11

Object-Oriented Perl

As we've mentioned before, there are several schools of thought in programming. One in particular has gained a lot of popularity over the past five or ten years – it's called **object-oriented programming**, or **OOP** for short. The type of programming we've been doing so far has been based around tasks – splitting projects up into smaller and smaller tasks, using subroutines to define a single task, and so on. This is called **procedural programming**, because the focus is on the procedures involved in getting the job done. In object-oriented programming, on the other hand, the focus is on the data. Chunks of data called **objects** can have various properties and can interact with each other in various ways.

In this chapter, we'll learn how to start thinking in object-oriented (OO) terms. OO involves a lot of jargon, so the first thing we'll do is look at all the new terms associated with OO and what they mean to a Perl programmer. After that, we'll see how to approach a problem using this style of programming. We'll use some CPAN modules that involve objects, and we'll also construct some object-oriented modules of our own.

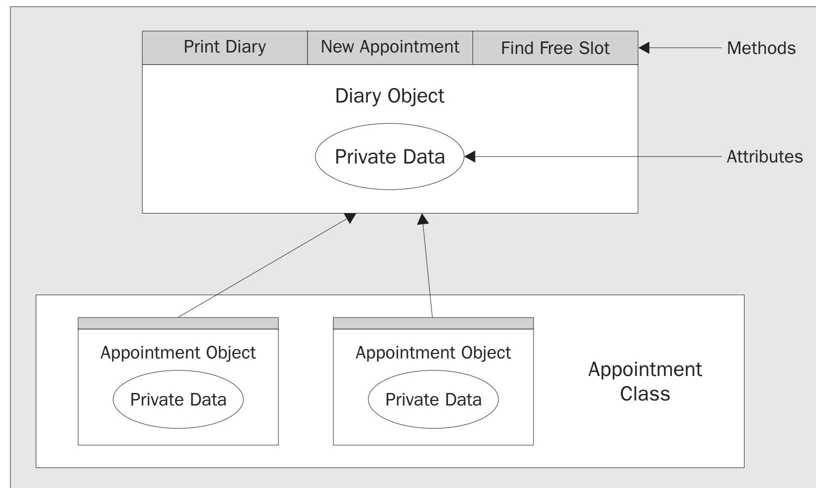
Finally, we'll examine ties, which offer a way to hide the workings of a module behind an ordinary looking variable.

Working with Objects

Procedural programming deals with tasks; the basic unit of operation is the subroutine, which describes how a task is carried out. It's also concerned with breaking tasks down into smaller and smaller stages until they become easy to describe to the computer.

Object-oriented programming, on the other hand, is more concerned with groups of actions and interactions between data. Here, the basic unit of operation is the object (a chunk of data). Attached to that chunk of data is a set of controls that the user can use to interact with it.

For instance, if you're writing a calendar program for keeping track of appointments, there aren't any specific tasks that are fundamental to the usefulness of the calendar. What is fundamental is the chunk of data that represents your diary – so you'd start off by creating a diary object. There are certain things you might want that diary object to do: print out a calendar, tell you today's appointments, and so on. You could also ask that diary object to create an appointment. However, this would require creating a different sort of chunk of data, to represent an appointment. All the data for a given appointment would be stored in an appointment object, which would then be attached to the diary:



Turning Tasks into OO Programs

So how do you decide whether or not you should be using a procedural or an OO style in your programs? Here are five guidelines to help you decide.

Are your subroutines tasks?

If your program naturally involves a series of unconnected tasks, you probably want to be using a procedural style. If your application is **data-driven**, then you're dealing primarily with data structures rather than tasks, so consider using an OO style instead.

Do you need persistence?

After your task is completed, do you need somewhere to store data that you want to receive next time you process that data? If so, you may find it easier to use an OO interface. If each call to a subroutine is completely independent of the others, you can use a procedural interface.

For instance, if you're producing a cross-reference table, your cross-reference subroutine will need to know whether or not the thing it's processing has turned up before or not. Since an object packages up everything we know about a piece of data, it's easy to deal with that directly.

Do you need sessions?

Do you want to process several different chunks of data with the same subroutines? For instance, if you have two different 'sessions' that signify database connections or network connections, you will find it much, much easier to package up each session into an object.

Do you need OO?

Object-oriented programs run slightly slower than equally well-written procedural programs that do the same job, because packaging things into objects and passing objects around is expensive, both in terms of time spent and resources used. If you can get away without using object orientation, you probably should.

Do you want the user to be unaware of the object?

If you need to store an object, but want to hide away any processing you do with it behind an ordinary variable, you need to use a part of object-oriented programming called 'tying'. For example, as we'll see in Chapter 13, you can hide a database object behind an ordinary hash, so that when you look something up in the hash, the object looks it up in the database on disk. When something is stored in the hash, the object writes the data to the disk. The end-user is not necessarily aware that he (or she) is dealing with an object, but some special things happen when s/he accesses the hash.

Are you still unsure?

Unless you know you need an OO model, it's probably better to use a procedural model to help maintenance and readability. If you're still unsure, go with an ordinary procedural model.

Improving Your Vocabulary

'Object-oriented programming' wouldn't be a good 'buzz-phrase' if it didn't use a lot of familiar words in unfamiliar contexts. Before we go any further, let's investigate the jargon terms that we'll need in order to understand Perl OOP.

The first thing to note is that OOP is a concept, rather than a standard. There are a few things that OO languages should do, a lot they can do, but nothing that they absolutely *have* to do. Other languages may implement more or less of these ideas than Perl does and may well do so in a completely different way. We'll explain here the terms that are most commonly used by object-oriented programmers.

Objects

What is an object, anyway? I mentioned briefly above that an object is a chunk of data – but that's not all. To be honest, an object can be anything – it really depends on what your application is. For instance, if you're writing a contact management database, a single contact might be an object. If you're communicating with a remote computer via FTP, you could make each connection to the remote server an object.

An object can always be described in terms of two things:

- what it can do
- what we know about it

With a 'contact record' object, we'd probably know the contact's name, date of birth, address, and so on. These are the object's **attributes**. We might also be able to ask it to do certain things: print an address label for this contact, work out how old they are, or send them an email. These are the object's **methods**.

In Perl, what we see as an object is simply a reference. In fact, you can convert any ordinary reference *into* an object simply by using the `bless` operator. We'll see later on how that happens. Typically, however, objects are represented as references to a hash, and that's the model we'll use in this chapter.

Attributes

As we've just seen, an attribute is something we know about an object. A contact database object will possess attributes such as date of birth, address, and name. An FTP session will possess attributes such as the name of the remote server we're connected to, the current directory, and so on. Two contacts will have different values for their name attribute (unless we have duplicates in the database), but they will both have the name attribute.

If we're using a reference to a hash, it's natural to have the attributes as hash entries. Our person object then becomes a `blessed` version of the following:

```
my $person = {
   surname    => "Galilei",
   forename   => "Galileo",
   address    => "9.81 Pisa Apts.",
   occupation => "bombadier"
};
```

We can get to (and change) our attributes simply by accessing these hash values directly (that is, by saying something like `$person->{address}`. Remember that we use this syntax because we're dealing with a reference), but this is generally regarded as a bad idea. For starters, it requires us to know the internal structure of the object and where and how the attributes are stored which, as an end-user, we should have no need or desire to fiddle with. Secondly, it doesn't give the object a chance to examine the data you're giving it to make sure it makes sense. Instead, access to an attribute usually goes through a **method**.

Methods

A method is anything you can tell the object to do. It could be something complicated, such as printing out address labels and reports, or something simple such as accessing an attribute. Those methods directly related to attributes are called **get-set** methods, as they'll typically either **get** the current value of the attribute, or **set** a new one.

The fact that methods are instructions for doing things may give you a clue as to how we represent them in Perl – methods in Perl are just subroutines. However, there's a special syntax called the 'arrow' operator `(->)`, which we use to call methods. So instead of getting the address attribute directly, as above, we're more likely to say something like this:

```
print "Address: ", $person->address(), "\n";
```

We're also able to set an attribute (change its value) like this:

```
$person->address("Campus Mirabilis, Pisa, Italy");
```

Alternatively, we can call a method to produce an envelope for this object:

```
$person->print_envelope();
```

This syntax `$object->method(@arguments)` 'invokes' the method, which just means that it calls the given subroutine – in our examples this is either `address` or `print_envelope`. We'll see how it's done shortly.

Classes

Our contact object and FTP session object are very different things – they have different methods and attributes. While `$person->date_of_birth()` may make sense, you wouldn't expect, for instance `$ftp_session->date_of_birth()` to do anything sensible.

A **class** is the formal term for a *type* of object. They define the methods an object can have, and how those methods work. All objects in the `Person` class will have the same set of methods and possess the same attributes, and these will be different from the `FTP` class. An object is sometimes referred to as an **instance** of a class, this just means that it's a specific thing created from a general category.

