Object Oriented Perl

```
#!/usr/bin/perl
# Copyright (c) Marcus Post, <marcus@marcuspost.com>
                #               #               #               #
$_=q,my(@f|@c|x$_=q.my(@f|@c|x$_=q.my(@f|@c|x$_=q.m(@f||@c|x$_=q.m(@f||@c|xx
@w);@a=@f=<DAT%@w);@a=@f=<DAT%@w);@a=@f=<DAT%@w;@a=@f=<DAAT%@w;@a=@f=<DAAT%%
A>;seek(DATA|0!A>;seek(DAA|0!!A>;seek(DAA|0!A>;seek(DAA|0!!A>;seek(DAA|0!!AA
|0);@c=<DATA>;Y|0);@c<DATA>;Y||0);@c<DATA>Y||0);@c<DATA>Y|||0);@c<DATA>Y||||
until(($_=pop(zutil(($_==pp(zuttil(($_==p(zuttil(($_==p(zutttil(($_==p(zuttt
@c))=~/^_/){};Qc))=~/^_/){};Qc)))=~/^_/{};Qc)))=~/^_/{};Qc))))=~/^_/{};Qc)))
unshift(@a|$_)xnshift(@a|$_)xnshhift(a|$_)xnshhift(a|$_)xnshhiift(a|$_)xnshh
;for(1..3){pri%;for(1.3){pri%;ffor1.3){pri%;ffor1.3){pri%;ffor11.3){pri%;fff
nt(shift(@c));!nt(shft(@c));!ntt(hft(@c));!ntt(hft(@c));!ntt(hftt(@c));!nttt
}for(@f){my($sY}for@f){my($sY}for@f){my($sY}for@f){my($sY}for@f){mmy($sY}foo
);split//;$_=sz);splt//;$_=sz);splt//;$_=sz);splt//;$_=sz);splt//;$_==sz);ss
hift(@c);$_=~sQhift(c);$_=~sQhift(c);$_=~sQhift(c);$_=~sQhift(c);$_=~sQQhiff
/(.{15}).*/\1/x/(.{15})*/\1/x/(.{15})*/\1/x/(.{15})*/\1/x/(.{15}})*\1/xx/(..
;@w=split//;fo%;@w=split/;fo%;@w=split/;fo%;@w=split/;fo%;@w=spllit;fo%%;@ww
r(@_){$w[$s+15!r(@_){$w[$s15!r(@_){$w[$s15!r(@_){$w[$s15!!(@_){$$w[s15!!!(@@
-$_]=(($w[$s]eY-$_]=(($w[$s]YY-_]=(($w[$s]YY-_]=(($w[$s]YY-_]=((($[$s]]YY-__
q"|")?".":$w[$zq"|")?".":$w[$zq|")?"."::$[$zq|")??."::$[[$z|")???.::$$[[$z||
s]);$s++;}for(Qs]);$s++;}for(Qs];$s++;}}or(Qs];$$s+;}}orr(Qs]$$s++;}}orr(Qss
1..75){unless(x1..75){unless(x1.75){unnlss(x1.775){uulsss(x1.75){uuulsss(x11
$w[$_]ne’’){$w%$w[$_]ne’’){$w%$w$_]nee’’{$w%$$w$_]nn’’{{$w%$$w_]nnn’’{{$w%$$
[$_]=$w[($_-1)![$_]=$w[($_-1)![$_=$w[[($_-)![$_==w[[($$_-)![$__w[[[($$_-)![[
];}}print(joinY];}}print(joinY];}prinnt(joinY;}prinntt(joinY;}pinnntt(joinYY
""|@w);print"\z""|@w);print"\z""|w);;print"\z"|w);;pprint"\z"|w;;ppprint"\zz
n";}print@a;,;#n";}print@a;.;#n";priint@a;.;#n;priintt@a;.;#n;piinntt@a;.;##
y!|zY\!%x!,Q!;#y!|zY\!%x!.Q!;#y!zY\!!%x!.Q!;#!zY\!!%x!!.Q!;#!z\!!!%x!!.Q!;##
s{Q.*\n}[]g;#<>s{Q.*\n}[]g;#<>sQ.*\nn}[]g;#>sQ.*\nn}[]]g;#>sQ.\nnn}[]]g;#>ss
eval;#EndFini!$eval;#EndFini!$eal;#EEndFin!$eal;;##nddFin!$ea;;###nddFin!$ee
```


Paul Fenwick
Jacinta Richardson

Object Oriented Perl
by Paul Fenwick and Jacinta Richardson

Conventions used throughout this text are based upon the conventions used in the Netizen training manuals by Kirrily Robert, and found at http://sourceforge.net/projects/spork

This training manual is maintained by Perl Training Australia, and can be found at http://www.perltraining.com.au/notes.html.

This is revision 1.4 of Perl Training Australia's "Object Oriented Perl" training manual.

Table of Contents

List of Figures

Chapter 1. Introduction

Welcome to Perl Training Australia's Object Oriented Perl training course. This is a two-day module in which we will cover object oriented programming concepts in Perl.

Course outline

- Object oriented refresher
- What are packaged and modules
- How to write packages and modules
- A first Perl object
- Using this knowledge
- Passing arguments by name
- Class methods and variables
- Destructors
- Inheritance
- Redispatching method calls
- Abstract classes
- Polymorphism
- Using this knowledge
- Operator overloading

Assumed knowledge

This training module assumes the following prior knowledge and skills:

- Thorough understanding of operators and functions, conditional constructs, subroutines and basic regular expressions in Perl.
- Thorough understanding of arrays, scalars and hashes in Perl.
- Thorough understanding of references and complex data structures in Perl.

Module objectives

- Understand basic concepts of object oriented programming in Perl.
- Understand how to write and use modules and packages.
- Be able to write basic classes and class methods.
- Understand how and when to write destructor functions.

- Understand inheritance and multiple inheritance and how to handle the issues these create.
- Be able to use the NEXT pseudo-class to assist in cases of multiple inheritance.
- Understand polymorphism.
- Understand and be able to overload operators in useful manners.

Platform and version details

This module is taught using Unix or a Unix-like operating system. Most of what is learnt will work equally well on Windows NT or other operating systems; your instructor will inform you throughout the course of any areas which differ.

All Perl Training Australia's Perl training courses use Perl 5, the most recent major release of the Perl language. Perl 5 differs significantly from previous versions of Perl, so you will need a Perl 5 interpreter to use what you have learnt. However, older Perl programs should work fine under Perl 5.

At the time of writing, the most recent stable release of Perl is version 5.8.4, however older versions of Perl 5 are still common. Your instructor will inform you of any features which may not exist in older versions.

The course notes

These course notes contain material which will guide you through the topics listed above, as well as appendices containing other useful information.

The following typographical conventions are used in these notes:

System commands appear in **this typeface**

Literal text which you should type in to the command line or editor appears as `monospaced font`.

Keystrokes which you should type appear like this: **ENTER**. Combinations of keys appear like this: **CTRL-D**

```
Program listings and other literal listings of what appears on the
screen appear in a monospaced font like this.
```

Parts of commands or other literal text which should be replaced by your own specific values appears *`like this`*

Notes and tips appear offset from the text like this.

