterates over the list of items to buy and, so long as you have enough money left in your budget, you buy the item (line 72). You `carp()` if you cannot find the price for the item (line 65) or if you don't have enough money left (line 68).

In particular, pay attention to lines 58 and 59:

```
58:      my $remaining_budget = $self->get_budget;
59:      my $name             = $self->get_name;
```

Because you're inside the `Shopper::Personal` class, why not just grab the data directly instead of the calling these accessors?

```
58:      my $remaining_budget = $self->{attributes}{budget}
59:      my $name             = $self->{attributes}{name};
```

There are several reasons for this:

- Calling the method is often more readable.
- You don't need to worry about misspelling the hash keys.
- If you change the logic of the methods, you don't have to change this code.

The final point is particularly true when you learn about inheritance in the "Subclassing" section later in this chapter.

Looking at `get_invoice()`:

```
77:  sub get_invoice {
78:      my $self        = shift;
```

```
79:      my @items       = $self->_purchased_items;
80:      my $money_spent = $self->_money_spent;
81:      my $shopper     = $self->get_name;
82:      my $date        = localtime;
83:      unless (@items) {
84:          return "No items purchased";
85:      }
86:      my $invoice =<<"END_HEADER";
87:  Date:    $date
88:  Shopper: $shopper
89:
90:  Item       Cost
91:  END_HEADER
92:      foreach my $item (@items) {
93:          $invoice .= sprintf "%-10s %0.2f\n", $item, \
$self->_find_cost_of($item);
94:      }
95:      $invoice .= "\nTotal + 10%: $money_spent\n";
96:      return $invoice;
97:  }
```

You may not be familiar with `localtime`, a Perl builtin used in line 82, but `perldoc -f localtime` can reveal that in scalar context it returns a human-readable form of the current date and time, which is exactly what you need for the header of the report that you generate in lines 86–89.

Lines 92–94 add the individual items to the invoice, and line 95 adds the total. Actually, `get_invoice()` is a normal method.

After `get_invoice()`, however, all methods begin with underscores. These are private methods that only the class should use. There are only a couple of them you might find interesting.

First, the `_purchased_items()` method is rather simple:

```
99:  sub _purchased_items { @{ shift->{purchased_items} } }
```

Why wouldn't you just go ahead and make that public so that anyone can use it to get a list of purchased items? A good rule of thumb is to make nothing in a class public unless absolutely necessary. As soon as you make something a public method, you've now committed your class to maintaining that interface because you don't want to break others' code. By making it a private method, you give yourself flexibility. You can always make a private method public later, but making a public method private is much more likely to break someone's code.

The `get_name()` and `get_budget()` methods should have been private and the `set_budget()` budget method should not have existed at all. Why? Because if you look at the sample code, you see that you need only the `new()`, `buy()` and `get_invoice()` methods. Those are the only three methods that need to be made public, but you made a few others public just to show a bit more about how getters/setters typically work in Perl.

Remember: If you don't need to make a method public, don't.

The other potentially interesting method here is the `_buy_item()` method:

```
120:  sub _buy_item {
121:      my ( $self, $item ) = @_;
122:      $self->{money_spent} += $self->_find_cost_of($item);
123:      push @{ $self->{purchased_items} }, $item;
123:  }
```

What do you do when you buy an item at a store: You pay for it and take it with you. That's exactly what the `_buy_item()` method does and it does nothing else. Just as a class should contain all the logic necessary to be an "expert" on whatever problem domain the class is for — and not do anything else — individual methods should contain all the logic needed to handle their smaller piece of the problem — and not do anything else.

OBJECTS – ANOTHER VIEW

Sometimes objects don't do complicated tasks like buying things. Sometimes they're just there to encapsulate a complex data structure and make sure it has all the needed properties of a class and doesn't allow invalid data to be created.

When your author worked at the BBC, he was one of the developers responsible for handling metadata. *Metadata* is information about information. It seems strange, but it's fairly natural when you get used to it. For example, an episode of a TV show might present a lot of information about animals, but what about the information regarding the episode? For your purposes, TV show episode objects won't model everything you need, but you'll have just enough to show how this works. You can create a small class to model this.

This really isn't different from the "objects as experts" example earlier, but it's a good foundation to show how objects can sometimes be viewed as complex data types.

Using TV::Episode

You'll startout with a basic `TV::Episode` class, making read-only accessors for all your data. You can find the following code in the code file `lib/TV/Episode.pm`:

```
package TV::Episode;

use strict;
use warnings;

use Carp 'croak';
use Scalar::Util 'looks_like_number';

our $VERSION = '0.01';

my %IS_ALLOWED_GENRE = map { $_ => 1 } qw(
  comedy
  drama
  documentary
```

```
  awesome
);

sub new {
    my ( $class, $arg_for ) = @_;
    my $self = bless {} => $class;
    $self->_initialize($arg_for);
    return $self;
}

sub _initialize {
    my ( $self, %arg_for ) = @_;
    my %arg_for = %$arg_for;
    foreach my $property (qw/series director title/) {
        my $value = delete $arg_for{$property};
        # at least one non-space character
        unless ( defined $value && $value =~ /\S/ ) {
            croak("property '$property' must have at a value");
        }
        $self->{$property} = $value;
    }
    my $genre = delete $arg_for{genre};
    unless ( exists $IS_ALLOWED_GENRE{$genre} ) {
        croak("Genre '$genre' is not an allowed genre");
    }
    $self->{genre} = $genre;
    foreach my $property (qw/season episode_number/) {
        my $value = delete $arg_for{$property};
        unless ( looks_like_number($value) && $value > 0 ) {
            croak("$property must have a positive value");
        }
        $self->{$property} = $value;
    }
    if ( my $extra = join ', ' => keys %arg_for ) {
        croak("Unknown keys to new(): $extra");
    }
}

sub series         { shift->{series} }
sub title          { shift->{title} }
sub director       { shift->{director} }
sub genre          { shift->{genre} }
sub season         { shift->{season} }
sub episode_number { shift->{episode_number} }

sub as_string {
    my $self       = shift;
    my @properties = qw(
      series
      title
      director
      genre
      season
      episode_number
```

```
    );
    my $as_string = '';
    foreach my $property (@properties) {
        $as_string .= sprintf "%-14s - %s\n", ucfirst($property),
          $self->$property;
    }
    return $as_string;
}

1;
```

There's nothing terribly unusual about it; though there is a huge amount of tedious validation in the `_initialize()` method. Chapter 13 covers the `Moose` object system and shows you how to make most of this code go away.