In Perl's object-oriented philosophy, a class is an ordinary package – now let's start piecing this together:

- A method is a subroutine in a package. For instance, the `date_of_birth` method in the `Person` class is merely the subroutine `date_of_birth` in the `Person` package.
- Blessing a scalar just means telling it what package to take its methods from. At that point, it's more than just a complex data structure, or scalar reference. It has attributes – the data we've stored in the hash reference or elsewhere. It has methods – the subroutines in its package, so it can be considered a fully-fledged object.

Classes can also have methods, in order to do things relevant to the whole class rather than individual objects. Instead of acting on an object, as you would by saying `$object->method()`, you act on the class: `$Person->method()`. An important thing to note is that Perl doesn't necessarily know whether a given subroutine is a class method, an object method, or just an ordinary subroutine, so the programmer has to do the checking himself.

Similarly, classes can have attributes that refer to the whole class – in Perl these are just package variables. For instance, we might have a `population` attribute in our `Person` class, which tells us how many `Person` objects are currently in existence.

One final note – you'll probably have noticed that we capitalized `$Person`. It's quite usual in Perl to capitalize all class names, so as to distinguish them from object names.

Polymorphism

The word **polymorphism** comes from the Greek •••• ••••••, meaning 'many forms'. What it means in object-oriented programming is that a single method can do different things depending on the class of the object that calls it. For instance, `$person->address()` would return the person's address, but `$ftp_session->address()` might return the IP address of the remote server. On the other hand, `$object->address()` would *have* to do the right thing according to which class `$object` was in.

In some other OO programming languages, you have to ensure that the single method `address` can cope with the various classes that make sense for it. Perl's OO model, on the other hand, neatly sidesteps the problem. This is because `$person->address()` and `$ftp_session->address()` aren't a single method at all. In fact, they're two different methods, because they refer to different subroutines.

One of these is the subroutine `Person::address`, and the other is the subroutine `FTP::address`. They're defined completely separately, in different packages, possibly even in different files. Since perl already knows what class each object belongs to, neither you nor perl need to do anything special to make the distinction. Perl looks at the object, finds the class it is in, and calls the subroutine in the appropriate package. This brings us on to…

Encapsulation

One of the nice things about object-oriented programming is that it hides complexity from. The user this is known as **encapsulation** (or **abstraction**). This means that you needn't care how the class is structured, or how the attributes are represented in the object. You don't have to care how the methods work, or where they come from. You just use them.

This also means that the author of the class has complete freedom to change its internal workings at any time. As long as the methods have the same names and take the same arguments, all programs using the class should continue to work and produce the same results. That's as long as they use the method interface as they should, rather than trying to poke at the data directly.

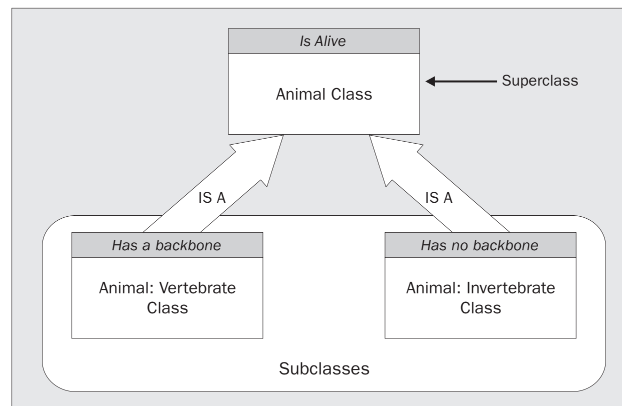
In this sense, working with objects is a little like driving a car. Our object, the car, has a set of attributes, such as the model, current speed, and amount of fuel in the tank. We can't get at these directly, but some read-only methods like the speedometer and the fuel gauge expose them to us. It also provides us with some more methods and a well-defined interface to get it to do things.

We have a pedal to make it accelerate and one to make it brake, a stick to change gear, a hole to put fuel into the tank, and so on. We don't actually need to know how the engine works if we're prepared to stick to using these methods, of course we do need to know what each of them does. We don't even need to know the whereabouts of the fuel tank, we just put fuel in the appropriate hole. If we really want to, we can take the hood off, look inside it, and fiddle with it. But then we only have ourselves to blame if it breaks!

Inheritance

Another property that makes OOP easy to use is its support for **inheritance**. Classes can be built quickly by specifying how they differ from other classes. For example, humans inherit attributes from their parents, such as hair color and height, while Perl's classes inherit methods. If I inherit from a class, I receive the ability to call every method that class defines. If your class wants to implement a method differently, you define the method in my class. If you don't, you automatically get the method from the parent class. The parent classes, which provide your class with the methods, are called **superclasses**, and your class is a **subclass** of them.

The relationship between the classes can be described as an '**IS-A**' relationship. If you have a superclass `Animal`, you may create a subclass `Vertebrate`. You could then say that a `Vertebrate` IS-A `Animal`. In fact, the classification system for animals can be thought of as a series of IS-A relationships, with more specific subclasses inheriting properties of their superclasses:



Here we see that vertebrates and invertebrates are both subclasses of a general animal class. They both inherit the fact that they are alive, and so we need not specify this in the subclass. Next we could create an `Animal::Vertebrate::Mammal` class, which would be a subclass of `Animal::Vertebrate`. We wouldn't need to specify that the mammal had a backbone or was alive, because these characteristics would be inherited from the superclass.

Constructors

Objects have to come from somewhere, and in keeping with the principles of encapsulation, the user of a class shouldn't be expected to put together an object himself. This would require knowledge of how the object is represented and what initialization is required. To take this responsibility away from the user, there's a class method that all classes should possess – it's called the **constructor**.

As the name implies, this constructs and returns a new object. For this reason, it's usually called `new()`. We may pass arguments to the constructor, which it can then use to do the initial setup of the object. Sometimes these arguments are in the form of a hash, allowing us to create an object like this:

```
my $galileo = Person->new(
   surname    => "Galilei",
   forename   => "Galileo",
   address    => "9.81 Pisa Apts.",
   occupation => "bombadier",
);
```

There's also another syntax for calling methods, which you'll particularly see used with the constructor:

```
my $galileo = new Person (...);
```

This is supported for the benefit of C++ programmers, as that language uses the `method Class ()` syntax instead of the more Perl-like `Class->method()`. Recognize it, but try and avoid using it.

The constructor will now check that the arguments are acceptable, do any conversion it requires, make up a hash reference, `bless` it and return it to us.

Destructors

When the object is no longer in use – when it's a lexical variable that goes out of scope, perl automatically destroys it. However, before doing so, perl will attempt to call a method called `DESTROY`. If the class provides this method, it should be responsible for any tasks that need to be performed before the object is disposed of. For instance, our FTP session object will want to ensure that it has closed the connection to the remote server.

Try It Out : Using a Net::FTP Object

We'll now use the `Net::FTP` module once again, to create an object that will let us get files from an FTP site. In our case, we'll connect to CPAN and download the `readme` file:

```
#!/usr/bin/perl
# ftp.plx
use warnings;
use strict;
use Net::FTP;
```

```
my $ftp = Net::FTP->new("ftp.cpan.org")
   or die "Couldn't connect: $@\n";
$ftp->login("anonymous");
$ftp->cwd("/pub/CPAN");
$ftp->get("README.html");
$ftp->close;
```

Network and firewalls permitting, this should retrieve the file – although it may take some time:

```
> perl ftp.plx
> dir README.html
README~1 HTM          2,902       ...    ...      README.html
>
```

How It Works

After loading the `Net::FTP` module, we create ourselves an object:

```
my $ftp = Net::FTP->new("ftp.cpan.org")
   or die "Couldn't connect: $@\n";
```

Our class is called `Net::FTP`, the same as the module – this is because, as we mentioned above, a class is just an ordinary package.

We create the object by calling the constructor, which is the class method `new`. This takes a number of arguments: a remote machine to connect to and a hash specifying things like whether we have a firewall, which port to connect to, whether we want debugging information, and so on. These arguments will become the attributes of the object. If we don't specify them, the constructor comes up with some sensible defaults for us. In our case, the defaults are fine, so we just need to supply a remote machine – we'll use the CPAN server, `ftp.cpan.org`.

Now we know that a Perl method is just a subroutine. A method call, with the arrow operator, is like calling a subroutine, but with two important differences. First inheritance means that perl won't immediately complain if the subroutine doesn't exist, but instead will look first to see if the subroutine is in each of the superclasses. Second, what goes before the arrow becomes the first parameter of the subroutine.

So, we could, if we wanted to, rewrite our call to the constructor like this:

```
my $ftp = Net::FTP::new("Net::FTP","ftp.cpan.org")
      or die "Couldn't connect: $@\n";
```

However, although this shows us exactly what's happening in excruciating detail, it's rather unwieldy, and disguises the fact that we're dealing with objects.