Notes which are marked "Advanced" are for those who are racing ahead or who already have some knowledge of the topic at hand. The information contained in these notes is not essential to your understanding of the topic, but may be of interest to those who want to extend their knowledge.

Notes marked with "Readme" are pointers to more information which can be found in your textbook or in online documentation such as manual pages or websites.

Notes marked "Caution" contain details of unexpected behaviour or traps for the unwary.

Other materials

In addition to these notes, it is highly recommend that you obtain a copy of Programming Perl (2nd or 3rd edition) by Larry Wall, et al., more commonly referred to as "the Camel book". While these notes have been developed to be useful in their own right, the Camel book covers an extensive range of topics not covered in this course, and discusses the concepts covered in these notes in much more detail. The Camel Book is considered to be the definitive reference book for the Perl programming language.

The page references in these notes refer to the *3rd edition* of the camel book. References to the 2nd edition will be shown in parentheses.

An essential book on object oriented programming in Perl is Damian Conway's "Object Oriented Perl". This book is referenced through-out the text.

Chapter 2. An object oriented refresher

In this chapter...

In this section we provide a quick refresher or lesson on basic object oriented concepts.

Object orientation in brief

This course does not aim to teach you all aspects of Object Oriented (OO) Programming, if we were to do that it would leave precious little time to cover the aspects of the language (Perl) we wish to implement it in. Rather, this course assumes that you already know the basics of OO, or are willing to learn them rather quickly.

Damian Conway wrote in his book *Object Oriented Perl* the following on the essentials of Object Orientation (used with permission):

> You really need to remember only five things to understand 90 percent of the theory of object orientation:
>
> - An *object* is anything that provides a way to locate, access, modify, and secure data;
> - A *class* is a description of what data is accessible through a particular kind of object, and how that data may be accessed;
> - A *method* is the means by which an object's data is accessed, modified or processed;
> - *Inheritance* is the way in which existing classes of objects can be upgraded to provide additional data or methods;
> - *Polymorphism* is the way that distinct objects can respond differently to the same message, depending upon the class to which they belong.

Conway's book is an excellent and enjoyable read, and a superb reference for all aspects of object oriented programming in Perl. In fact, it's so good that we'll refer to it extensively throughout these notes. After you complete this course you'll find these notes are greatly enhanced if you have a copy of Conway's book to refer to as well.

Objects and methods

Put simply, an *object* is a way of accessing data. The data it allows you to access are usually referred to as *attributes*. The thing that makes attributes special is that they're associated exclusively with a given object.

Now, if that's all objects are, then we could say that a hash, array or scalar are objects. However, while all these things can be turned into objects, they're not objects in their own right. That's because one of the cornerstones of object oriented programming is that attributes are not accessible to the entire program. In fact, you should only access them through special subroutines associated with the object. These subroutines are referred to as *methods*, and they're usually accessible to anyone who can use your object.

Methods are very important, as they can be used to restrict the ways in which an object's attributes can be modified or accessed. A method which sets the date, for example, might forbid any attempt to set the date to the 31st of February. Methods are also important because they allow the internal representation of objects to change. Provided that the way of calling the method remains the same, it doesn't matter if an object changes its internal date representation from seconds from 1st January 1970 (often called "seconds from the epoch" or "Unix timestamp"), to using three integers containing the year, month and day.

Objects are so named because there are many analogies to real-world objects, so an example here should help make things more clear. Let's consider the humble drinks machine.

A drinks machine has a number of attributes; the amount and type of coins with which to give change, inventories of the various drinks available, the cost of each drink, the current internal temperature, whether or not the refrigeration unit is operating, and so on. People don't have direct access to those attributes, instead they're restricted by the buttons and coin slots on the machine. This *interface* is designed to ensure that only certain operations may be performed so that the machine maintains a consistent internal state. For example, the owners of the drinks machine only want to dispense a drink if an appropriate amount of money has been inserted.

The restrictions aren't just in the interest of the machine's owner, some of them are to help the customer as well. By maintaining a consistent state it's possible to ensure that customers get both the drink they asked for as well as correct change. Some restrictions (like the machine being bolted to the floor) can stop potentially dangerous operations, like people trying to rock the machine. Other restrictions help ensure that the internal temperature setting can't be changed and spoil the drinks for others.

What we'll now investigate is how we set up the association between an object and its interface and attributes.

Classes

A *class* provides a set of methods which become associated with a particular kind of object. A class also provides a specification of the attributes to be used as well. It's effectively a blueprint, describing what the object is to look like and how it will act.

When a program needs an object of a particular type, it calls upon this blueprint along with some information about the initial state of the object (like the advertising to put on the front of the drink machine, or which drinks it has to start with). The blueprint (class) makes sure these initial values are sensible, manufactures the appropriate object, and returns it.

When we call a method on an object (such as `give_change`), the class definition is consulted again to ensure that's a valid method for the object and has been called correctly. If it is, that method is invoked and does its thing. If it isn't, then an error or exception is usually raised.

A common mistake in object oriented programming is to forget the distinction between objects and their classes. The class is the description of the object and the object in an instance of the class. For example the class of humans would describe us as having the attributes such as arms, hands, legs and heads; and methods such as talk, think, eat, and sit. However each of us would be an instance of that class, an object. If I can jump it's because humans can jump, if I can laugh, it's because humans can laugh. That is, the class defines the methods and attributes for each object belonging to that class.

Inheritance

Let's say that we want to build a new type of drink vending machine, one that not only accepts small change, but also accepts credit cards as well. We wouldn't want to start from scratch, designing an entirely new refrigeration unit, cash dispenser, can holder, and so forth. Instead, we'd start with our existing blueprint for a standard drinks machine, and modify it appropriately to our needs.

The idea of taking an existing class and extending it to add new functionality is highly encouraged in object oriented programming. In this case we'd say that our new drinks machine is derived from, or *inherits*, the existing `DrinksMachine` class. In this case we say that the `DrinksMachine` is the *parent* or *base* class, and the `DrinksMachine::Credit` is the *child* or *derived* class.

Once we've stated that `DrinksMachine::Credit` inherits the behaviour of `DrinksMachine`, we're free to make changes to extend our new class as we see fit. For example, we might add a `swipe_card` method, and redefine `request_drink` to allow multiple drinks to be purchased in a single transaction. Except for these changes, our new drinks machine operates just like the old one.

Parent and child classes are related by an "*is a*" relationship. A credit-card drinks machine *is a* drinks machine, and a grandfather-clock *is a* clock. Inheritance extends not just to parents, but to grandparents and great-grandparents and so on as well. So a credit card drinks machine *is a* drinks machine *is a* vending machine *is a* machine. The further up the ancestry we go, the more generalised things become.

We should note that there's a difference between "*is a*" relationships and "*has a*" relationships. A car *has a* steering wheel (which may be a class unto itself that inherits from a steering device class), but it is not a steering wheel. A car *has a* front light (or two) but the car is not a front light. On the other hand, the car may inherit from the vehicle class and hence we'd say the car *is a* vehicle. It's usually fairly straight forward to determine which is the correct relationship.

We'll explain the use of the `::` (double-colon) in the next chapter.