One strange bit you'll notice in the `as_string()` method is this:

```
$self->$property;
```

If you have code like this:

```
my $method = 'genre';
print $self->$method;
```

That's equivalent to:

```
$self->genre;
```

Using a variable as a method name is illegal in many other OO languages, but Perl allows this, and it's handy because there are times when you might want to delay the decision about which method to call until runtime. Otherwise, the previous code may have had this:

```
my $format = "%-14s - %s\n";
my $episode = sprintf $format, 'Series',   $self->series;
$episode .=   sprintf $format, 'Title',    $self->title;
$episode .=   sprintf $format, 'Director', $self->director;
$episode .=   sprintf $format, 'Genre',    $self->genre;
$episode .=   sprintf $format, 'Season',   $self->season;
$episode .=   sprintf $format, 'Episode number',
 $self->episode_number;
return $episode;
```

That is error prone and the `foreach` loop makes it simpler.

Moving along, you can use your class like this (code file `listing_12_1_episode.pl`):

```
use strict;
use warnings;

use lib 'lib';
use TV::Episode;
```

```
my $episode = TV::Episode->new({
    series         => 'Firefly',
    director       => 'Marita Grabiak',
    title          => 'Jaynestown',
    genre          => 'awesome',
    season         => 1,
    episode_number => 7,
});
print $episode->as_string;
```

And that prints out:

```
Series         - Firefly
Title          - Jaynestown
Director       - Marita Grabiak
Genre          - awesome
Season         - 1
Episode_number - 7
```

And that's great! Except for one little problem you probably don't know about. When you create objects, you must model your objects to fit real-world needs, and you've never actually seen an episode. In reality, you've seen a broadcast on television or an *ondemand*, a streaming version that you can watch on demand on a website (and you're ignoring that there are different versions of episodes, DVDs and other issues). A broadcast might have a broadcast date and an ondemand might have an availability date range. What you need is more specific examples of your `TV::Episode` class. That's where subclassing comes in.

Subclassing

A *subclass* of a class (also known as a *child class*) is a more specific version of that class. For example, a `Vehicle` class might have `Vehicle::Automobile` and `Vehicle::Airplane` subclasses. The `Vehicle::Airplane` class might in turn have `Vehicle::Airplane::Jet` and `Vehicle::Airplane::Propeller` subclasses.

A subclass uses *inheritance* to provide all the *parent* (also known as a *superclass*) behavior. A method provided by a parent class and used by the subclass is called an *inherited method*. For example, if class `A` provides a `foo()` method and class `B` inherits from `A`, class `B` will also have the `foo()` method, even if it does not implement one itself. (If `B` does have a `foo()` method, this is called *overriding* the inherited method.)

> ***NOTE*** *From here on out, I'll use* parent *and* superclass, child *and* subclass *interchangeably. This is because they mean the same thing in Perl and the literature on the subject uses both. Thus, I want you to be very familiar with both terms.*

For the `TV::Episode` class, you need a `TV::Episode::Broadcast` subclass and a `TV::Episode::OnDemand` subclass.

THE LISKOV SUBSTITUION PRINCIPLE

I mentioned that subclasses should extend the behavior of their parent classes but not alter this behavior. This is due to something known as the *Liskov Substitution Principle*. This principle effectively states the same thing. The purpose of this principle is to ensure that in any place in your program you can use a given class; if you actually use a subclass of that class, your program should still function correctly.

It has a few more subtleties than merely not changing parent behavior. For example, subclasses are allowed to be less restrictive in the data they accept but not in the data they emit. There is some controversy over the Liskov Substitution Principle, but it's a good idea to follow unless you have strong reasons not to.

The principle was created by Barbara Liskov, Ph.D. She won the 2008 Turing Award (the Nobel prize for computer science) for her work in computer science and her work has influenced much of computing today.

See `http://en.wikipedia.org/wiki/Liskov_substitution_principle` and `http://en.wikipedia.org/wiki/Barbara_Jane_Liskov` for more information. Just remember that if you cannot use a subclass in the same place where you can use a parent class, you might have a design flaw.

Using TV::Episode::Broadcast

When something like the `TV::Episode::Broadcast` class uses the `TV::Episode` class as its parent, you can say that `TV::Episode::Broadcast` *inherits* from `TV::Episode`. To represent the broadcast date, use the `DateTime` module you can download from the CPAN. Here's how `TV::Episode::Broadcast` class works. You can find the following code in code file `lib/TV/Episode/Broadcast.pm`.

```
package TV::Episode::Broadcast;

use strict;
use warnings;

use Try::Tiny;
use Carp 'croak';
use base 'TV::Episode';  # inherit!

sub _initialize {
    my ( $self, $arg_for ) = @_;
    my %arg_for = %$arg_for;
    my $broadcast_date = delete $arg_for{broadcast_date};
    try {
        $broadcast_date->isa('DateTime') or die;
    }
    catch {
        croak("broadcast_date must be a DateTime object");
    };
```

```
    $self->{broadcast_date} = $broadcast_date;
    $self->SUPER::_initialize( \%arg_for );
}

sub broadcast_date { shift->{broadcast_date} }

sub as_string {
    my $self    = shift;
    my $episode = $self->SUPER::as_string;
    my $date    = $self->broadcast_date;
    $episode .= sprintf "%-14s - %4d-%2d-%2d\n"
=> 'Broadcast date',
$date->year,
$date->month,
$date->day;
    return $episode;
}

1;
```

And this looks similar to `TV::Episode`, but now you supply the broadcast date:

```
my $broadcast = TV::Episode::Broadcast->new(
    {
        series         => 'Firefly',
        director       => 'Allan Kroeker',
        title          => 'Ariel',
        genre          => 'awesome',
        season         => 1,
        episode_number => 9,
        broadcast_date => DateTime->new(
            year  => 2002,
            month => 11,
            day   => 15,
        ),
    }
);
print $broadcast->as_string;
print $broadcast->series;
```

Running the program prints out:

```
Series         - Firefly
Title          - Ariel
Director       - Allan Kroeker
Genre          - awesome
Season         - 1
Episode_number - 9
Broadcast date - 2002-11-15
Firefly
```

Because `TV::Episode::Broadcast` has inherited from `TV::Episode`, broadcasts have all the behavior of episodes, so you can still call `$broadcast->series`, `$broadcast->director`, and so on.