When we call the constructor, it takes our argument (the remote host) and stashes it away in a hash – encapsulation means we don't need to know how or where. Then it takes a reference to that hash, `bless`es the reference, and returns it to us. That `bless`ed reference is our new object (our FTP session), and we're now ready to do things with it:

```
$ftp->login("anonymous");
```

First of all, we have to log in to the server. The usual way of getting things from an FTP server is by logging in with a username of 'anonymous' and your email address as the password. The `login` method tells the object to issue the appropriate login commands. As before, this is an ordinary subroutine. It could be written like this:

```
Net::FTP::login($ftp,"anonymous");
```

How did perl know that it should use `Net::FTP::login` rather than any other `login`? Well, when our constructor `bless`ed the reference, it gave the reference knowledge of where to find the methods. To quote from the `perlobj` documentation, "an object is just a reference that happens to know which class it belongs to".

Since perl takes care of passing the object to the subroutine as the first parameter, the sub automatically receives all the data it needs. This means we can easily have multiple objects doing different things:

```
my $ftp1 = Net::FTP->new("ftp.cpan.org");
my $ftp2 = Net::FTP->new("ftp.wrox.com");
$ftp1->login("anonymous");
```

The object `$ftp1` is just a `blessed` reference to a hash, and that hash contains all the data about the connection to CPAN, like the settings, the filehandles, and anything else that `Net::FTP` needs to store. These are the object's attributes. Everything we know about the connection is bundled into that object. The important thing to note is that it's completely independent from `$ftp2`, which is another hash containing another set of data about a different connection. Hence, the method call `$ftp1->login()` has no impact on the other connection at all.

What's the difference between a class method and an object method? Well, nothing really as far as perl is concerned. Perl doesn't make a distinction. In theory, there's nothing to stop you saying something like `Net::FTP->login()` or `$ftp->new()`. However, because a class method expects a string as it's first parameter and an object method expects an object, they're likely to get confused if you give them the wrong thing. Alternatively, if the class author is particularly conscientious, the method will check whether it is being called on a class or an object and take the appropriate action. On the other hand, don't expect it to work:

```
> perl -e 'use Net::FTP; Net::FTP->login'
Can't use string ("Net::FTP") as a symbol ref while "strict refs" in use at
/usr/local/lib/perl5/site_perl/5.6.0/Net/FTP.pm line 238.
>
```

Here, the `login()` method was called as a class method. As we saw with `new` above, when you call a class method, perl calls the subroutine with the class name as the first parameter, in effect it's doing this:

```
Net::FTP::login("Net::FTP");
```

Because `login` was written to be an object method rather than a class method, it's expecting to get an object rather than the name of a class. Now an object, as we know, is just a `blessed` reference. When `login` tries to use the parameter it's got as a reference, it finds out that it's actually a string, and perl gives the error message you see above.

So, while perl calls object and class methods in exactly the same way, because a method is usually written to be one rather than another, it's likely to blow up if you call it inappropriately.

Anyway, back to our example! After logging in, we change directory and get the file:

```
$ftp->cwd("/pub/CPAN");
$ftp->get("README.html");
```

`cwd` and `get` are two more methods our object supplies. The object has a huge number of methods, due to the fact that it has a long chain of inheritance. It inherits from a superclass, which inherits from a superclass, which inherits from a superclass and so on. However, there are some methods which `Net::FTP` defines directly and which you should know about. They mainly relate directly to FTP commands – here is an incomplete list of them:

Method Name	Behaviour
`$session->login($login,$passwd)`	Logs into the server with the given username and password.

`$session->type($type)`
`$session->ascii()`
`$session->binary()`
	Set the transfer type to ASCII or binary: this is quite similar to Perl's `binmode` operator.

`$session->rename($old,$new)`	Rename a file.
`$session->delete($file)`	Delete a file.
`$session->cwd($directory)`	Change directory.
`$session->pwd()`	Give the name of the current directory.
`$session->ls()`	List the current directory.
`$session->get($remote, $local, $offset)`	Get a file from the remote server.
`$session->put($local, $remote)`	Put a file to the remote server.

There are also some get-set methods that will affect the object's attributes. For instance, the `$session->hash()` method controls an attribute that determines whether or not to print # signs after every 1024 bytes transferred.

After we've called the `get` method to get our file, we'll call the `close` method to shut down the connection to the server.

```
$ftp->close;
```

Again, this is equivalent to `Net::FTP::close($ftp)`, but more convenient.

So, we've used our first class. Hopefully, you can see that using objects and classes in Perl is just as easy as using subroutines. In fact, it's easier – perl not only takes care of finding out where to find the subroutine you're trying to call, but it also takes care of passing a whole bunch of data to the subroutine for you.

Because this all goes on behind our backs, we can happily pretend that an object contains a bunch of methods that act on it, and it alone. In fact, it doesn't – it only contains information regarding where to find methods that can act on any object in that class.

Rolling Your Own

We've now seen how to use a class and an object. Let's now see how to make our own classes. As an example, we'll implement the `Person` class we used in our definitions.

As we mentioned above, a class is just a package – nothing more, nothing less. So the simplest class looks like this:

```
package Person;
```

That's it. Of course, it has nothing – no methods, no attributes, no constructor, nothing. It's a totally empty class. Usually, you'll want to put your class into a module. It's not necessary by any means, but it gets the implementation out of the way. So, let's create a module, and put the following in the file `Person1.pm`

```
package Person;
# Class for storing data about a person
#person1.pm
use warnings;
use strict;

1;
```

Remember that we need the `TRUE` value as the last thing to tell Perl that everything went OK when loading the file. Now in a separate program, we can say `use Person` and start using the class. However, we can't create any objects yet, because we don't have a constructor. So, how do we write a constructor?

Well, what does our constructor create? It creates an object, which is a `bless`ed reference. Before we go any further, then, let's have a look at what `bless` is and what it does.

Bless You, My Reference

The `bless` operator takes a reference and turns it into an object. The way it does that is simple: It changes the type of the reference. Instead of being an array reference or a hash reference, perl now thinks of it as a `Person` reference (or whatever other class we `bless` the reference into).

As we saw in Chapter 7, we can use the `ref` operator to tell us what type of reference we have. To refresh your memory:

```
#!/usr/bin/perl
# reftypes.plx
use warnings;
use strict;

my $a = [];
my $b = {};
my $c = \1;
my $d = \$c;
print '$a is a ', ref $a, " reference\n";
print '$b is a ', ref $b, " reference\n";
print '$c is a ', ref $c, " reference\n";
print '$d is a ', ref $d, " reference\n";
```

```
>perl reftypes.plx
$a is a ARRAY reference
$b is a HASH reference
$c is a SCALAR reference
$d is a REF reference
>
```

Now let's see what happens when we use `bless`. The syntax of `bless` is:

```
bless( <reference>, <package> );
```

If the package isn't given, the reference is `bless`ed into the current package. Let's `bless` a reference into the `Person` package:

```
#!/usr/bin/perl
# bless1.plx
use warnings;
use strict;

my $a = {};

print '$a is a ', ref $a, " reference\n";

bless($a, "Person");

print '$a is a ', ref $a, " reference\n";
```

```
>perl bless1.plx
$a is a HASH reference
$a is a Person reference
>
```

Okay, so we've changed `$a` into a `Person` reference. So what just happened?

Actually, nothing changed in the structure of `$a` at all. It's still a hash reference, and we can still dereference it – or add, access, and delete entries in the hash, and so on. It still has the same keys and values. Nothing magical has happened.

But `$a` is now a reference with knowledge of which package it belongs to. If we try and call a method on it, perl now knows that it should look in the `Person` package for a definition of that method. It's become an object.

What if we `bless` it again? What happens then? Let's try it.

```
#!/usr/bin/perl
# bless2.plx
use warnings;
use strict;

my $a = {};
```

```
print '$a is a ', ref $a, " reference\n";

bless($a, "Person");
print '$a is a ', ref $a, " reference\n";

bless($a, "Animal::Vertebrate::Mammal");
print '$a is a ', ref $a, " reference\n";
```

```
> perl bless2.plx
$a is a HASH reference
$a is a Person reference
$a is a Animal::Vertebrate::Mammal reference
>
```

All that's happened is we've once again changed what type of reference it is. We've changed where perl should look if any methods are called on the reference. Note that at this stage we haven't even **defined** an `Animal::Vertebrate::Mammal` package, but that's OK because we're not going to call any methods yet – if we did, they would surely fail.

Again, the internal structure of that reference hasn't changed. It's still a hash reference with the same keys and values. You usually don't want to `bless` an object that's already been `blessed`. This is because something that was originally a `Person` may have different attributes to what the new class expects it to have when methods are called. Worse still, the program using the object could well try and call a method that was fine in the old class but doesn't exist in the new one – attempting to magically turn a person into an FTP session can only have undesirable (and pretty weird) results.

Storing Attributes

We've got an object. Before we look at methods, let's examine attributes. An attribute is, as we defined it at the start of this chapter, something we know about the object. In other words, it's a piece of data that belongs to this particular object. How do we store this data, then?