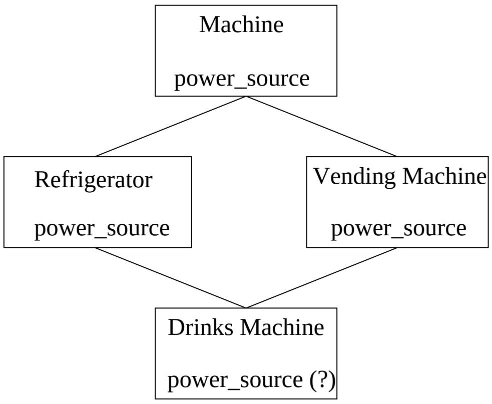
Multiple inheritance

Sometimes we'll want to create a class that requires the behaviour of two or more different classes, and we'd like to inherit from both of them. This situation is called multiple inheritance, and is often very useful. For example, our `DrinksMachine` class might inherit from both the `Refrigerator` and `VendingMachine` classes, gaining the behaviour and capabilities of both.

Multiple inheritance is not without its pitfalls. There may be cases when method calls become ambiguous. If I have a person who inherits from both `TruckDriver` and `Golfer`, which method should be used as a request to `drive`?

Another problem occurs when a class has two or more parents which share a common ancestor class. Say that both `Refrigerator` and `VendingMachine` both inherit from the `Machine` class. Should our `DrinksMachine` receive the `power_source` attribute twice, or should it be merged together because they're both inherited from the same source?

Figure 2-1. The DrinksMachine inheritance tree



There are ways to solve all these problems, although different languages take different approaches. For example, we might require that ambiguous methods be renamed, or we could mark (perhaps arbitrarily) one method to have priority over another. We can do similar things with attributes. We'll explain how Perl solves these problems later in the course.

Polymorphism

Polymorphism is the ability for objects to respond differently to the same message, depending upon what type of object they are. For example, members from each of the following classes; `Spouse`, `YoungerBrother`, `TotalStranger` or `LawEnforcementOfficer`, are likely to behave differently when the `hug` method is called upon them.

Polymorphism becomes very useful when we have a group of related objects upon which we want to perform an operation, but those objects may need to react in different ways. For example, an `ElectricCar` will need to react differently to a `FossilFuelCar` when asked to `accelerate`.

There are two distinct forms of polymorphism in object oriented programming; *interface* and *inheritance* polymorphism. Interface polymorphism is where two or more unrelated classes provide the same interface to certain methods. For example both sparrows and aeroplanes can fly. Although sparrows and aeroplanes fly in completely different ways, if we implement our classes in such a way that the methods have the same arguments and argument order then we have an example of interface polymorphism.

Inheritance polymorphism is where a child class inherits methods from an ancestor. Hence if the `Machine` class, mentioned above, implemented a `turn_on` method then the `DrinksMachine` class would inherit that method. If we were to call the `turn_on` method on a `DrinksMachine` object the `DrinksMachine` object would behave as if it were merely a `Machine` object for that method call.

Exercise

Imagine the following situation. Software is to be written to handle information about the aircraft housed at a particular airport. There are various kinds of these aircraft and these fall into three categories: personal, elite and passenger. Personal and elite aircraft are privately owned and the airport keeps information about the owner's name and contact details. Personal aircraft are never piloted by the airport's pilots. Elite aircraft also have a V.I.P. associated with them. Elite aircraft are usually owned by companies but usually use the airport's pilots. Passenger aircraft are owned by the airport and have a regular route with predetermined destinations and only use the airport's pilots.

All aircrafts have fuel quantities, hanger numbers, a maximum person carrying capacity as well as luggage and cargo, a maximum flying distance and several other values.

1. What classes can you identify in this description?
2. Draw these classes and their relations to each other. Can you identify any places where inheritance might be useful?
3. With each class list any methods and attributes you can think of that belong to that class.
4. Can we make use of inheritance polymorphism to reduce code duplication? Mark any methods you've included in child classes that can be inherited in total from an ancestor class.

Chapter summary

- An object is anything that provides a way to locate, access, modify and secure data.
- A class is a description of what data is accessible through a particular kind of object and how that data may be accessed.
- A method is the means by which an object's data is accessed, modified or processed.
- Inheritance is the way in which existing classes of objects can be upgraded to provide additional data or methods.
- Multiple inheritance is where a class of objects inherit from more than one super/parent class.
- Polymorphism is the way that distinct objects can respond differently to the same message depending upon the class to which they belong.

Chapter 3. Modules and packages

In this chapter...

In this chapter we'll discuss modules from a user's standpoint. We'll find out what a module is, how they are named, and how to use them in our work.

Module uses

Perl modules can do just about anything. In general, however, there are three main uses for modules:

- Changing how the rest of your program is interpreted. For example, to enforce good coding practices (`use strict`) or to allow you to write in other languages, such as Latin (`use Lingua::Romana::Perligata`), or to provide new language features (`use Switch`).
- To provide extra functions to do your work (`use Carp` or `use CGI qw/:standard/`).
- To make available new classes (`use HTML::Template` or `use Finance::Quote`) for object oriented programming.

Sometimes the boundaries are a little blurred. For example, the `CGI` module provides both a class and the option of extra subroutines, depending upon how you load it.

What is a module?

A module is a separate file containing Perl source code, which is loaded and executed at compile time. This means that when you write:

```
use CGI;
```

Perl looks for a file called `CGI.pm` (.pm for *Perl Module*), and upon finding it, loads it in and executes the code inside it, before the looks at the rest of your program.

Sometimes you need to tell Perl where to look for your Perl modules, especially if some of them are installed in a non-standard place. Like many things in Perl, There's More Than One Way To Do It. Check out **perldoc -q library** for some of the ways to tell Perl where your modules are installed.

Sometimes you might choose to pass extra information to the module when you load it. Often this is to request the module create new subroutines in your namespace.

```
use CGI qw/:standard/;
```

Note the use of `qw//`, this is a list of words (in our case, just a single word). It's possible to pass many options to a module when you load it. In the case above, we're asking the `CGI` module for the `:standard` bundle of functions.

Each module has a different set of options (if any) that it will accept. You need to check the documentation of the module you're dealing with to which (if any) are applicable to your needs.

To find out what options exist on any given module read its documentation: **perldoc** ***module_name***.

Where does Perl look for modules?

Perl searches through a list of directories that are determined when the Perl interpretor is compiled. You can see this list (and all the other options Perl was compiled with), by using **perl -V**.

The list of directories which Perl searches for modules is stored in the special variable `@INC`. It's possible to change `@INC` so that Perl will search in other directories as well. This is important if you have installed your own private copy of some modules.

Of course, being Perl, there's more than one way to change `@INC`. Here are some of the ways to add to the list of directories inside `@INC`:

- Call Perl with the `-I` command-line switch with the location of the extra directory to search. For example:

  ```
  perl -I/path/to/libs
  ```

  This can be done either in the shebang line, or on the command-line.
- Use the `lib` pragma in your script to inform Perl of extra directories. For example:

  ```
  use lib "/path/to/libs";
  ```
- Setting the `PERL5LIB` environment variable with a colon-separated list of directories to search. Note that if your script is running with taint checks this environment variable is ignored.

Since `use` statements occur before regular Perl code is executed, modifying `@INC` directly usually does not have the desired effect.

Finding installed modules

Perl comes with many modules in its standard distribution. You can get a list of all of them by doing a **perldoc perlmodlib**. The Camel book describes the standard modules in chapters 31 and 32 (chapter 7, 2nd Ed).