There's no need to re-implement these behaviors. This is because when you call a method on an object, Perl checks to see if that method is defined in the object's class. If it's not, it searches the parent class, and then the parent's parent class, and so on, until it finds an appropriate method to call, or dies, telling you that the method is not found.

This is why `TV::Episode::Broadcast` does not have a `new()` method. When you try to call `TV::Episode::Broadcast->new(...)`, Perl looks for `TV::Episode::Broadcast::new()` and, not finding it, starts searching the superclasses and calls the first `new()` method it finds (`TV::Episode::new()` in this case). This is one of the reasons why OO is so powerful: It makes it easy to reuse code.

Perl knows that `TV::Episode` is the parent of `TV::Episode::Broadcast` because of this line:

```
use base 'TV::Episode';
```

The `base` module is commonly used to establish inheritance. There's a newer version named `parent` that does the same thing:

```
use parent 'TV::Episode';
```

It's a fork of the `base` module and mostly involves cleaning up some of the internal cruft that `base` has accumulated over the years. It's not entirely compatible with it, but you'll likely not notice the difference.

> ***NOTE*** *The* `base` *and* `parent` *modules also take lists allowing you to inherit from multiple modules at once:*
>
> ```
> TV::Episode::AllInOne;
> use base qw(
>   TV::Episode::Broadcast
>   TV::Episode::OnDemand
> );
> ```
>
> *This is referred to as* multiple inheritance. *It's usually a bad idea and its use is controversial enough that many programming languages forbid it outright. Chapter 13 talks about multiple inheritance when discussing roles.*

For some older Perl modules, you see inheritance established with the `@ISA` array:

```
package TV::Episode::Broadcast;

# with @ISA, you must first 'use' the modules
# you wish to inherit from

use TV::Episode;
use vars '@ISA';
@ISA = 'TV::Episode';
# optionally: our @ISA = 'TV::Episode';
```

When Perl tries to figure out a module's parent or parents, it looks at the module's `@ISA` package variable and any classes contained therein are considered parents. Although this method to establish inheritance is now discouraged, you can still sometimes see code messing with the `@ISA` array, so it's important to remember it. The base and parent modules are merely loading the parents and assigning to `@ISA` for you. They make it harder to forget to `use` the parent modules and also protect from *circular inheritance*, a problem whereby a class accidentally inherits from itself.

Now look at your new `_initialize()` method. This overrides the `_initialize()` method from the parent class. Because it overrides, the `TV::Episode::_initialize()` method will not be called unless you call it explicitly, as you do in line 13:

```
 1:  sub _initialize {
 2:      my ( $self, $arg_for ) = @_;
 3:      my %arg_for = %$arg_for;
 4:      my $broadcast_date = delete $arg_for{broadcast_date};
 5:
 6:      try {
 7:          $broadcast_date->isa('DateTime') or die;
 8:      }
 9:      catch {
10:          croak("Not a DataTime object: $broadcast_date");
11:      };
12:      $self->{broadcast_date} = $broadcast_date;
13:      $self->SUPER::_initialize(\%arg_for);
14:  }
```

The `$self->SUPER::_initialize()` syntax is what you use to call the superclass method. If it doesn't exist, you'll get an error like:

```
Can't locate object method "_initialize" via package "main::SUPER"
```

This allows you to override a parent method but still rely on its behavior if you need to. In this case, you supply an extra parameter but remove it from the `%arg_for` hash to ensure that the parent `_initialize()` method does `croak()` when it sees the extra argument. You can test that the parameter is suitable with a `try/catch` block and an `isa()` test, but this is explained a bit more when I cover the `UNIVERSAL` package in the "Using UNIVERSAL" section of this chapter.

> ***NOTE*** *Although the example code shows an overridden method calling their parent versions with* `$self->SUPER::some_method`*, there is actually no requirement that you call the parent method. Use this technique here to show how you can supplement parent method behavior, but replace it entirely with an overridden method, which is fine so long as you don't change the semantics of the method. (Well, you could have your* `as_string()` *method do something radically different from the parent method, such as return an array reference, but that's not a good idea.)*

You can do the same thing on line 3 with the `as_string()` method:

```
 1:  sub as_string {
 2:      my $self    = shift;
 3:      my $episode = $self->SUPER::as_string;
 4:      my $date    = $self->broadcast_date;
 5:      $episode .= sprintf "%-14s - %4d-%2d-%2d\n" => 'Broadcast date',
 6:        $date->year,
 7:        $date->month,
 8:        $date->day;
 9:      return $episode;
10:  }
```

In this case, you use the parent's `as_string()` method to create the text representation of the object and then add an extra line of data. You probably should have pulled the format out into its own method so that you could override the format if needed. You could have done something like this:

```
sub _as_string_format { return "%-14s - %4d-%2d-%2d\n" }
sub as_string {
    my $self    = shift;
    my $episode = $self->SUPER::as_string;
    my $date    = $self->broadcast_date;
    $episode .= sprintf $self->_as_string_format => 'Broadcast date',
      $date->year,
      $date->month,
      $date->day;
    return $episode;
}
```

But that would have required a change to the base class to support the same `_as_string_format()` and you may have not had access to change the base class. If that's the case and you needed a different format, you would have to override the parent `as_string()` method and duplicated most of its logic and not call `$self->SUPER::as_string`.

Class Versus Instance Data

Sometimes you want to share data across all instances of a class, for example:

```
package Universe;
sub new {
    my ( $class, $name ) = @_;
    return bless { name => $name }, $class;
}
sub name { shift->{name} }
sub pi   { 3.14159265359 }
1;
```

That creates a read-only `pi()` method that you can access via `Universe->pi`. You can also call it on an instance and it behaves the same way:

```
my $universe1 = Universe->new('first universe name');
print $universe1->pi, "\n";

my $universe2 = Universe->new('second universe name');
print $universe2->pi, "\n";
```

Each `Universe` you create will have a different name, but share the same value of `pi()`.