Well, this is what the reference is for, if we store our data in the reference, our object carries around both a set of data unique to it, plus knowledge of where to find methods to act on that data. If we know that our object is only going to contain one attribute (one piece of data), we could conceivably use a scalar reference, like this:

```
my $attribute = "green";
my $object = \$attribute;
bless($object, "Simple");
```

Now we have a nice simple object that stores a single attribute contained in the `Simple` class. We can access and change the attribute just as we'd work with an ordinary scalar reference:

```
$attribute = $$object;
$$object = "red";
```

This is nice and simple, but it's not very flexible. Similarly, we could have an array reference and `bless` that to turn it into an object, which is slightly more flexible. We can access attributes as elements in the array, and we can add and delete attributes by using array operations. If we are storing a set of unnamed data, this is perfectly adequate.

However, for maximum flexibility, we can use a hash to give names to our attributes:

```
my $object = {
   surname    => "Galilei",
   forename   => "Galileo",
   address    => "9.81 Pisa Apts.",
   occupation => "bombadier",
};
bless $object, "Person";
```

This allows us easy access to the individual attributes, as if we were carrying a bunch of variables around with us. Therefore, we generally use a hash reference for any non-trivial class.

The Constructor

We're now ready to create objects. Let's put this knowledge into a constructor, and put a constructor into our currently empty `Person` class.

Try It Out : Our First Constructor

To construct an object, we make a hash reference, and bless it into the class. That's all we need to do:

```
package Person;
# Class for storing data about a person
#person2.pm
use warnings;
use strict;

sub new {
   my $self = {};
   bless ($self, "Person");
   return $self;
}

1;
```

Now we can use our `Person` class to create an object:

```
#/usr/bin/perl
# persontest.plx
use warnings;
use strict;
use Person2;

my $person = Person->new();
```

which should execute without any errors!

How It Works

Our constructor does a simple job, and does it well. First, we create our hash reference:

```
   my $self = {};
```

`$self` is the traditional name for an object when it's being manipulated by methods inside the class. Now we'll turn it into an object by telling it which class it belongs to:

```
bless ($self, "Person");
```

Finally, we send the object back:

```
return $self;
```

Excellent. Now let's see how we can improve this.

Considering Inheritance

The first thing we've got to think about is inheritance. It's possible that someone, someday will want to inherit from this class, and we won't necessarily get told about it. If they don't provide their own constructor, they'll get ours, and as things stand, that'll produce an object blessed into our class – not theirs.

We really need to remove the hard-wired `"Person"` in our constructor and replace it with the called class. How do we know what the called class is though? Well, the thing to remember is that Perl translates `Class->new()` into `new("Class")`. We know what class the user wants, because it's the first argument to the constructor. All we need to do is take that argument and use that as the class to `bless` into. So here's a more general constructor that takes inheritance into account:

```
sub new {
   my $class = shift;
   my $self = {};
   bless($self, $class);
   return $self;
}
```

As usual, `shift` without any arguments means `shift @_`. It takes the first element of the argument array. This gives us the first thing we were passed, the class name. We can therefore use this to bless our reference, without needing to hard-code the name in.

Providing Attributes

Now let's make one more enhancement. At the moment, we can create a completely anonymous `Person` with no attributes at all. We can give the end-user of the class the opportunity to specify some attributes when the `Person` is created.

Try It Out : Initializing Attributes In The Constructor

As before, we're going to store the data in the hash reference. We'll take the data as arguments to the constructor. Ideally, we'll want the constructor to be called something along these lines:

```
my $object = Person->new (
   surname    => "Galiiei",
   forename   => "Galileo",
   address    => "9.81 Pisa Apts.",
   occupation => "bombardier"
);
```

This is the easiest syntax for the user, because it allows them to specify the attributes in any order and give as many or as few as they want. It's also a lot easier to use and remember than if we make them use a list like this:

```
my $object = Person->new ("Galilei","Galileo","9.81 Pisa Apts.","bombardier");
```

In fact, it's the easiest syntax for us, too. Since we want our attributes stored in a hash, and the key-value syntax we propose above **is** a hash, all we have to do is place the arguments straight into our hash reference:

```
my $self = {@_};
```

Let's plug this into our package:

```
package Person;
# Class for storing data about a person
#person3.pm
use warnings;
use strict;

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

1;
```

How It Works

What have we done? Well, now when we call the constructor, Perl sees something like this:

```
my $object = Person::new("Person",
   "surname",   "Galilei",
   "forename",  "Galileo",
   "address",   "9.81 Pisa Apts.",
   "occupation","bombardier"
);
```

The first line of the constructor takes up the class name as before:

```
   my $class = shift;
```

Now what's left in the argument array `@_` is what we specified when we called the constructor:

```
@_=(
   "surname",   "Galilei",
   "forename",  "Galileo",
   "address",   "9.81 Pisa Apts.",
   "occupation","bombardier"
);
```

This is what we put verbatim into our hash reference:

```
my $self = {@_};
```

Our hash now contains all the attributes we provided. As usual, it's `blessed` and returned to the caller.

> `my $self` *creates a lexical variable, which is destroyed once the subroutine ends. Doesn't this mean that all our attributes will get wiped out too? Nope – this is exactly why we use a reference. Perl will never destroy data while a reference to it exists (see the section on 'Reference Counting' in Chapter 7) our data will persist everywhere the object goes.*

We've now got a fully featured constructor. We've taken some initial data and constructed an object out of it, storing the data as attributes in the object. Now it's time to add some methods so we can actually do something with it!

Creating Methods

Our constructor was a class method; creating an object method will be very similar. In the same way that a class method gets passed the name of the class as the first argument, an object method is just a subroutine that gets passed the object as the first argument.

Try It Out : A Simple Accessor

Let's create a method to return the surname of the person. This directly accesses an attribute – sometimes called an **accessor method**. Remember that the surname attribute is just an entry in the hash, referenced by the object – so what does this involve? We'll need to:

- receive the object being passed to us
- extract the 'surname' entry from the hash
- pass it back to the caller

Here's how we'd code it:

```
sub surname {
   my $self = shift;
   my %hash = %$self;
   return $hash{surname}
}
```

However, with the techniques we learned in Chapter 7 for directly accessing values in a hash reference, we can trim it down a bit and add it into our class:

```
package Person;
# Class for storing data about a person
#person4.pm
use warnings;
use strict;
```

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

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

1;
```

Now we can create an object with some attributes and retrieve the attributes again:

```
#!/usr/bin/perl
# accessor1.plx
use Person4;

my $object = Person->new (
   surname    => "Galilei",
   forename   => "Galileo",
   address    => "9.81 Pisa Apts.",
   occupation => "bombadier"
);
print "This person's surname: ", $object->surname, "\n";
```

If all's well, we should be told the surname:

> perl accessor1.plx
This person's surname: Galilei
>

How It Works

Our method is a very simple one – it takes an object and extracts an attribute from that object's data store. First, we use `shift` to get the object passed to us:

```
my $self = shift;
```

Then we take out the relevant hash entry and pass it back:

```
return $self->{surname}
```

Don't confuse the arrow used here for accessing parts of a reference with the arrow used as a method call. When accessing a reference, there will always be some kind of brackets at the end of the arrow:

```
$reference->{surname}; # Accesses a hash reference
$reference->[3];       # Accesses an array reference
$reference->();        # Accesses a function reference
```

When calling a method, there will be a name following the arrow:

```
$reference->surname;
```

There may be brackets after that if parameters are being passed to the method or if the programmer wants to use () to be unambiguous:

```
$reference->surname();
```

> **A direct access to a reference will always have some kind of bracket after the arrow, no matter what type of reference. A method call, whether on a class or on an object, will have the name of the method directly after the arrow with no brackets in between.**

So while our method is called with `$object->surname`, the surname entry in the hash is accessed with `$self->{surname}`. Thankfully, the only time you'll see this is when you're creating accessors. An accessor is the only place where it's safe to directly access part of an object's reference. Accessors are the interface through which everything talks to the attributes.

The reasoning behind this is, as before, inheritance. If someone inherits a method from your class but changes the internal representation of the object slightly, they'll be stuck if some of your methods access the object directly. For instance, if an inheriting class decides that it should take data from a file on disk but your methods persist in accessing data inside a hash reference, your methods won't be usable. However, if you always use accessor methods, the inheriting class can provide its own accessors, which will use the file instead of a hash, and all will be well.

Distinguishing Class and Object Methods

Now, we've mentioned a few times that Perl doesn't distinguish between class and object methods. What if **we** want to? Our `surname` method only makes sense with an object, and not with a class, so if our surnames are called incorrectly, we want to shout about it.

Well, the thing that lets us know how we're being called is that first parameter. If we have a string, we're being called as a class method. If we have a reference, we're being called as an object method. So, let's make one more alteration to our accessor method to trap incorrect usage.

Try It Out : Checking Usage

If our accessor is called with something that isn't an object, we'll give an error message and blame the programmer:

```
package Person;
# Class for storing data about a person
#person5.pm
use warnings;
use strict;
use Carp;

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

```
sub surname {
   my $self = shift;
   unless (ref $self) {
      croak "Should call surname() with an object, not a class";
   }
   return $self->{surname}
}

1;
```

Now if we add the following line to the end of our `accessor1.plx` file:

```
Person->surname;
```

we should generate the following complaint:

```
>perl accessor1.plx
This object's surname: Galilei
Should call surname() with an object, not a class at accessor.plx line 12
>
```

How It Works

We use the `ref` operator to make sure that what we're being passed is actually a reference. We could be more stringent, and ensure that it's an object in the `Person` class, but again that would break inheritance.