Besides from the modules in the standard distribution, you can also see any other modules that were installed on your system by using **perldoc perllocal**.

You can get more information on any module that you have installed by using **perldoc** ***module_name***. For example, **perldoc English** will give you information about the `English` module.

Most importantly, there's a great resource for finding modules called the *Comprehensive Perl Archive Network*, or *CPAN* for short. The CPAN website (http://www.cpan.org/) provides many ways of finding the modules you're after and browsing their documentation on-line. It's highly recommended that you become familiar with CPAN's search features, as many common problems have been solved and placed in CPAN modules.

Exercise

1. Open a web browser to CPAN's search site (http://search.cpan.org) and spend a few minutes browsing the categories provided.
2. Perform a search on CPAN for a problem domain of your choice. If you can't think of one, search on `CGI`, `XML` or `SOAP`.

Using CPAN modules

At the time of writing, CPAN provides more than 5,500 separate and freely available modules. This makes CPAN an excellent starting point when you wish to find modules to help solve your particular problem. However, you should keep in mind that not all CPAN modules are created equal. Some are much better documented and written than others. Some (such as the `CGI` or `DBI`) modules have become de-facto standards, whereas others may not be used by anyone except the module's author.

As with any situation when you're using third party code, you should take the time to determine the suitability of any given module for the task at hand. However, in almost all circumstances it's better to use or extend a suitable module from CPAN rather than trying to re-invent the wheel.

Many of the popular CPAN modules are pre-packaged for popular operating systems. In addition, the `CPAN` module that comes with Perl can make the task of finding and installing modules from CPAN much easier.

Most CPAN modules come with `README` and/or `INSTALL` files which tell you how to install the modules. However in almost every case, the process is the same:

```
perl Makefile.PL
make
make test
make install
```

If you install your module in a different directory than your other Perl modules you may have to use the `lib` pragma, mentioned in the previous section, to tell Perl where to find your files. Once a module is installed, you can use it just like any other Perl module.

The double-colon

Sometimes you'll see modules with double-colons in their names, like `Finance::Quote`, `Quantum::Superposition`, or `CGI::Fast`. The double-colon is a way of grouping similar modules together, in much the way that we use directories to group together similar files. You can think of everything before the double-colon as the category that the module fits into.

In fact, the file analogy is so true-to-life that when Perl searches for a module, it converts all double-colons to your directory separator and then looks for that when trying to find the appropriate file to load. So `Finance::Quote` looks for a file named `Quote.pm` in a directory called `Finance`. That two modules are in the same category doesn't necessarily mean that they're related in any way. For example, `Finance::Quote` and `Finance::QuoteHist` have very similar names, and their maintainers even enjoy very similar hobbies, they certainly have similar uses, but neither package shares any code in common with the other.

It's perfectly legal to have many double-colon separators in module names, so `Chicken::Bantam::SoftFeather::Pekin` is a perfectly valid module name.

What is a package?

A package is simply a namespace, where you can use variables, subroutines, and filehandles, without the fear of clashing with identically named items from other name spaces. It's possible for a file to contain multiple packages, and packages may also stretch across multiple files. Since a module is just a file with a special name, this means that modules can contain multiple packages, too.

Poking inside packages

Modules usually put their code, variables, and other things to do their jobs into their own namespace, or *package*. It's very rare that you should ever need to poke around inside someone else's package, and if you do so and break things, then you only have yourself to blame. However, in case you ever have a very good reason, you can get into someone else's namespace using the scoping operator, `::`.

Yes, that's the same operator we saw before with module names. If it helps, you can think that modules can sit inside the namespace of other modules, in the same way that directories can sit inside other directories. Here's how we use the scoping operator to access variables inside a package:

```
use Carp;

# Turning on $Carp::Verbose makes carp() and croak() provide
# stack traces, making them identical to cluck() and confess()

$Carp::Verbose = 1;
```

When referring to a variable in another package, the sigil (punctuation denoting the variable type) always goes *before* the package name. Hence to get to the scalar `$bar` in the package `Foo`, we would write `$Foo::bar` and not `Foo::$bar`.

It is not possible to access lexically scoped variables (those created with `my`) in this way. Lexically scoped variables can *only* be accessed from their enclosing block.

We can call a subroutine inside another package in much the same way. For example, we can call `Carp`'s `cluck` subroutine which isn't usually exported like this:

```
use Carp;

sub verify {
        my $i = shift;

        # Call Carp’s cluck subroutine directly.
        Carp::cluck("Bad data") unless $i > 10;
}
```

Calling subroutines like this is a perfectly acceptable alternative to exporting them into your own namespace. You might want to do this if you already define a subroutine of the same name in your own package.

Unlike accessing another package's variables, which you should never do without good reason, most modules make their subroutines available so that you *can* call them directly if you desire.

Package variables and our

It is not possible to access lexically scoped variables (those created with `my`) outside of their enclosing block. This means that we need another way to create variables to make them global. These global variables are called *package variables*. The preferred way to do this, under Perl 5.6.0 and above, is to declare them with `our`. Of course, there are alternatives you can use with older version of Perl, which we also show here:

```
package Carp;

our $Version = '1.01';          # Preferred for Perl 5.6.0 and above

# use vars qw/$Version/;        # Preferred for older versions
# $Version = '1.01';

# $Carp::Version = '1.01';      # Acceptable but requires that we then
                                # always use this full name under strict
```

In all of the cases above, both our package and external code can access the variable using `$Carp::Version`.

Exercises

1. Look at the documentation for the `Carp` module using the **perldoc Carp** command. This is one of Perl's most frequently used modules.
2. Using `File::Copy` make a copy of one of your files. If you're eager, ask the user which file to copy and what to name the copy. Don't forget to clean the user's input.

Chapter summary

- A module is a separate file containing Perl source code.
- We can use modules by writing `use` *`module_name`*`;` before we want to start using it.
- Perl looks for modules in a list of directories that are determined when the Perl interpretor is compiled.
- Module names may contain double-colons (`::`) in their names such as `Finance::Quote`, these tell where Perl to look for a module (in this case in the `Finance/` directory.
- Modules can be used for class definitions or as libraries for common code.
- A package is a namespace within a module

- A module can have more than one package defined within it.
- We can get to subroutines and variables within packages by using the double-colon as a scoping operator for example `Foo::bar()` calls the `bar()` subroutine from the `Foo`

Chapter 4. Writing packages and modules

In this chapter...

This chapter will teach you the basics of writing your own packages and modules.

Writing packages

As we mentioned in the previous chapter, Perl provides the notation of separate name spaces, so that unrelated sections of code can work in their own spaces and not have to worry about clobbering someone else's variables or subroutines. These name spaces are also known as packages, and it's possible to move between them using the `package` statement:

```
package Foo;

sub hello { print "Hello FooWorld\n"; }
hello();

package Bar;
sub hello { print "Hello BarWorld\n"; }

hello();        # Implies Bar::hello(), since Bar is our current package.
Foo::hello();   # Explicitly calls Foo::hello()
```

Here we've created two entirely separate subroutines -- both of which are named `hello`, but which exist in different packages. By default, whenever we refer to a subroutine or variable, we refer to the one in our current package. We can also explicitly prefix a package name, followed by the double-colon operator, to get at things in another package, as we did with `Foo::hello` above.