You can also make it read-write:

```
package Universe;

sub new {
    my ( $class, $name ) = @_;
    return bless { name => $name }, $class;
}

sub name { shift->{name} }

{
    my $pi = 3.14159265359;
    sub pi {
        my $class = shift;
        if ( @_ ) {
            $pi = shift;
        }
        return $pi;
    }
}
1;
```

However, be aware that this is little more than a global variable. If you change it for one universe, you will change it for all of them. (And you didn't even have data validation for it!)

There is, as you probably suspect by now, a CPAN module to make this easier: `Class::Data::Inheritable`. This allows you to easily define class data but override it in a subclass, if needed:

```
package Universe;
use parent 'Class::Data::Inheritable';
__PACKAGE__->mk_classdata( pi => 3.14159265359 );
```

With that, you can now call `Universe->pi` and get the right answer. Of course, you can still change it:

```
Universe->pi(3);   # oops
```

A better strategy, instead of allowing this hidden global into your code, is sometimes to provide a default:

```
sub new {
    my ( $class, $arg_for ) = @_;
    $arg_for->{pi} ||= $class->_default_pi;
    my $self = bless {}, $arg_for;
    $self->_initialize($arg_for);
    return $self;
}

# You can override this in a subclass, if desired
sub _default_pi { 3.14159265359 }
```

With that, all instances of a class default to a valid value of `pi`, but if you change it later for one class, it does not impact other instances. Whether this is appropriate depends on your needs. Sometimes it's easier to share data across instances.

A BRIEF RECAP

You've covered the basics of OO, so now, have a brief recap of what you've learned so far.

First, there are three rules to Perl's Object-Oriented programming:

- A class is a package.
- An object is a reference blessed into a class.
- A method is a subroutine.

Classes can inherit from other classes to provide more specific types of a class. A class that inherits from another class is called a subclass or child class, and the class it inherits from is the superclass or parent class.

Methods are inherited from parent classes, but the child class can override the methods to provide more specific behavior, including calling back to the parent class methods if need be. The child class can also provide additional attributes or methods as needed.

And that's it for basic OOP in Perl. There's nothing complicated about it, and you can get most of the basics down in a couple of hours. Now, however, it's time to move along and explain a few more things about classes that you should know about.

Overloading Objects

When you have normal variables such as scalars, it's easy to print them, compare them, add or concatenate them, and so on. You can do this with objects, too, by overloading them. You use the `overload` pragma to do this. You're going to create a `TV::Episode::OnDemand` subclass to show how this works. You can skip (some) of the data validation to focus on the actual overloaded behavior. You also take advantage of assuming that your new attributes use `DateTime` objects. `DateTime` is also overloaded and you can see how several overloaded objects can work together to make life easier. We're not going to explain in-depth how overloading works (but see `perldoc overload`) because most objects don't actually use overloading, but you should be familiar with this technique when you come across it and want to use it later.

An ondemand is industry shorthand for *Video On Demand* (*VOD*) and refers to technology allowing you to watch the video when you want (in other words, "on demand"), such as when you watch something on Hulu, YouTube, or the BBC's iPlayer service. Rather than having a broadcast date, an ondemand has availability. In loose terms, this means "when you can watch it." Now you'll create a subclass of `TV::Episode` named `TV::Episode::OnDemand` and it will have `start_date` and `end_date` attributes along with an `available_days` method. The following code uses the code file `lib/TV/Episode/OnDemand.pm` and `listing_12_1_episode.pl`:

```
package TV::Episode::OnDemand;

use strict;
use warnings;
use Carp 'croak';

use overload '""' => 'as_string';

use base 'TV::Episode';

sub _initialize {
    my ( $self, $arg_for ) = @_;
    my %arg_for = %$arg_for;

    # assume these are DateTime objects
    $self->{start_date} = delete $arg_for{start_date};
    $self->{end_date}   = delete $arg_for{end_date};

    # note the > comparison of objects
    if ( $self->start_date >= $self->end_date ) {
        croak("Start date must be before end date");
    }
    $self->SUPER::_initialize( \%arg_for );
}

sub start_date { shift->{start_date} }
sub end_date   { shift->{end_date} }

sub as_string {
    my $self       = shift;
    my $episode    = $self->SUPER::as_string;
    my $start_date = $self->start_date;
    my $end_date   = $self->end_date;

    # overloaded stringification
    $episode .= sprintf "%-14s - $start_date\n" => 'Start date';
    $episode .= sprintf "%-14s - $end_date\n"   => 'End date';
    $episode .= sprintf "%-14s - %d\n"          => 'Available days',
      $self->available_days;
    return $episode;
}

sub available_days {
    my $self     = shift;
    # hey, we can even subtract DateTime objects
    my $duration = $self->end_date - $self->start_date;
    return $duration->delta_days;
}
1;
```

And the script to show how this works:

```
use strict;
use warnings;
use DateTime;
```

```
use lib 'lib';
use TV::Episode::OnDemand;

my $ondemand = TV::Episode::OnDemand->new(
    {
        series         => 'Firefly',
        director       => 'Allan Kroeker',
        title          => 'Ariel',
        genre          => 'awesome',
        season         => 1,
        episode_number => 9,
        start_date => DateTime->new(
            year  => 2002,
            month => 11,
            day   => 21,
        ),
        end_date => DateTime->new(
            year  => 2002,
            month => 12,
            day   => 12,
        ),
    }
);
print $ondemand;
```

Running the script should produce output similar to the following:

```
Series         - Firefly
Title          - Ariel
Director       - Allan Kroeker
Genre          - awesome
Season         - 1
Episode_number - 9
Start date     - 2002-11-21T00:00:00
End date       - 2002-12-12T00:00:00
Available days - 21
```

Note that this code prints `$ondemand` and not `$ondemand->as_string`. What allows you to do that is this line:

```
use overload '""' => 'as_string';
```

The `'""'` argument says "we want to overload this object's behavior when it is used as a string" and the `"as_string"` is the name of the method you will use to handle this behavior. Without this, the print `$ondemand` line would produce something useless like this:

```
TV::Episode::OnDemand=HASH(0x7f908282c9a0)
```

The `DateTime` objects have even more overloading. You can compare dates in your `_initialize()` method:

```
if ( $self->start_date >= $self->end_date ) {
    croak("Start date must be before end date");
}
```

If overloading was not provided, you would either have to do something like this (assuming that `DateTime` offered the appropriate method):

```
if ( $self->start_date->is_greater_than_or_equal_to($self->end_date) ) {
    ...
}
```

Or worse, try to figure out the date math yourself. (And that's harder than it sounds.)