Get-Set Methods

As well as getting the value of an attribute, we may well want to set or change it. The syntax we'll use is as follows:

```
print "Old address: ", $person->address(), "\n";
$person->address("Campus Mirabilis, Pisa, Italy");
print "New address: ", $person->address(), "\n";
```

This kind of accessor is called a get-set method, because we can use it to both get and set the attribute. Turning our current read-only accessors into accessors that can also set the value is simple. Let's create a get-set method for `address`:

```
sub address {
   my $self = shift;
   unless (ref $self) {
      croak "Should call address() with an object, not a class";
   }

   # Receive more data
   my $data = shift;
   # Set the address if there's any data there.
   $self->{address} = $data if defined $data;

   return $self->{address}
}
```

If we don't particularly want to trap calling the method as a class method (since it'll generate an error when we try to access the hash entry, anyway), we can write really miniature get-set methods like this:

```
sub address  { $_[0]->{address }=$_[1] if defined $_[1]; $_[0]->{address } }
sub surname  { $_[0]->{surname }=$_[1] if defined $_[1]; $_[0]->{surname } }
sub forename { $_[0]->{forename}=$_[1] if defined $_[1]; $_[0]->{forename} }
```

While that's fine for getting classes up and running quickly, writing out the get-set method in full as above allows us to easily extend it in various ways, like testing the validity of the data, doing any notification we need to when the data changes, and so on.

Class Attributes

Classes can have attributes, too. Instead of being entries in a hash, they're variables in a package. Just like object attributes it's a really good idea to access them through get-set methods, but since they're ordinary variables, our methods are a lot simpler. Let's use a class attribute to keep score of how many times we've created a `Person` object. We'll call our attribute `$Person::Population`, and we'll get the current value of it via the method `headcount`.

Try It Out : Adding A Class Attribute

A class attribute is a package variable, and an accessor method just returns or sets the value of that variable. Here, we make our accessor method read-only, to stop the end user changing it and confusing his own code:

```
package Person;
# Class for storing data about a person
#person6.pm
use warnings;
use strict;
use Carp;

my $Population = 0;

sub new {
   my $class = shift;
   my $self = {@_};
   bless($self, $class);
   $Population++;
   return $self;
}

# Object accessor methods
sub address  { $_[0]->{address }=$_[1] if defined $_[1]; $_[0]->{address } }
sub surname  { $_[0]->{surname }=$_[1] if defined $_[1]; $_[0]->{surname } }
sub forename { $_[0]->{forename}=$_[1] if defined $_[1]; $_[0]->{forename} }
sub phone_no { $_[0]->{phone_no}=$_[1] if defined $_[1]; $_[0]->{phone_no} }
sub occupation  {
   $_[0]->{occupation}=$_[1] if defined $_[1]; $_[0]->{occupation}
}

# Class accessor methods
sub headcount { $Population }

1;
```

Now as we create new objects, the population increases:

```
#!/usr/bin/perl
# classatr1.plx
use warnings;
use strict;
use Person6;

print "In the beginning: ", Person->headcount, "\n";
my $object = Person->new (
   surname    => "Galilei",
   forename   => "Galileo",
   address    => "9.81 Pisa Apts.",
   occupation => "bombadier"
);
print "Population now: ", Person->headcount, "\n";

my $object2 = Person->new (
   surname    => "Einstein",
   forename   => "Albert",
   address    => "9E16, Relativity Drive",
   occupation => "Plumber"
);
print "Population now: ", Person->headcount, "\n";
```

```
>perl classatr1.plx
In the beginning: 0
Population now: 1
Population now: 2
>
```

How It Works

There's actually nothing OO specific about this example. All we're doing is taking advantage of the way Perl's scoping works. A lexical variable can be seen and used by anything in the current scope and inside any brackets. So, naturally enough, with:

```
Package Person;
my $Population;

sub headline { $Population }
```

the package variable `$Population` is declared at the top of the package and is therefore visible everywhere in the package. Even though we call `headcount` from another package, it accesses a variable in its own package.

Similarly, when we increase it as part of `new`, we're accessing a variable in the same package. Since it's a package variable, it stays around for as long as the package does, which is why it doesn't lose its value when we do things in our main program.

Let's make one more addition. We'll allow our main program to go over all of the names of people in our contacts database, and we'll have a class method to give us an array of the objects created. Instead of keeping a separate variable for the population, we'll re-implement `$Population` in terms of the scalar value of that array:

```
package Person;
# Class for storing data about a person
#person7.pm
use warnings;
use strict;
use Carp;

my @Everyone;

sub new {
   my $class = shift;
   my $self = {@_};
   bless($self, $class);
   push @Everyone, $self;
   return $self;
}

# Object accessor methods
sub address  { $_[0]->{address }=$_[1] if defined $_[1]; $_[0]->{address } }
sub surname  { $_[0]->{surname }=$_[1] if defined $_[1]; $_[0]->{surname } }
sub forename { $_[0]->{forename}=$_[1] if defined $_[1]; $_[0]->{forename} }
sub phone_no { $_[0]->{phone_no}=$_[1] if defined $_[1]; $_[0]->{phone_no} }
sub occupation  {
   $_[0]->{occupation}=$_[1] if defined $_[1]; $_[0]->{occupation}
}

# Class accessor methods
sub headcount { scalar @Everyone }
sub everyone  { @Everyone        }

1;
```

Note that we're pushing one reference to the data onto the array, and we return another reference. There are now two references to the same data, rather than two copies of the data. This becomes important when it comes to destruction. Anyway, this time we can construct our objects and look over them:

```
#!/usr/bin/perl
# classatr2.plx
use warnings;
use strict;
use Person7;

print "In the beginning: ", Person->headcount, "\n";
my $object = Person->new (
   surname    => "Galilei",
   forename   => "Galileo",
   address    => "9.81 Pisa Apts.",
   occupation => "bombadier"
);
print "Population now: ", Person->headcount, "\n";

my $object2 = Person->new (
   surname    => "Einstein",
   forename   => "Albert",
   address    => "9E16, Relativity Drive",
   occupation => "Plumber"
);
```

```
print "Population now: ", Person->headcount, "\n";

print "\nPeople we know:\n";
for my $person(Person->everyone) {
   print $person->forename, " ", $person->surname, "\n";
}
```

```
>perl classatr2.plx
In the beginning: 0
Population now: 1
Population now: 2

People we know:
Galileo Galilei
Albert Einstein
>
```

Normally, you won't want to do something like this. It's not the class's business to know what's being done with the objects it creates. Since we know that in these examples we'll be putting all the Person objects into a database, it's reasonable to get the whole database with a single method. However, this isn't a general solution – people may not use the objects they create, or may use them in multiple databases, or in other ways you haven't thought of. Let the user keep copies of the object themselves.

Privatizing Your Methods

The things we did with our class attributes in `new` in the two examples above were a bit naughty. We directly accessed the class variables, instead of going through an accessor method. If another class wants to inherit, it has to make sure it too carries a package variable of the same name in the same way.

What we usually do in these situations is to put all the class-specific parts into a separate method and use that method internally in the class. Inheriting classes can then replace these **private methods** with their own implementations. To mark a method as private, for use only inside the class, it's customary to begin the method's name with an underscore. Perl doesn't treat these methods any differently – the underscore means nothing significant to Perl but is purely for human consumption. Think of it as a 'keep out' sign, to mark the method as: for use by authorized personnel only!