The package that you're in when the Perl interpreter starts (before you specify any package) is called `main`. Package declarations use the same rules as `my`, that is, it lasts until the end of the enclosing block, file, or eval. Here's an example:

```
#!/usr/bin/perl -w
use strict;

sub hello { print "This is hello in the main package\n"; }

{
        package Foo;
        sub hello { print "This is hello in the Foo package\n"; }
}

# Here we're back in the main package again.

hello();        # main's hello
Foo::hello();   # Foo's hello
```

Perl convention states that package names (or each part of a package name, if it contains many parts) starts with a capital letter. Packages starting with lower-case are reserved for pragmas (such as `strict`).

There's a shorthand for accessing variables and subroutines in the `main` package, which is to use double-colon without a package name. This means that `$::foo` is the same as `$main::foo`.

Writing modules

Modules contain regular Perl code, and for most modules the vast majority of that code is in subroutines. Sometimes there are a few statements which initialise variables and other things before any of those subroutines are called, and those get executed immediately. The subroutines get compiled and tucked away for later use.

Besides from the code that's loaded and executed, two more special things happen. Firstly, if the last statement in the module did not evaluate to true, the Perl compiler throws an exception (usually halting your program before it even starts). This is so that a module could indicate that something went wrong, although in reality this feature is almost never used. Virtually any Perl module you care to look at will end with the statement `1;` to indicate successful loading.

The other thing that happens when a module is `use`d is that its `import` subroutine (if one exists) gets called with any directives that were specified on the `use` line. This is useful if you want to export functions or variables to the program that's using your module for functional programming but is almost never used (and very often discouraged) for object oriented programming.

As you've no doubt guessed by now, modules and packages often go hand-in-hand. We know how to use a module, but what are the rules on writing one? Well, the big one is this:

> A module is a file that contains a package of the same name.

That's it. So if you have a package called `Tree::Fruit::Citrus::Lime`, the file would be called `Tree/Fruit/Citrus/Lime.pm`, and you would use it with `use Tree::Fruit::Citrus::Lime;`.

A module can contain multiple packages if you desire. So even though the module is called `Chess::Piece`, it might also contain packages for `Chess::Piece::Knight` and `Chess::Piece::Bishop`. It's usually preferable for each package to have its own module, otherwise it can be confusing to your users how they can load a particular package.

When writing modules, it's important to make sure that they are well-named, and even more importantly that they won't clash with any current or future modules, particularly those available via CPAN. If you are writing a module for internal use only, you can start its name with `Local::` which is reserved for the purpose of avoiding module name clashes.

You can read more about writing modules in **perldoc perlmodlib**, and a little on pages 554-556 of the Camel book.

If you *really* want to know how to export things from your modules, then read **perldoc Exporter** for the story. Please don't export things by default unless you have a very, very good reason.

Warnings

When your module is used by a script, whether or not it runs with warnings depends upon whether the calling script is running with warnings turned on. You can (and should) invoke the `use warnings` pragma to turn on warnings for your module without changing warnings for the calling script.

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

Exercises

1. Create a directory named `lib` and open a file named `MyTest.pm` in it.
2. Create a package named `MyTest` in the file, and define at least two functions, `pass` and `fail` that print some amusing output. Make sure that it uses both `strict` and `warnings`.
3. Test that your module has no syntax errors by running **perl -c MyTest.pm**. (The `-c` tells Perl to check your code)
4. Save the file, and create a simple Perl script that uses your module. Have it call the functions `MyTest::fail` and `MyTest::pass`

Chapter summary

- To write a package, just write `package` *`package_name`* where you want the package to start.
- Package declarations last until the end of the enclosing block, file or eval (or until the next package statement).
- A module is a file that contains a package of the same name.
- Modules usually end with the statement `1;`
- A module can contain multiple packages, but this is often a bad idea.
- It's often a good idea to put your own modules into the `Local` namespace.

Chapter 5. Our first Perl Object

In this chapter...

We’ve just learnt (or been reminded of) some of the basic concepts of Object Oriented programming, but how do we do that in Perl? It’s easier than most people think. This is what we’re going to cover.

Classes are just packages

To create a class in Perl, just create a package of the same name. For example, if we wanted to create a PlayingCard class, we’d do the following:

```
package PlayingCard;
```

That’s all there is to it. Mind you, our class is very boring as it doesn’t do anything, but it does exist. Anything after our `package` line to the end of the file (or until another `package` line) is placed into the `PlayingCard` class.

Methods are just subroutines

Methods are just subroutines that exist within a particular class that can perform operations on objects of that class. So if we write a subroutine in our `PlayingCard` package, that becomes a method. Here’s an example:

```
package PlayingCard;            # This sets our class

sub get_suit {
        # Code here to return the card’s suit.
}
```

This sure has been easy so far. How do we call that method which we just wrote? Well, if you’ve used a class like `HTML::Template` (or one of many other object oriented modules), you probably already know. Calling a method is very similar to doing any other sort of access which involves a reference, you use the `->` operator:

```
$object_ref->method(@args);    # Access method via object reference.
```

Compare this with:

```
$array_ref->[$index];          # Access array element via array reference.
$hash_ref->{"key"};            # Access hash value via hash reference.
```

Note that calling a *method* on an object is very similar to accessing data via a reference. In this case the reference to the object is on the left of the arrow, and what we wish to access is on the right.

There’ll usually be many methods available on an object. Here might be some examples with our `PlayingCard` class:

```
$card->get_suit();
$card->get_value();
$card->swap_with_ace_up_sleeve();
```

When a method gets called, it receives a reference to the object upon which it was invoked as its first argument. As such, it's common (and recommended) to have code like this:

```
package PlayingCard;

sub get_suit {
       my ($self, @args) = @_;
       # $self now contains my object reference, and @args any
       # arguments that were passed to this method.

       # Code to return my suit goes here.
}
```

Blessing a referent creates an object

A *referent*, for those not familiar with the term, is something that is referred to by a *reference*.[1] In Perl, any type of variable (such as an array, hash or scalar) can also be an object. Hence there's no real trick in creating your object, instead the magic comes from how you tell Perl to associate an object with a particular class. We do this by *blessing* (that's Perl-specific terminology) the object-to-be.

To bless an object we use the in-built Perl function which is aptly named `bless`. The `bless` function takes two arguments, a reference to the variable to bless, and a string containing the class in which to bless. Here's an example of how we might do this for a member of the PlayingCard class.

```
my $card = {
        _suit  => "spades",
        _value => "ace"
};

bless($card,"PlayingCard");
```

Pretty painless, isn't it? You need to remember that while we pass a reference to the `bless` function, it's the underlying (in this case anonymous) hash that changes, not the reference. You'll note that our hash keys started with underscores, this is a convention used for marking attributes that are intended to remain private to this class. Note that it doesn't guarantee privacy, but merely relies upon convention.

Actually, in Perl you can bless anything you can get a reference to. This includes not only the arrays, hashes and scalars that we've just mentioned, but also things you wouldn't normally expect, such as subroutines, regular expressions, and typeglobs.