> ***NOTE*** *The* `TV::Episode`*,* `TV::Episode::Broadcast` *and* `TV::Episode::OnDemand` *classes all provide private* `_initialize()` *methods and public* `as_string()` *methods. When you call the* `as_string()` *method on an* `$episode`*,* `$broadcast`*, or* `$ondemand`*, Perl calls the correct* `as_string()` *method for you. This behavior is known as* subtype polymorphism*; though most people just call it* polymorphism*. It allows you to have a uniform interface for related objects of different types.*

The `DateTime` objects also have stringification overloaded, allowing you to do this:

```
$episode .= sprintf "%-14s - $start_date\n" => 'Start date';
$episode .= sprintf "%-14s - $end_date\n"   => 'End date';
```

Otherwise, we would have to fall back to this:

```
$episode .= sprintf "%-14s - %4d-%2d-%2d\n" => 'Broadcast date',
    $date->year,
    $date->month,
    $date->day;
```

> ***NOTE*** *The* `DateTime` *format wasn't pretty when you printed the* `DateTime` *objects directly. Read "Formatters And Stringification" in* `perldoc DateTime` *for fine-grained control of the print format.*

You can also overload subtraction. When you subtract one `DateTime` object from another, it returns a `DateTime::Duration` object:

```
sub available_days {
    my $self     = shift;
    my $duration = $self->end_date - $self->start_date;
    return $duration->delta_days;
}
```

If you've realized how annoying it can be to figure out if one date is greater than another (think about time zones and daylight savings time, among other things), then you can imagine how painful calculating the actual distance between two dates can be. A well-designed module coupled with intelligently overloaded behavior makes this simple.

Using UNIVERSAL

All objects ultimately inherit from the `UNIVERSAL` class. The `TV::Episode` inherits directly from `UNIVERSAL` and `TV::Episode::Broadcast` and `TV::Episode::OnDemand` inherit from `TV::Episode`, meaning that they both inherit directly `UNIVERSAL` through `TV::Episode`. The object graph looks like Figure 12-1.

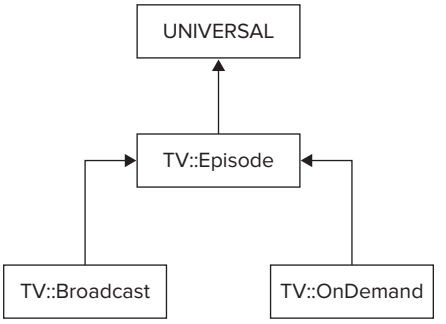

FIGURE 12-1

The `UNIVERSAL` class provides three extremely useful methods that all classes will inherit: `isa()`, `can()`, and `VERSION()`. As of 5.10.1 and better, there is also a `DOES()` method provided, but we won't cover that until we explain roles in Chapter 13.

Understanding the isa() Method

The `isa()` method tells you whether your object or class inherits from another class. It looks like this:

```
$object_or_class->isa(CLASS);
```

Where `$object_or_class` is the object (or class) you want to test and `CLASS` is the class you're comparing against. It returns true if `$object_or_class` matches `CLASS` or inherits from it. The following will all return true:

```
$broadcast->isa('TV::Episode::Broadcast');
$broadcast->isa('TV::Episode');
TV::Episode::OnDemand->isa('TV::Episode');
$ondemand->isa('UNIVERSAL');
$episode->isa('UNIVERSAL');
```

In fact, every object will respond true if you test it against `UNIVERSAL`.

Naturally, all the following return false:

```
$broadcast->isa('TV::Episode::OnDemand');
$episode->isa('TV::Episode::OnDemand');
UNIVERSAL->isa('TV::Episode');
```

You may recall that the `TV::Episode::Broadcast::_initialize()` method had the following bit of code to check to see if you had a valid broadcast date:

```
try {
    $broadcast_date->isa('DateTime') or die;
}
```

```
catch {
croak("Not a DateTime object: $broadcast_date");
};
```

You could have written it like this:

```
if ( not $broadcast_date->isa('DateTime') ) {
    croak("broadcast_date must be a DateTime");
}
```

However, what if someone passed something strange for the `broadcast_date` parameter, or passed nothing at all? The `$broadcast_date->isa()` check would be called against something that might not be an object, and you could get a strange error message. Trapping the error with a `try/catch` block allows you to ensure the user gets exactly the error message you want them to get.

Please note that sometimes you'll see the following mistake:

```
if ( UNIVERSAL::isa($broadcast_date, 'DateTime') ) {
   # BAD IDEA!
}
```

The idea behind this is simple: Because the first argument to a method call is the invocant, calling a method like a subroutine and passing the invocant manually is the same thing. Plus, you don't have to do that annoying `try/catch` stuff or check to see if the invocant is actually an object.

It's a bad idea, though. Sometimes classes override `isa()`, and if you call `UNIVERSAL::isa()` instead of `$object->isa()`, you won't get the class's overridden version, thus leading to a possible source of bugs. Most of the time `UNIVERSAL::isa()` will work just fine, but the one time it doesn't can lead to hard-to-find bugs.

Understanding the can() Method

The `can()` method tells you whether a given object or class implements or inherits a given method. It looks like this:

```
$object_or_class->can($method_name);
```

Because `TV::Episode::Broadcast` and `TV::Episode::OnDemand` both inherit from `TV::Episode`, they will respond to true the following:

```
$episode->can('episode_number');
$broadcast->can('episode_number');
$broadcast->can('episode_number');
```

However, because `TV::Episode` does not implement the `broadcast_date()` method, `$episode->can('broadcast_date')` will return false.

> ***NOTE*** *In reality, the* `can()` *method returns a reference to the method that would be invoked. Some programmers use this to avoid having Perl look up the method twice:*
>
> ```
> if ( my $method = $object->can($method_name) ) {
>     $object->$method;
> }
> ```
>
> *Because a subroutine reference can evaluate to true, that's the same as:*
>
> ```
> if ( $object->can($method_name) ) {
>     $object->$method_name;
> }
> ```
>
> *And yes, objects can call a method that is in a variable name, as shown here. Use this with care to make sure you're not calling a method you don't want to call. Your author has seen many bugs and security holes in Perl code that allows someone to pass in the name of the method to be called.*

Just like that, you'll sometimes see:

```
if ( UNIVERSAL::can( $object, $method_name ) ) {
    # BAD IDEA!
}
```

Again, this is a bad idea because if one of your objects provides its own `can()` method (and this is even more common than providing a new `isa()` method), then the above code is broken. Use the proper OO behavior: `$object->can($method)`.