Try It Out : Private Methods

Typically, the constructor is one place where we'll want to do a private set-up, so let's convert the code for adding to the `@Everyone` array into a private method:

```
package Person;
# Class for storing data about a person
#person8.pm
use warnings;
use strict;
use Carp;

my @Everyone;

# Constructor and initialisation
sub new {
   my $class = shift;
```

```
    my $self = {@_};
    bless($self, $class);
    $self->_init;
    return $self;
}

sub _init {
    my $self = shift;
    push @Everyone, $self;
    carp "New object created";
}

# Object accessor methods
sub address  { $_[0]->{address }=$_[1] if defined $_[1]; $_[0]->{address } }
sub surname  { $_[0]->{surname }=$_[1] if defined $_[1]; $_[0]->{surname } }
sub forename { $_[0]->{forename}=$_[1] if defined $_[1]; $_[0]->{forename} }
sub phone_no { $_[0]->{phone_no}=$_[1] if defined $_[1]; $_[0]->{phone_no} }
sub occupation  {
    $_[0]->{occupation}=$_[1] if defined $_[1]; $_[0]->{occupation}
}

# Class accessor methods
sub headcount { scalar @Everyone }
sub everyone  { @Everyone        }

1;
```

Try It Out

What we've got now is pretty much the standard constructor, let's go over it again:

```
sub new {
```

First, we retrieve our class name, which will be passed to us automatically when we do `Class->new`, by using shift as a shorthand for `shift @_`

```
    my $class = shift;
```

Then we put the rest of the arguments, which should be a hash with which to initialize the attributes, into a new hash reference:

```
    my $self = {@_};
```

Now we bless the reference to tell it which class it belongs in, making it an object:

```
    bless($self, $class);
```

Do any further initialization we need to do by calling the object's `private _init` method. Note that due to inheritance, this private method may be provided by a subclass.

```
    $self->_init;
```

Finally, return the constructed object:

```
   return $self;
}
```

Utility Methods

Our methods have mainly been accessors so far, but that's by no means all we can do with objects. Since methods are essentially subroutines, we can do almost anything we want inside them. Let's now add some methods that do things – **utility methods**:

```
# Class for storing data about a person
#person9.pm
use warnings;
use strict;
use Carp;

my @Everyone;

# Constructor and initialisation
#...

# Object accessor methods
#...

# Class accessor methods
#...

# Utility methods
sub fullname {
   my $self = shift;
   return $self->forename." ".$self->surname;
}

sub printletter {
   my $self    = shift;
   my $name    = $self->fullname;
   my $address = $self->address;
   my $forename= $self->forename;
   my $body    = shift;
   my @date    = (localtime)[3,4,5];
   $date[1]++;      # Months start at 0! Add one to humanise!
   $date[2]+=1900;  # Add 1900 to get current year.
   my $date    = join "/", @date;

   print <<EOF;
$name
$address

$date

Dear $forename,

$body

Yours faithfully,
EOF
   return $self;
}

1;
```

This creates two methods, `fullname` and `printletter`. `fullname` returns the full name of the person the object describes. `printletter` prints out a letter with a body supplied by the user. Notice that to print the name in the text of the letter, `printletter` itself calls `fullname`. It's good practice for utility methods to return the object if they have nothing else to return. This allows you to string together calls by using the returned object as the object for the next method call, like this: `$object->one()->two()->three();`

Here's an example of those utility methods in use.

```
#!/usr/bin/perl
# utility1.plx
use warnings;
use strict;
use Person9;

my $object = Person->new (
   surname    => "Galilei",
   forename   => "Galileo",
   address    => "9.81 Pisa Apts.",
   occupation => "bombadier"
);
$object->printletter("You owe me money. Please pay it.");
```

This produces our friendly demand:

```
> perl utility1.plx
Galileo Galilei
9.81 Pisa Apts.

4/5/2000

Dear Galileo,

You owe me money. Please pay it.

Yours faithfully,
>
```

Death of an Object

We've seen how we construct an object, and we've made ourselves a constructor method that returns a `blessed` reference. What happens at the end of the story, when an object needs to be destructed? Object destruction happens in two possible cases, either implicitly or explicitly:

- Explicit destruction happens when no references to the object's data remains. Just like when dealing with ordinary references, you may have more than one reference to the data in existence. As we saw in Chapter 7, some of these references may be lexical variables, which go out of scope. As they do, the reference count of the data is decreased. Once it falls to zero, the data is removed from the system.
- Implicit destruction happens at the end of your program. At that point, all the data in your program is released.

When Perl needs to release data and destroy an object, whether implicitly or explicitly, it calls the method DESTROY on the object. Unlike other utility methods, this doesn't mean Perl is telling you what to do. Perl will destroy the data for you, but this is your chance to clean up anything else you have used, close any files you opened, shut down any network sockets, and so on. (Larry Wall joked that it should have been called something like YOU_ARE_ABOUT_TO_BE_SHOT_DO_YOU_HAVE_ANY_LAST_REQUESTS instead.)

If Perl doesn't find a method called DESTROY, it won't complain but will silently release the object's data. If you do provide a DESTROY method, be sure that it doesn't end up creating any more references to the data, because that's really naughty.

Our Finished Class

Let's put all the pieces of our class together finally and examine the class all the way through:

```
package Person;
```

First of all, let me reiterate that a class is nothing more than a package. We start off our class by starting a new package. As usual, we want to make sure this package is at least as pedantic as the one that called it, so we turn on warnings and strictness, and we load the Carp module to report errors from the caller's perspective.

```
# Class for storing data about a person
use warnings;
use strict;
use Carp;
```

Next we declare our class attributes. These are ordinary package variables, and there's nothing special about them:

```
# Class attributes
my @Everyone;
```

We provide a nice and general constructor, which calls a private method to do its private initialization. We take the class name, create a reference, and bless it.

```
# Constructor and initialisation
sub new {
   my $class = shift;
   my $self = {@_};
   bless($self, $class);
   $self->_init;
   return $self;
}
```

Our private method just adds a copy of the current object to a general pool. In more elaborate classes, we'd want to check that the user's input makes sense and get it into the format we want, open any external files we need, and so on.

```
sub _init {
   my $self = shift;
   push @Everyone, $self;
}
```

Next we provide very simple object accessor methods to allow us to get at the keys of the hash reference where our data is stored. These are the only interface we provide to the data inside the object, and everything goes through them.

```
# Object accessor methods
sub address  { $_[0]->{address }=$_[1] if defined $_[1]; $_[0]->{address } }
sub surname  { $_[0]->{surname }=$_[1] if defined $_[1]; $_[0]->{surname } }
sub forename { $_[0]->{forename}=$_[1] if defined $_[1]; $_[0]->{forename} }
sub phone_no { $_[0]->{phone_no}=$_[1] if defined $_[1]; $_[0]->{phone_no} }
sub occupation  {
   $_[0]->{occupation}=$_[1] if defined $_[1]; $_[0]->{occupation}
}
```

Accessing class attributes is even easier, since these are simple variables:

```
# Class accessor methods
sub headcount { scalar @Everyone }
sub everyone  { @Everyone        }
```

Finally, we have a couple of utility methods, which perform actions on the data in the object. The `fullname` method uses accessors to get at the forename and surname stored in the object and returns a string with them separated by a space:

```
# Utility methods
sub fullname {
   my $self = shift;
   return $self->forename." ".$self->surname;
}
```

Secondly, `printletter` is a slightly more elaborate method that prints out a letter to the referenced person. It uses the address and forename accessors, plus the `fullname` method to get the object's details. Notice that in both methods we're using `my $self = shift` to grab the object as it was passed to us.

```
sub printletter {
   my $self    = shift;
   my $name    = $self->fullname;
   my $address = $self->address;
   my $forename= $self->forename;
   my $body    = shift;
   my @date    = (localtime)[3,4,5];
   $date[1]++; # Months start at 0! Add one to humanise!
   $date[2]+=1900;  # Add 1900 to get current year.
   my $date    = join "/", @date;

   print <<EOF;
$name
$address

$date

Dear $forename,

$body

Yours faithfully,
EOF
}

1;
```

Inheritance

As we've gone along, we've taken certain steps to ensure that our class is suitable for having other classes inherit from it – but what exactly do we mean by this?

Inheritance, like the use of modules, is an efficient way of reusing code. If we want to build an `Employee` class, similar to the `Person` class but with additional attributes (`employer`, `position`, `salary`) and additional methods, (`hire`, `fire`, `raise`, `promote` and so on) inheritance means we don't have to write out all the code again. Instead, we simply specify the differences.

How does it work? It's simple– we tell perl the names of other classes to look in if it can't find a method. So, in our `Employee` class, all we need to write are the accessors for `employer`, `position`, `salary`, and the new methods. We then say 'in all other respects, we're like the Person class'. We don't need to write our constructor – when `Employee->new` is called, perl doesn't find a subroutine called new in our class, so it looks in `Person`, where, sure enough, it finds one. The same goes for all the other methods and accessors.

We can also inherit from multiple sources. To do this, we give a list of classes to look in. Perl will consult each class in order, using the first method it finds. We can also have a chain of inheritance – we can create an `Employee::Programmer` derived from the `Employee` class but with a fixed position and with additional methods (`$geek->read("userfriendly")`, `$geek->drink("cola")`, `$geek->hack("naked")`, and so on).

What Is It?

The fact that we can give a *list* of classes should give you an idea of how we do this – we use an array. Specifically, the package global `@ISA` is used to tell Perl what our class is derived from. I must admit that for a long time I pronounced it I-S-A, which didn't help me understand it one bit. If you haven't got it yet, read it as two words: 'is a'. For example, `@Employee::ISA = ("Person")`, or we could say `@Employee::Programmer::ISA = ("Employee")`, and grammar be damned.

Try It Out : Inheriting from a class

First of all, let's create a subclass that is exactly the same as `Person`. The 'empty subclass test' ensures that we can inherit from a class. Create a file `Employee1.pm`, and put the following in it:

```
package Employee;
#Employee1.pm
use Person9;
use warnings;
use strict;

our @ISA = qw(Person);
```

That's all we need to do to create a new class based on `Person`. We now have at our disposal all the methods that `Person` provides, and we can test this by changing our examples to use `Employee` instead of `Person`:

```
#!/usr/bin/perl
# inherit1.plx
use warnings;
use strict;
use Employee1;
```

```
my $object = Employee->new (
   surname    => "Galilei",
   forename   => "Galileo",
   address    => "9.81 Pisa Apts.",
   occupation => "bombadier"
);

$object->printletter("You owe me money. Please pay it.");
```

This does exactly the same as before.

How It Works

This is how our new class is constructed:

```
package Employee;
```

We provide a package declaration to start the class. Now, if we're going to bring in the `Person` class, we'd better make sure that the file that contains it is loaded. We ensure this by loading it:

```
use Person9;
```

Good habits dictate that we include these lines:

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

Finally, we come to the action – the package array `@ISA`, which does the magic:

```
our @ISA = qw(Person);
```

Perl looked for `Employee::new`. But since we didn't specify one, it used `Person::new` instead. The same goes for `Employee::printletter`. With this one line, we've reproduced the entire class, or rather, Perl has done it all for us, behind the scenes. Nice and easy.

Next we need to extend the class to provide our new methods.

Adding New Methods

At this stage, adding new methods is easy – just define them:

```
package Employee;
#Employee2.pm
use Person9;
use warnings;
use strict;

our @ISA = qw(Person);

sub employer { $_[0]->{employer}=$_[1] if defined $_[1]; $_[0]->{employer} }
sub position { $_[0]->{position}=$_[1] if defined $_[1]; $_[0]->{position} }
sub salary   { $_[0]->{salary  }=$_[1] if defined $_[1]; $_[0]->{salary  } }
```

```
sub raise {
   my $self = shift;
   my $newsalary = $self->salary + 2000;
   $self->salary($newsalary);
   return $self;
}
```

Now we can add and change these additional attributes:

```
#!/usr/bin/perl
# inherit2.plx
use warnings;
use strict;
use Employee2;

my $object = Employee->new (
   surname    => "Galilei",
   forename   => "Galileo",
   address    => "9.81 Pisa Apts.",
   occupation => "bombadier"
);

$object->salary("12000");
print "Initial salary: ", $object->salary, "\n";
print "Salary after raise: ", $object->raise->salary, "\n";
```

>**perl inherit2.plx**
Initial salary: 12000
Salary after raise: 14000
>

Overriding Methods

As well as adding new methods, we can provide our own version of the old ones. We certainly should provide our own version of `_init`, since that's a private method. In this case, we'll replace the employer with another `Person` object:

```
sub _init {
   my $self = shift;
   my $employer = $self->employer || "unknown";
   unless (ref $employer) {
      my $new_o = Person->new( surname => $employer );
      $self->employer($new_o);
   }
   $self->SUPER::_init();
}
```

Now when we create a new Employee object, the constructor will call `$self->_init`. This will now be found in our class, and the subroutine above will be run. What does it do?

```
sub _init {
```

As usual, we get the object we were passed:

```
my $self = shift;
```

From the object, we use the employer accessor we provided to extract the employer data from the object. If the user didn't provide an employer when calling the constructor, we use the word 'unknown':

```
my $employer = $self->employer || "unknown";
```

Now, the employer may already be an object, if so, we don't need to do anything else:

```
unless (ref $employer) {
```

Otherwise, we create a new object, and assign that as the current employee's employer:

```
   my $new_o = Person->new( surname => $employer );
   $self->employer($new_o);
}
```

Now for this wonderfully cryptic piece:

```
$self->SUPER::_init();
```

What on earth is `SUPER::`? Well, `SUPER::method` means 'call this method in the superclass'. In our case, once we've finished doing what we need to do, we tell Perl to call `Person::_init` so that the Person class can have a chance to arrange things as it likes.

Here's our complete employee class:

```
package Employee;
#Employee3.pm
use Person9;
use warnings;
use strict;
our @ISA = qw(Person);
sub employer { $_[0]->{employer}=$_[1] if defined $_[1]; $_[0]->{employer} }
sub position { $_[0]->{position}=$_[1] if defined $_[1]; $_[0]->{position} }
sub salary   { $_[0]->{salary  }=$_[1] if defined $_[1]; $_[0]->{salary  } }

sub raise {
   my $self = shift;
   my $newsalary = $self->salary + 2000;
   $self->salary($newsalary);
   return $self;
}

sub _init {
   my $self = shift;
   my $employer = $self->employer || "unknown";
   unless (ref $employer) {
      my $new_o = Person->new( surname => $employer );
      $self->employer($new_o);
   }
   $self->SUPER::_init();
}
```

Now let's see what we can do with it. We start off our new program as usual:

```
#!/usr/bin/perl
# inherit3.plx
use warnings;
use strict;
use Employee3;
```

We create a new Employee object, and a new employer:

```
my $dilbert = Employee->new (
   surname    => "Dilbert",
   employer   => "Dogbert",
   salary     => "43000"
);
```

This automatically creates a new Person as the employer, so we can now get at Dogbert and change *his* attributes:

```
my $boss = $dilbert->employer;
$boss->address("3724 Cubeville");
```

Of course, there's nothing to say that the employer **has** to be a Person object. There are always bigger fish:

```
my $dogbert = Employee->new (
   surname    => "Dogbert",
   employer   => "PHB",
   salary     => $dilbert->salary*2
);
$dilbert->employer($dogbert);
```

This creates a new employee object for Dogbert, with his boss recorded as 'the PHB'. We put this object in Dilbert's employer data. We can now get at the PHB in two ways:

```
my $phb = $dogbert->employer;
```

Or, starting at the bottom of the chain, since `$dilbert->employer` is `$dogbert`:

```
my $phb = $dilbert->employer->employer;
```

There's one class of which everything is a subclass, the `UNIVERSAL` *class, which is is the prime mover in Perl's OO world. You'll hardly ever use it, but it will provide you with some methods.* `isa($package)` *will return true if your class inherits from that package.* `can($method)` *is true if your class can perform the named method, and* `VERSION` *returns the value of the package variable $VERSION in your class, if one exists.*

Ties

As we've already mentioned, one particular thing that Perl gives us is the ability to hide an object behind a simple variable. What this means is that if we access such a variable, perl will call an object method behind our back; we can use this to do all sorts of magical things with our variables. This is called 'tying', and we say that the variable is **tied** to the class. When we looked at `Win32::TieRegistry`, the registry, a file on the disk, was accessed through an ordinary-looking hash, `%Registry`. To tie a variable to a class, we use the `tie` statement, like this:

```
tie $variable, 'Class', @parameters;
```

This you can tie any kind of variable – a scalar, an array, a hash, and even a filehandle. `tie` returns the object that we use to manipulate the variable, but we don't usually take much notice of that – the idea of `tie` is to keep the object hidden away from the program.

There's nothing special we need to do to our class to tell Perl we're going to tie to it. However, if we do tie a variable, perl expects to be able to call certain object and class methods. So, for instance, when you tie a scalar as above, perl will call the `TIESCALAR` method in the class `Class`. In fact, it'll do this:

```
$tied = Class->TIESCALAR(@parameters);
```

This should be an ordinary constructor, setting up whatever it is we're trying to do with this variable and returning an object. If the variable you're tying is an array, perl expects to call `TIEARRAY`. We also have `TIEHASH` and `TIEHANDLE` for hashes and filehandles, respectively. You don't need to provide all four constructors, just the one for the type of variable your class should be tied to. As a nice simple example, we'll create a 'counter', which increases in value every time you access it:

```
package Autoincrement;
#autoincrement.pm
use warnings;
use strict;

sub TIESCALAR {
   my $class = shift; # No parameters
   my $realdata = 0;
   return bless \$realdata, $class;
}

1;
```

We'll keep the real value of the counter in a scalar, and we'll have our object being a blessed reference to that scalar. We can now tie a scalar to this class:

```
#!/usr/bin/perl
# tiescalar.plx
use warnings;
use strict;
use Autoincrement;

my $count;
tie $count, 'Autoincrement';
```

The next thing we need to do is define what happens when we access the variable: Perl will call various object methods depending on what we do. For example, when we try and retrieve the value of the variable, by saying `print $count` or similar, Perl will try and call the `FETCH` method. Try it now – put this at the end of the program:

```
print $count;
```

You should see the following error:

```
Can't locate object method "FETCH" via package "Autoincrement" at tiescalar.plx line 9.
```

So, we need to provide a method called FETCH. What should it do? Well, in our case, we need to look inside the object reference and get out the real value of our counter, and we need to increase it by one. We can do these both at the same time, by adding this method:

```
sub FETCH {
   my $self = shift;
   return $$self++;
}
```

Notice that the value we return from our method is what we want our tied variable, `$count`, to produce. Now let's print it a few more times and see what happens:

```
#!/usr/bin/perl
# tiescalar.plx
use warnings;
use strict;
use Autoincrement;

my $count;
tie $count, 'Autoincrement';
print $count, "\n";
print $count, "\n";
print $count, "\n";
print $count, "\n";
```

You'll see the following:

```
>perl tiescalar.plx
0
1
2
3
>
```

Our `FETCH` method allows us to dictate what `$count` will produce, programmatically; in our case, it's producing a number that will increment each time we read it.