Understanding the VERSION() Method

The `UNIVERSAL` class also provide a `VERSION()` method. (Why it's in ALL CAPS when the `is()` and `can()` are not is merely one of life's little mysteries.) This returns the version of the object. You'll notice that your code often has things like:

```
our $VERSION = '3.14';
```

That `$VERSION` is precisely what `$object->VERSION` returns, but in a clean interface. As you defined the version as being `'0.01'` for all of our `TV::` classes, calling `->VERSION` on any of them will return `'0.01'`.

Understanding Private Methods

This chapter already mentioned that private methods traditionally begin with an underscore. This bears a bit of explaining. In Perl, all methods are actually public. There is nothing to stop someone from calling your "private" methods. Most good programmers know better than to call these methods, but sometimes they get sloppy or they need behavior from the class that was not made "public."

This also means that subclasses inherit your "private" methods, effectively making them what some other languages would call a protected method. This is a method that is inherited but should not be called outside the class. Generally this is not a problem, but look at the following code:

```
package Customer;

sub new {
    my ( $class, $args ) = @_;
    return bless $args, $class;
}

sub outstanding_balance {
    my $self = shift;
    my @accounts = $self->_accounts;
    my $total = 0;
    $total += $_->total foreach @accounts;
    return $total;
}

sub _accounts {
    my $self = shift;
    # lots of code
    return @accounts;
}

# more code here
```

Now imagine a `Customer::Preferred` class that inherits from `Customer` but implements its own `_accounts()` method that returns an array reference of the customer accounts. If the `outstanding_balance()` method is not overridden, you'll have a run-time error when `outstanding_balance()` expects a list instead of an array reference.

In reality, this problem doesn't happen a lot. Part of the reason is simply because programmers who want to subclass your code often read it and make sure they're not breaking anything, or they write careful tests to verify that they haven't broken anything. However, as your systems get larger, you're more likely to accidentally override methods, and you should consider yourself lucky if it causes a fatal error. It's also possible to cause a subtle run-time error that generates bad data rather than killing your program. When you try to debug a problem in a system with a few hundred thousand lines of code, this type of error can be maddening.

If you are concerned about this, there are a couple of ways to deal with this. One is to simply document your "private" methods and whether they're appropriate to subclass. Another strategy is to declare private methods as subroutine references assigned to scalars:

```
package Customer;

sub new {
    my ( $class, $args ) = @_;
    return bless $args, $class;
}

my $_accounts = sub {
```

```
    my $self = shift;
    # lots of code
    return @accounts;
};

sub outstanding_balance {
    my $self = shift;
    my @accounts = $self->$_accounts;
    my $total = 0;
    $total += $_->total foreach @accounts;
    return $total;
}
```

Here, you assign a code reference to the `$_accounts` variable and later call it with `$self->$_accounts`. You can even pass arguments as normal:

```
my @accounts = $self->$_accounts(@arguments);
```

Note that this technique creates truly private methods that cannot be accidentally overridden. (Actually, you can change them from outside the class, but it's an advanced technique that requires advanced knowledge of Perl.) Most Perl programmers do not actually use this technique and expect people who subclass their modules to test that they haven't broken anything.

> ***NOTE*** *For what it's worth, your author did an informal poll of Perl developers and all of them denied that they have ever worked on code where someone has accidentally overridden a private method. This leaves me in the awkward position of recommending a solution to a problem that no one seems to have experienced.*

TRY IT OUT Creating Episode Versions

Earlier this chapter pointed out that you don't watch episodes; you usually watch a broadcast of an episode or an ondemand of an episode. That was actually a bit of a lie. You watch a broadcast of a version of an episode, or an ondemand of a version of an episode. It might be the original version, edited for adult content (such as naughty words bleeped out), edited for legal reasons (accidentally defaming someone, for example), or any number of reasons. So you're going to create a version subclass of episodes and alter your ondemands and broadcasts to inherit from that instead. All the code in this Try It Out is available in the code file `lib/TV/Episode/Version.pm`.

1. Type in the following program and save it as `lib/TV/Episode/Version.pm`:

```
package TV::Episode::Version;

use strict;
use warnings;
use base 'TV::Episode';

our $VERSION = '0.01';
```

```
sub new {
    my ( $class, $arg_for ) = @_;
    my $self = bless {} => $class;
    $self->_initialize($arg_for);
    return $self;
}

sub _initialize {
    my ( $self, $arg_for ) = @_;
    my %arg_for = %$arg_for;
    $self->{description} = exists $arg_for{description}
       ? delete $arg_for{description}
       : 'Original';
    $self->SUPER::_initialize( \%arg_for );
}

sub description { shift->{description} }

sub as_string {
    my $self      = shift;
    my $as_string = $self->SUPER::as_string;
    $as_string .= sprintf "%-14s - %s\n" => 'Version',
      $self->description;
    return $as_string;
}

1;
```

2. At this point, you have a decision to make. Many developers prefer to have the class structure reflected in the name of the class, meaning the `TV::Episode::Broadcast` and `TV::Episode::OnDemand` would become `TV::Episode::Version::Broadcast` and `TV::Episode::Version::OnDemand`. Each part of the class name shows how you're getting more and more specific. But what if your code is used in other projects that you don't have control over? Instead, you'll decide to keep their class names, and for broadcasts and on demands, you'll merely change their inheritance line to:

```
package TV::Episode::Broadcast;
# snip
use base 'TV::Episode::Version';
```

This may not be the best name for the broadcast or ondemand classes, but it's the sort of compromises you make in real-world code.

Another choice (which, for the sake of simplicity, you're not taking) is to create the new classes like this:

```
package TV::Episode::Version::Broadcast;
use base 'TV::Episode::Broadcast::_initialize;
1;
```

That allows people to use either name, but there's one more change to make:

```
package TV::Episode::Broadcast;

use Carp 'cluck';

sub new {
    my ( $class, $arg_for ) = @_;
    if ( $class eq __PACKAGE__ ) {
       cluck(<<"END");
Package TV::Episode::Broadcast is deprecated. Please use
TV::Episode::Version::Broadcast instead.
END
    }
    my $self = bless {} => $class;
    $self->_initialize($arg_for);
    return $self;
}
```

By adding such a deprecation warning (and documenting this in your POD!), you can give other programmers advance warning of the package name change. This allows their code to continue working and gives them time to make the needed updates to their code.

3. After you update `TV::Episode::OnDemand` to inherit from `TV::Episode::Version`, write the following and save it as `example_12_2_episode.pl`:

```
use strict;
use warnings;
use DateTime;

use lib 'lib';
use TV::Episode::OnDemand;

my $ondemand = TV::Episode::OnDemand->new(
    {
        series         => 'Firefly',
        director       => 'Allan Kroeker',
        title          => 'Ariel',
        genre          => 'awesome',
        season         => 1,
        episode_number => 9,
        start_date => DateTime->new(
            year  => 2002,
            month => 11,
            day   => 21,
        ),
        end_date => DateTime->new(
            year  => 2002,
            month => 12,
            day   => 12,
        ),
    }
);
print $ondemand;
```

4. Run the program with `perl example_12_2_episode.pl`. You should see the following output:

```
Series         - Firefly
Title          - Ariel
Director       - Allan Kroeker
Genre          - awesome
Season         - 1
Episode_number - 9
Version        - Original
Start date     - 2002-11-21T00:00:00
End date       - 2002-12-12T00:00:00
Available days - 21
```

How It Works

By this time you should have an idea of how subclassing works, and there is nothing new here, but now look at a couple of interesting bits, starting with the `_initialize()` method:

```
1:  sub _initialize {
2:      my ( $self, $arg_for ) = @_;
3:      my %arg_for = %$arg_for;
4:
5:      $self->{description} = exists  $arg_for{description}
6:         ? delete $arg_for{description}
7:         : 'Original';
8:      $self->SUPER::_initialize( \%arg_for );
9:  }
```

Instead of calling `croak()` when you don't have a `description`, note how lines 5 through 7, assign the value `Original` to it. This allows you to create a new version and, if this value is not present, assume that it's the original version. However, it has a more important benefit. If other developers are already using the `TV::Episode::Broadcast` and `TV::Episode::OnDemand` classes, they are not setting the description property. If you simply called `croak()` here, you'd break everyone's code and they'd probably be upset with you.

Also, note the`as_string()` method:

```
1:  sub as_string {
2:      my $self      = shift;
3:      my $as_string = $self->SUPER::as_string;
4:      $as_string .= sprintf "%-14s - %s\n" => 'Version',
5:        $self->description;
6:      return $as_string;
7:  }
```

You have again duplicated the `"%-14s - %s\n"` format, so it's probably a good time to abstract this out into a method in your `TV::Episode` base class. If you want to change how this behavior formats in the future, it will be easier to do so.

GOTCHAS

When writing object-oriented code, there are a number of problem areas you should be aware of. We'll only cover a few, but these are important issues that can make your code harder to use or more likely to break.

Unnecessary Methods

Often when people write objects, they correctly think of them as "experts." However, they then rationalize that the object must do everything conceivable that someone wants, rather than simply provide an intended behavior. Rule: Don't provide behavior unless you know that people need it. A good example is people making all object attributes read-write. For example, with `TV::Episode`, say that you want to make the episode number optional and people can set it later if they want to:

```
use Scalar::Util 'looks_like_number';
sub episode_number {
    my $self = shift;
    if (@_) {
        my $number = shift;
        unless ( looks_like_number($number) and $number > 0 ) {
 croak("episode_number is not a positive integer: $number");
        }
        $self->{episode_number} = $number;
    }
    return $self->{episode_number};
}
```

That looks harmless enough, right?

Later on you create a `TV::Season` object and it looks like this:

```
my $season = TV::Season->new({
    season_number => 3,
    episodes      => \@episodes,
});
```

If you assume that all `TV::Episode` objects in `@episodes` must have unique number, you can easily validate this when you construct the `TV::Season` object. However, if you later do this to one of the objects passed to `TV::Season`:

```
$episode->episode_number(3);
```

If another one of the episodes already has an `episode_number` of `3`, you may have two episodes in a season with the same `episode_number`! That's because objects are merely blessed references. Change the data contained in a reference, and the place you store that reference will be pointing to the same data. Errors like this are much harder to avoid if you allow attributes to be set after you've constructed the object. Think carefully if this is a design requirement.

"Reaching Inside"

If you know something is an attribute, it can be tempting to do this:

```
my $name = $shopper->{name};
```

That seems OK because you know that name is in that hash slot, and hey, dereferencing the hash is faster than calling an object method!

And it's stupid, too. The reason that OO developers provide methods to let you get that data is because they must be free to change how the objects work internally, even if you don't see the change on the outside. You want to use `$shopper->name` because although it may be defined like this:

```
sub name { $_[0]->{name} }
```

The next release of the software might define it like this:

```
sub name {
    my $self = shift;
    return join ' ' => $self->first_name, $self->last_name;
}
```

Even inside the class you should avoid reaching inside of the object. You might say, "But I know that `$self->{name}`" is okay — until someone subclasses your module and the `name()` method is completely redefined. Or you are moaning over a nasty bug, not realizing that `$self->{naem}` is embedded somewhere in your code.

Finally, the object method that sets a value might validate that the value is valid. Reaching inside the object completely skips this validation.

Multiple Inheritance

Tighten up your seat belts. This is going to get a little rough, and it's worth reading through a couple of times to understand what's going on.

Multiple inheritance is inheriting from more than one class at the same time. For example, imagine you're writing a game and you want to create a talking box. Because your `Creature` class can speak and your `Box` class is a box, you decide that you want to inherit from both of them rather than rewrite the behaviors:

```
package Creature;
use base 'Physical::Object';
sub speak { ... }
package Box
use base 'Physical::Object';
sub put_inside { ... }
sub take_out { ... }
package Box::Talking;
use base qw(Creature Box);
```

And now your `Box::Talking` can respond to the `speak()`, `put_inside()`, and `take_out()` methods.

On the surface, this looks okay, but multiple inheritance is so problematic that many programming languages ban it outright. What are the constructors going to look like? Do you call both of your parent constructors? What if they do conflicting things?