What else can we do with a scalar? Well, as well as fetch data from it, we can store data to it. The relevant method we need to provide for that is `STORE`. With an auto-incrementing variable, we'll fix things so that every attempt to store something to it winds us back to zero:

```
sub STORE {
   my $self = shift;
   $$self = 0;
}
```

Note that when we try to store something to `$count` now, it doesn't destroy the fact that it's a tied variable; the only way to remove the special treatment it gets is to untie the variable, like this:

```
untie $count;
```

So now we can write to our variable, and it'll just reset the counter to zero:

```
#!/usr/bin/perl
# tiescalar2.plx
use warnings;
use strict;
use Autoincrement;

my $count;
tie $count, 'Autoincrement';
print $count, "\n";
print $count, "\n";
print $count, "\n";
print $count, "\n";
$count = "Bye bye!";
print $count, "\n";
print $count, "\n";
print $count, "\n";
print $count, "\n";
```

>**perl tiescalar2.plx**
0
1
2
3
0
1
2
3
>

In this case, we didn't do anything with the value the program tried to store (we ignored the `"Bye Bye!"`). However, if we want to get hold of the value, it's passed as a parameter to our method:

```
sub STORE {
   my $self  = shift;
   my $value = shift;
   warn "Hi, you said $value\n";
   $$self = 0;
}
```

will now cause our program to print:

```
>perl tiescalar2.plx
0
1
2
3
Hi, you said Bye, bye!
0
1
2
3
>
```

This is all we need to do to control the treatment of a scalar, because it's all a scalar can do – initialize itself, retrieve a scalar value, and store a scalar value. Arrays, hashes, and filehandles are trickier, but let's quickly recap what we've provided:

`TIESCALAR(@parameters)`	Construct an object ready to be used for a tied scalar.
`FETCH()`	Retrieve the value of a tied scalar, however we wish to do so.
`STORE($value)`	Actions to perform when `$value` is stored into our tied scalar.

For an array, there are some extra methods you need to provide:

`TIEARRAY(@parameters)`	Construct an object ready to be used for a tied scalar.
`FETCH($element)`	Fetch the `$element`'th element of the array.
`STORE($element, $value)`	Store `$value` in the `$element`'th element of the array.
`FETCHSIZE()`	Provide the size of the array, for when the user says `$#tied_array` or `scalar @tied_array`.
`STORESIZE($size)`	Store the size, for when the user says `$#tied_array=$size` or similar.

If you're implementing tied arrays, it's recommended that you inherit from the class `Tie::StdArray` that lives in the standard `Tie::Array` package. `Tie::StdArray` provides you with a tied equivalent for a standard Perl array, and you should overload whichever methods you need to customize it for your purposes.

Likewise, ties can be inherited from the `Tie::StdHash` package in `Tie::Hash`. This will expect your underlying object to be a blessed hash, and provide `EXISTS` and `DELETE` which are called by the Perl operators of the same name, as well as the key iterators `FIRSTKEY` and `NEXTKEY` which are used when the user calls `keys` or `each` on the tied hash. These iterators are not things you want to write yourself, and it's far easier to inherit from Perl if possible.

To close, here's a very, very simple way of making a hash persistent that is, the data in the hash will be stored on disk when your program finishes so you can use it again next time.

Try It Out : Simple Persistent Tied Hashes

We'll store a hash on disk in the following format:

```
Key:value
Key:value
...
```

We know how to read this in – we'll just split each line on a colon, like this:

```
if ( -e $file ) {
   open FH, $file or die $!;
   while (<FH>) {
      chomp;
      my ($k, $v)   = split /:/,$_,2;
      $realhash{$k} = $v;
   }
}
```

Let's turn this into our constructor:

```
package PersistHash;
#persisthash.pm
use strict;
use warnings;
use Tie::Hash;
our @ISA = qw(Tie::StdHash);
sub TIEHASH {
   my $class = shift;
   my %realhash;
   my $file = shift;
   if ( -e $file ) {
      open FH, $file or die $!;
      while (<FH>) {
         chomp;
         my ($k, $v)   = split /:/,$_,2;
         $realhash{$k} = $v;
      }
   } # Otherwise we'll create it when we're done.
   $realhash{__secret__} = $file; # Need to stash this for when we write.
   return bless \%realhash, $class;
}
```

Now we have something that operates just like a standard Perl hash, with the exception that we read the data from a file when the hash is initially tied. We can say:

```
tie %dictionary, "PersistHash", "mydict.txt";
```

and the data will be loaded from the file. So far we can manipulate and change this data – but we've no way of writing it back when we're finished. The hash can go out of scope, become untied, or reach the end of the program, and we won't have written its contents to disk.

In each of these cases though, the destructor will be called to clear up the object – and that's exactly when we need to save the data back to the disk. This is why we have destructors!

```
sub DESTROY {
   my $self = shift;
   my %realhash = %$self;
   my $file = $realhash{__secret__}; # Extract the filename we stashed.
   delete $realhash{__secret__}; # Don't want that written to disk.
   open FH, ">$file" or die $!;
   for (keys %realhash) {
      print FH $_, ":", $realhash{$_}, "\n";
   }
   close FH;
}
```

There we are – we can load the data in, we can write it back out again, and with the exception of providing the file name, this all takes places without the user knowing they're dealing with an object. We've provided what looks like a normal hash, but we've hidden away the persistence.

Of course, this is a very simplified implementation; reading in a line at a time means we'll have problems if the keys or values the user wants to store contain new lines or if they contain colons. Storing references and objects in the hash won't work at all, so it's impossible to hold deep data structures. Worst of all, we store the filename inside the hash itself, which feels like a messy solution; it's unlikely that anyone would want to get at the value `__secret__`, but unlikely things do happen, particularly in programming.

Ideally, we'd separate that sort of data from the hash, and we'd also ensure that the data would get written in such a way that neither newlines nor colons, nor indeed anything we choose to store in our hash, will break the storage. Thankfully, this has already been done, and we'll look at the module that implements it in the next chapter.

Summary

Object-oriented programming is another way of thinking about programming. You approach it in terms of data and the relationships between pieces of data, which we call objects. These objects belong to divisions called classes – these have properties (**attributes**) and can perform activities (**methods**).

Perl makes object-oriented programming neat and simple:

- An **object** is a reference that has been blessed into a class.
- A **class** is an ordinary Perl package.
- A **method** is an ordinary Perl subroutine.

From these three basic principles, we can start to build data-driven applications. We've seen how easy it is to apply **inheritance**, deriving a more specific class (a **subclass**) from a more general class (a **superclass**) by merely specifying what's different.

Exercises

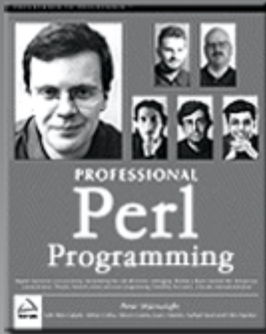
1. Give Dogbert a phone number, a position, and a value for anything else that's undefined. Make sure that you can see exactly what is happening in the code by printing out values at each stage in inherit3.plx.

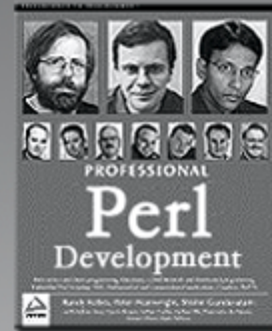
2. Create a new object method, to print out a business card. It must print out all the pertinent information of the chosen employee.

Source code available at : www.wrox.com

Peer discussion at : lamplists.com

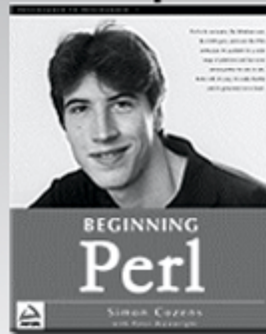
Also from Wrox





http://www.wrox.com/books/1861004494.htm

http://www.wrox.com/books/1861004389.htm



http://www.wrox.com/books/1861003145.htm

lamplists.com
The Open Source Programmer's Resource Centre

This work is licensed under the Creative Commons **Attribution-NoDerivs-NonCommercial** License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nd-nc/1.0 or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

The key terms of this license are:

Attribution: The licensor permits others to copy, distribute, display, and perform the work. In return, licensees must give the original author credit.

No Derivative Works: The licensor permits others to copy, distribute, display and perform only unaltered copies of the work -- not derivative works based on it.

Noncommercial: The licensor permits others to copy, distribute, display, and perform the work. In return, licensees may not use the work for commercial purposes -- unless they get the licensor's permission.