Imagine what happens if the classes `Box` and `Physical::Object` both have a `weight()` method. When you want to find out its weight you might do this:

```
my $weight = $talking_box->weight;
```

However, Perl, by default, uses a left-most, depth-first inheritance search strategy. Now look at the inheritance hierarchy in Figure 12-2.

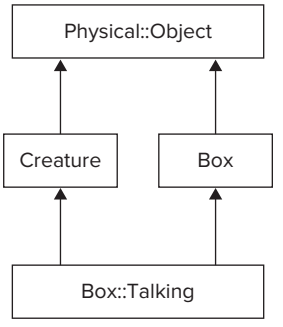

FIGURE 12-2

In this case, when you call `$talking_box->weight()`, it looks for the weight method in `Box::Talking` and, not finding it, searches `Creature`. And failing to find that, it looks for the `weight()` method in `Physical::Object` and calls that. The `Box::weight()` method will never get called. The `Physical::Object` might simply report its weight even though you wanted the `Box` class's `weight()` method because it responds with its weight plus all the objects inside of it.

You could fix that by reversing the order in which you inherit from those:

```
use base qw(Box Creature);
```

Then, when you call `$talking_box->weight()`, you'll get the `weight()` method from `Box`.

You can solve this problem without changing the inheritance order by using something called C3 linearization. (See the `C3` or `mro` modules on the CPAN.) They use a left-most, breadth-first method resolution strategy. Perl would search, in order, `Box::Talking`, `Creature`, `Box`, and then `Physical::Object` methods and would find `Box::weight()` before `Physical::Object::weight()`.

> ***NOTE*** *If Perl cannot find the method, you usually get an error message similar to:*
>
> ```
> Can't locate object method "do_stuff" via package "MyPackage"
> ```
>
> *However, sometimes there is an* `AUTOLOAD` *method available. If Perl does not find the method, it resumes its search through the inheritance hierarchy looking for a method named* `AUTOLOAD` *and calls the first* `AUTOLOAD` *method it finds. We generally do not recommend this because it is tricky to write properly, is slow, and can easily hide errors. See the* `Autoloading` *section in* `perldoc perlsub` *for more information.*

Confused yet? It gets worse.

Now assume that `Box::Talking` has inherited from `Box` first and then Creature?

```
use base ('Box', 'Creature');
```

Now imagine that you have a `move()` method in both `Create` and `Box` and you want to call the `Creature` method instead of the `Box` method? Perl's default method resolution order would be to search `Box::Talking`, `Box`, `Physical::Object`, and then `Creature`. You would never call the `Creature::move()` method.

If you switch to the C3 method resolution order, Perl searches `Box::Talking`, `Box`, `Creature`, and `Physical::Object`. Because it can still find the `Box::move()` method first, you still get the wrong method.

> ***NOTE*** *The order in which Perl searches for the method in classes is called the* method resolution order*, or MRO for short. There is an* `mro` *module on the CPAN that enables you to change the method resolution order.*
>
> *If you don't like Perl's default method order, your author recommends that you do not change it. Simply use the* `Moose` *OO system as explained in Chapter 13. It uses the left-most, breadth-first C3 MRO by default.*
>
> *Fortunately, if you never use multiple inheritance, the MRO issues do not apply to your code.*

You can solve this in your `Box::Talking` class with the following ugly code:

```
sub move {
    my ( $self, $movement ) = @_;
    return $self->Creature::move($movement);
}
```

Calling fully qualified method names like this is legal, but it's not common, and it's a symptom of bad class design. If you decided to refactor your classes, these hard-coded class names in your code can lead to confusing errors.

If this section of the chapter confused you, don't worry. Many good programmers have been bitten by multiple inheritance, and every year there seems to be a new computer science paper describing why it's bad. Strong advice: Even though Perl lets you use multiple inheritance, don't use it unless you're very, very sure you have no other choice.

Chapter 13 explains how to avoid this problem by using `Moose`. (Have we hyped `Moose` enough yet for you?)

SUMMARY

Object-oriented programming is a way to create "experts" for particular problems your software may need to solve. A class is a package and describes all data and behavior the object needs to deal with. The object is a reference blessed into that class. Methods are subroutines and the class name or object is always the first argument.

Inheritance is where you create a more-specialized version of a class. It inherits from another class and gains all its behavior and data, along with adding its own behavior and possibly data. If you call a method on an object and the objects class does not provide that method, Perl searches the objects inheritance tree to find the correct method to call.

All objects ultimately inherit from the `UNIVERSAL` class. This class provides `isa()`, `can()`, and `VERSION()` methods to all classes.

So you've taken a long time to get to this incredibly short summary. Objects in Perl are straightforward, but you've taken the time to consider examples of real-world objects to give you a better idea of what they're often like in Comp code.

EXERCISES

1. Representing people in software systems is a common task. Create a simple `Person` class with a `name` attribute and a `birthdate` attribute. The latter should be a `DateTime` object. Provide a method named `age()` that returns the person's age in years.

Hint: You can use `DateTime->now` to get a `DateTime` object for today's date. Subtracting the person's `birthdate` from today's date returns a `DateTime::Duration` object.

2. The following code works, but it will likely break if you try to subclass it. Why?

```
package Item;
use strict;
use warnings;
sub new {
    my ( $class, $name, $price ) = @_;
    my $self = bless {};
    $self->_initialize( $name, $price );
    return $self;
}
sub _initialize {
    my ( $self, $name, $price ) = @_;
    $self->{name}  = $name;
    $self->{price} = $price;
}
sub name  { $_[0]->{name} }
sub price { $_[0]->{price} }
1;
```

3. Using the `Person` class described in exercise 1 of this chapter, create a `Customer` subclass. Per company policy, you will not accept customers under 18 years of age.

▶ WHAT YOU LEARNED IN THIS CHAPTER

TOPIC	KEY CONCEPTS
Class	An abstract "blueprint" for an object.
Method	A subroutine in a class that takes the class name or object as its first argument.
Object	An "expert" about a problem domain.
`bless`	A builtin that binds a reference to a class.
Inheritance	How Perl creates a more specific version of a class.
Subclass	A more specific type of a class. Also called a child class.
Superclass	The parent of a subclass.
`UNIVERSAL`	The ultimate parent of all classes.