Now that we know how to create an object, and have an example of some likely attributes for it, we can fill in our `get_suit` method above.

```
package PlayingCard;

sub get_suit {
       my ($self) = @_;

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

Constructor functions

Now, if you've been thinking that creating a variable, populating it with data, and then blessing it every time we want a new object is a lot of hard work, you'd be right. Most of the advantages of object oriented programming would be lost if we had to do all this work ourselves. What we'd like is something (preferably in the appropriate class) which can create objects for us. That's commonly called a *constructor function*.

In Perl, constructor functions are always called `new` by convention. A standard constructor function takes some information about the initial state of the object, and returns an appropriately constructed object.

```
package PlayingCard;

sub new {
        my ($class, $value, $suit) = @_;

        # Create an anonymous hashref and naively fill in our
        # fields.

        my $self = { _value => $value, _suit => $suit };

        return bless($self,$class);
        # Since bless is often the last thing in a constructor, it
        # returns the reference for convenience, so this is the
        # same as:
        # bless($self, $class);
        # return $self;
}
```

Let's explain briefly how that all works. Our constructor expects a class as its first argument, and a card value and suit as its second and third. We create an anonymous hash reference, and populate that with the values that we've been passed. Having done that, we bless our anonymous hash (via its reference) into the class that we've been given and return a reference to the blessed hash.

That's pretty straightforward, but you might be wondering why we want a *class* passed to our constructor function. We already know that we're in the `PlayingCard` class, why have a class passed in?

The reason has to do with how constructor functions are usually called. Rather than calling the function directly like this:

```
my $card = PlayingCard::new("PlayingCard","Ace","Spades");
```

instead we treat the constructor as a *class method*, and call it thus:

```
my $card = PlayingCard->new("Ace","Spades");
```

If you've never done object oriented programming before, you're probably wondering what a class method is. Well, the methods we've been calling from objects are properly known as *object methods*. They call a method on an object, and the subroutine which handles the call gets the object as its first argument. A class method, as you've probably guessed, gets called upon a *class*, and receives the class-name as the first argument.

Class methods are methods that are attached to a class but not an object. Constructors are a good example of these, we want to form an object out of nothing. We'll see more of them as we continue through this course.

Now, that still doesn't answer why we want to use the class name that's been passed to us, rather than blessing into `PlayingCard` directly and saving ourselves a little typing.

The reason for that has to do with inheritance. If someone decides to use our class as a parent for their own derived class, then our constructor function would receive the name of the derived class when invoked. If we always blessed into the `PlayingCard` class, then someone wanting to derive a `PlayingCard::UpMySleeve` class, would find that our constructor simply wouldn't work for them (it would always return a normal `PlayingCard` object).

Don't worry too much about that yet, when we get to the chapter on inheritance you'll see many good examples on why this is so.

PlayingCard in full

Here's the `PlayingCard` class in full.

```
package PlayingCard;            # Our class name

# The constructor function (a class method)
sub new {
        my ($class, $value, $suit) = @_;

        # Create an anonymous hashref and naively fill in our
        # fields.

        my $self = { _value => $value, _suit => $suit };

        return bless($self,$class);
}

# An object method returning the value of this card's suit
sub get_suit {
       my ($self) = @_;

       return $self->{_suit};
}

# An object method returning the face value of this card
sub get_value {
        my ($self) = @_;

        return $self->{_value};
}

1;      # Required if we've written this as a module.
```

Exercises

1. Create a `Coin` class in a file called `Coin.pm`.
2. Create the following methods:
   - `toss`: this function randomly changes the state of the coin to heads or tails, as if the coin had just been tossed up.
   - `get_state`: this function tells us whether the coin is heads up or tails up at the moment.

3. Create a constructor for your `Coin` class making sure that it ensures that the state is set to something valid.
4. Write a program that `uses` your `Coin` class and creates two coins. Make it flip these two coins a number of times and report on each outcome.

An answer for this can be found in `exercises/answers/test_coin.pl`.

Chapter summary

- Perl objects are variables, a collection of attributes.
- Methods belong to *classes* not objects, and are divided into class methods and object methods.
- To create an object in Perl we need only remember three rules:
  - Classes are just packages
  - Methods are just subroutines
  - Blessing a referent creates an object
- In Perl objects are always accessed via a reference, objects themselves are never passed around.
- Calling an object method can be done using the arrow notation. `$object_ref->method()`
- Constructor functions in Perl are conventionally called `new()` and can be called by writing: `$new_object = ClassName->new()`.

Notes

1. Referents used to be called *thingies* (yes, that was the technical term), but that used to confuse people as well.

Chapter 6. Practical Exercise - Playing Cards

We've already seen the start of a `PlayingCard` class, and learnt the very basics of object oriented programming in Perl. Now we'll take that knowledge and practice writing a simple module.

Group Exercises - Planning the Class

An important part of any project is planning. Determining what is required of your class and how it will be required to function before you begin coding can save many hours of frustration later. This course does not aim to teach you these skills, but it does assume that you'll spend some time thinking and documenting a class before you begin to write it.

Let's say that we wish to continue with the `PlayingCard` class. It has many useful applications, simulating variants of Poker, counting cards at casinos, writing online card-game clients, and so on. Here are some things to think about as a group:

1. What sort of information will our PlayingCard need to store? Think of any attributes that we might need to store.
2. What would be the best way to store the attributes which we've just discussed? Remember that card values have an *ordering*. It would be nice to preserve this so that we can compare two cards and see which is the highest.
3. The card game *five-hundred* when played with six or more players introduces cards valued 11 and 12 in each suit, and cards valued 13 in both red suits, which come below the picture cards in value [1].

   Can our PlayingCard class handle this situation? What about jokers? What about games where aces are high instead of low?

4. What arguments should our constructor function take? How can we verify that we've been given valid arguments?
5. What sort of operations would we like to do with our cards? Which of these make sense with regards to an individual card?

Individual Exercise - Writing the Class

In `exercises/lib/PlayingCard.pm` you'll find some skeleton code for writing a `PlayingCard` class. Try the following exercises:

1. Fill in the constructor function so that it can be used to create new `PlayingCards`.
2. Fill in the two accessor methods, `get_suit` and `get_value`. Verify that they work correctly.
3. Add the other methods that you and your group decided upon in the exercises above.
4. Create a program that `uses` your module. Have it generate a deck of cards (without jokers), shuffle the deck, and print the first five cards. You can use `@deck = sort {rand() <=> 0.5 } @deck` to shuffle the deck.
5. What happens if you just try to print the object references in the last exercise (for example, `print $card`)? What needs to be done instead to print these in human readable forms?

Practical Usage - The Card Game "War"

There are many variations of the simple card game *war*, but they all share similar rules. The game is played by two players, and a 52 card deck is shuffled, and each player is dealt half the deck. The players then draw cards simultaneously, and compare their values. The player holding the highest value card wins their opponents cards, and these are placed onto the bottom of their pile. This process repeats until only one player has cards remaining, and is declared the winner.

If the value of the cards are equal, then each player draws another card and compares it to their opponent's. The winner then claims all four cards as theirs.

It's actually possible for infinite win-lose cycles to occur in this game, depending upon the initial card ordering. As such, you may wish to shuffle the players' cards after each round.

1. Use your `PlayingCard` class to write a program which implements the game of war. If you are used to playing with different rules, then you may use those instead of the ones listed above.

An answer for this game can be found in `exercises/answers/war.pl`. Make sure you try to solve the problem before peeking at the answer.

Notes

1. Not to mention other fun things such as the Jack of the other same colour suit suddenly becomes a member of the Trumps suit, when a suit is bid Trumps. Or that these two Jacks of the same colour each have a higher value than all the other cards of the Trumps suit (except the Joker which we'll ignore here). The two Jacks of the opposing colour suits remain in their normal value positions as immediately less than the appropriate Queen cards.

Chapter 7. Argument Passing

In this chapter...

In this chapter we look at how we can improve our subroutines and methods by using named parameter passing and default arguments. This is useful both in object oriented coding and standard coding, and is best used whenever a subroutine needs to take many arguments, or where more than one argument is optional

Named parameter passing

We'll use a particular form of parameter passing in these notes, and it's so useful that it deserves a special mention. It's called *named parameter passing* and it usually starts like this:

```
sub method {
        my ($self, %args) = @_;
        # ...
}
```

`$self` is the object, which you've already heard about. It's the `%args` that is important. The arguments for our methods are loaded into a hash for ease-of-use. We'll see how it works, and why it's so good.

Most programming languages, including Perl, pass their arguments *by position*. So when a function is called like this:

```
foo("Paul","Perl","Buffy");
```

the `foo()` function gets its arguments in the same order in which they were passed (in this case, `@_` is `("Paul","Perl","Buffy")`). For functions which take a few arguments, positional parameter passing is succinct and effective.

Positional parameter passing is not without its faults, though. If you wish to have optional arguments, they can only exist in the end position(s). If we want to take extra arguments, they need to be placed at the end, or we need to change every call to the function in question, or perhaps write a new function which appropriately rearranges the arguments and then calls the original. That's not particularly elegant. As such, positional passing results in a subroutine that has a very rigid interface, it's not possible for us to change it easily. Furthermore, if we need to pass in a long list of arguments, it's very easy for a programmer to get the ordering wrong.

Named parameter passing takes an entirely different approach. With named parameters, order does not matter at all. Instead, each parameter is given a name. Our `foo()` function above would be called thus:

```
foo(name => "Paul", language => "Perl", favourite_show => "Buffy");
```

That's a lot more keystrokes, but we gain a lot in return. It's immediately obvious to the reader the purpose of each parameter, and the programmer doesn't need to remember the order in which parameters should be passed. Better yet, it's both flexible and expandable. We can let any parameter be optional, not just the last ones that we pass, and we can add new parameters at any time without the need to change existing code.

The difference between positional and named parameters is that the named parameters are read into a hash. Arguments can then be fetched from that hash by name.

```
foo(name => "Paul", language => "Perl", favourite_show => "Buffy");

sub foo {
        my (%args) = @_;

        my $name = $args{name} || "Bob the Builder";
        my $language = $args{language} || "none that we know";
        my $favourite_show = $args{favourite_show} || "the ABC News";

        print "${name}'s primary language is $language.  " .
              "$name spends their free time watching $favourite_show\n";
}
```

Calling a subroutine or method with named parameters does not mean we're passing in an anonymous hash. We're passing in a list of name `=>` value pairs. If we wanted to pass in an anonymous hash we'd enclose the name-value pairs in curly braces `{}` and receive a hash reference as one of our arguments in the subroutine.

Some modules handle arguments this way, such as the `CGI` module, although `CGI` also accepts name `=>` value pairs in many cases.

It is important to notice the distinction here.

Default arguments

Using named parameters, it's very easy for us to use defaults by merging our hash of arguments with our hash of arguments, like this:

```
my %defaults = ( pager => "/usr/bin/less", editor => "/usr/bin/vim" );

sub set_editing_tools {
        my (%args) = @_;

        # Here we join our arguments with our defaults.    Since when
        # building a hash it's only the last occurrence of a key that
        # matters, our arguments will override our defaults.
        %args = (%defaults, %args);

        # print out the pager:
        print "The new text pager is: $args{pager}\n";

        # print out the editor:
        print "The new text editor is: $args{editor}\n";
}
```

Exercises

1. Rewrite the `foo` subroutine above to use a hash of default arguments rather than assigning individual defaults.

Named parameters and object constructors

In object oriented coding we use named parameters most during object construction. This allows us to choose reasonable defaults for arguments and saves the programmer from having to memorise the order for the (possibly numerous) arguments. When we reach the chapter on *inheritance*, we'll see why this is doubly useful.

Here's an example of being able to used named parameters to quickly and easily build a `DrinksMachine`:

```
package DrinksMachine;
use strict;
use warnings;
use Carp;

# The default_fields for a drinks machine.
#  desired_temperature - best temp for operation, deg. Cel
#  drinks              - drink flavours
#  price               - all drinks have the same price, standard decimal
#  starting_change     - the change we start out with.  Represented by a list
#                        of the number of coins in following denominations:
#                        $2,  $1,  50c,  20c,  10c,  5c
#                        so  [qw/0 0 50 50 30 10/] makes, 0 x $2, 0 x $1,
#                        50 x 50c, 50 x 20c, 30 x 10c, 10 x 5c.
my %default_fields = (
        desired_temperature => 4,
        drinks => [qw/cola orange lemonade squash water/],
        price => 1.20,
        starting_change => [qw/0 0 50 50 30 10/]
);
# Our required fields.
my @required_fields = qw/location/;

# All the fields that we expect to get
my @fields = (keys %default_fields, @required_fields);

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

        # Check we have all required arguments.
        foreach (@required_fields) {
                exists($args{$_}) or croak("Required field $_ omitted");
        }

        # Any args we pass in will override the defaults.
        %args = (%default_fields, %args);
        my %this;

        # Copy only the fields that we want into our hash.
        @this{@fields} = @args{@fields};

        return bless(\%this,$class);
}
```

You'll have noticed the line:

```
@this{@fields} = @args{@fields};
```

in that last listing of code. This is a hash-slice. We can take array slices as follows:

```
my @array = qw/a b c d/;
my @slice = @array[0,3]; # @slice = ('a', 'd')
```

to get at only parts of an array or to rearrange the ordering of an array, and hash-slices are exactly the same.

We precede a hash-slice with an `@` sign because we're working with a list of hash values. `@this{@fields}` are all the values from `%this` with the keys from in `@fields`.

Exercises

1. Modify your `PlayingCard` class so that its constructor uses named parameters rather than positional parameters.
2. Do the same for your `Coin` class as well.

Chapter summary

- Parameters in Perl are usually passed "by position".
- Positional parameter passing makes having independent optional arguments or extra arguments difficult.
- Named parameter passing makes independent optional arguments and extra arguments easy.
- To pass named parameters to a subroutine all we have to do is give the subroutine a list of name, value pairs when we call it, and to extract the hash from `@_` in our subroutine.
- Named parameter passing allows the programmer and class user to call subroutines with arguments in different orders.
- Named parameter passing makes it very easy for us to handle defaults, especially in constructor functions.

Chapter 8. Class methods and variables

In this chapter...

In this chapter we will discuss class methods and class variables. We'll look at some very special class methods as well as more generalised ones. In addition, we'll discuss some common uses of class variables.

What is a class method?

Most of the methods we've discussed so far have been methods that we call from objects, which are unsurprisingly called *object methods*. *Class methods*, on the other hand, are called directly on classes, and don't have a single object associated with them at all.

We've already seen one instance of a class method, and that's constructor functions. Even though constructors return a freshly created object, they're called directly on the class, like this:

```
my $card = PlayingCard->new(suit=>"diamonds", value=>"jack");
```

Notice that we invoke the name of the class in order to use a class method, in the same way that we invoke an object to use an object method. Like the constructors that we've already used, the method receives the name of the class as its first argument.

Class methods are used for functionality that affects a whole class of objects. For example, I might have a class method that increments the age of all stock in my inventory, or return the number of times a particular class of object has been created.

An example class method

We've already seen one instance of a *class method*, and that's `new`, the constructor function. In this case, we have a class method because there's no object to work upon. However, there are other times when class methods come in handy as well.

Let's think back to our `PlayingCard` class. Rather than requiring our user to deal have to manually create a deck of cards (a very common operation) we could write a class method to do it:

```
package PlayingCard;

sub new_deck {
        my ($class) = @_;      # This is the class which was invoked.
        my @deck;

        foreach my $suit (qw/hearts spades diamonds clubs/) {
                foreach my $value (2..10, qw/jack queen king ace/) {
                        push @deck, $class->new( $value, $suit );
                }
        }

        return @deck;
}
```

Notice that we use `$class->new(...)` rather than `PlayingCard->new(...)`. By using the first syntax we invoke `new` on the class which was used to invoke our `new_deck` method. This is important if we inherit from our class later on. We'll see more about how inheritance works later in this course.

Our class method *could* be called from an object, since Perl itself does not enforce how a particular method is invoked. In this case, we would have received an *object* and not a class as our first argument. If we wished we could test for this and give an appropriate warning, however it's very easy to allow this alternate calling syntax as well:

```
sub new_deck {
        my ($class) = @_;

        # Get the class of the invoking object, if required.
        $class = ref($class) || $class;

        # ...
}
```

The `ref` function returns the class of the object passed to it, or false if given something other than a reference. In this way, we can obtain our desired class regardless of if we're invoked as a class method or an object method.

Invoking a class method on an object isn't as uncommon or undesirable as you might think. There are many instances where programmers will use an object but not know which class of objects they're dealing with. For example, you probably don't know the class of statement handles in the `DBI` module, nor should you need to know this in order to use them.

Class variables

We've already covered attributes on objects in some detail, but from time to time we also wish to have an attribute or variable which is common to all objects in a class. We call such a variable a *class variable*. Class variables easy to create and use in Perl. As you might expect, there's more than one way to do it.

```
package PlayingCard;

my $Cards_created = 0;

sub new {
        # ...
        $Cards_created++;
}

sub card_count {
        return $Cards_created;
}
```

Here we create a lexically scoped variable `$Cards_created` which tracks the number of times our constructor function has been used. Since the declaration of this variable is in the same scope as the subroutines which use it, they are able to make use of it. Anything else cannot, not even with `$PlayingCard::Cards_created`, as it's not possible to name a lexical variable outside of its scope. You can think of class variables made in this way as being private if you're familiar with other object oriented languages.

It's possible to create *really* private class variables in this way. For example, here is a variable which is shared between two methods, but cannot be accessed by any other sections of code:

```
package PokerGame;

{
        # Only subroutines inside this block can use variables
        # declared within it.
        my $House_min_bet = 0.50;  # 50c minimum bet

        sub set_house_min {
                my ($class, $new_min) = @_;
                $House_min_bet = $new_min;
        }

        sub get_house_min {
                return $House_min_bet;
        }
}
```

In the code above, the *only* way to get access to `$House_min_bet` is through the two subroutines defined in the same block as it. Other code, even code in the same class, cannot access the variable except through these methods.

Package variables and class variables

You may recall from the modules and packages chapter that package variables can be used as globals throughout our program. If we wish to have a class variable that can be accessed by name from anywhere in our program, we'd need to declare it as global to our class. Since a class is just a package, we create a package variable and use that.

```
package PlayingCard;

our $Cards_created;             # Preferred for Perl 5.6.0 and above.
# use vars qw/$Cards_created/;  # Preferred for older versions.
# $PlayingCard::Cards_created;  # Acceptable, but requires we always
                                # use the full name under strict.
```

In all the cases above, both our package and external code can access the variable using `$PlayingCard::Cards_created`. You can think of this like a public variable if you're used to other object oriented languages.

Exercises

1. Create a class method `print_statistics` to your `Coin` class. Make this function print out what percentage of heads and what percentage of tails have come up over how many coin tosses. Add any class variables that you find you need.
2. Change your coin program to toss the 2 coins 100 times and then to print out the statistics using the class method `print_statistics`.

An example answer for this can be found in `exercises/answers/statistics.pl`.

Chapter summary

- Class methods are used when we wish to perform an operation which affects all members of a class, or for which no object exists on which to invoke the method.
- Class methods in Perl can be invoked from objects as well, should we desire.
- Perl allows us to create class variables which can be accessed by any part of our code, or variables which are only available within a particular class.
- We can create very private class variables which are only available to certain methods within a class, and not the entire class.

Chapter 9. Destructors

In this chapter...

We've seen how to bring objects into the world by blessing an appropriate data type. We haven't yet looked at how objects are destroyed, and that's the topic of this chapter.

Perl's garbage collection system

To understand how Perl manages variables and objects (which are really just specially blessed variables), we need to know a little about Perl's garbage collection system. As you've probably (consciously or sub-consciously) been aware, there's no need to explicitly allocate or free memory in Perl. When I create a variable, the memory is allocated automatically. When I assign elements to an array or hash, those structures are extended as needed. When I put data into a scalar, the capacity of that scalar grows as required. All of this saves oodles of headaches and programmer time.

What's not immediately obvious is under what circumstances Perl frees memory. Perl keeps a *reference count* to each data structure, recording how many things point to this particular chunk of data. When the reference count drops to zero, the garbage collector kicks in and the memory is freed. Here's an example

```
my $greet_ref;
{
        my $greeting = "Hello World";
        my $farewell = "Goodnight World";

        # Add another reference to $greeting.
        $greet_ref = \$greeting;

        # End of block means that variables created with my
        # go out of scope.  However, the data in $greeting
        # lives on, because there's still a reference to it.
}
```

It's possible to cause Perl's garbage collection system to screw-up if you're using circular references. For example, the following situation will leak memory:

```
{
        my ($a, $b);
        $a = \$b;
        $b = \$a;
}
```

even after `$a` and `$b` go out of scope. An even simpler case is when a variable is a reference to itself. Since the reference count never drops to zero, these bits of memory never get collected (although they're cleaned up as the perl interpretor shuts down at the end of the program).

Consider the case where we wish to model a railroad connection map. Inside our object we have references to stations and depots, each of which contains references to adjacent stations and depots. This data is likely to contain many circular references. When the last reference to our railroad map disappears, we want to make sure that we break the references in its internal representation so that memory can be correctly freed. One of the ways of doing this with objects is using a *destructor function*.

Destructor functions

Put simply, a destructor function in Perl is called to tidy up an object that's about to be destroyed, in the same way that a constructor function sets things up for an object that is being created.

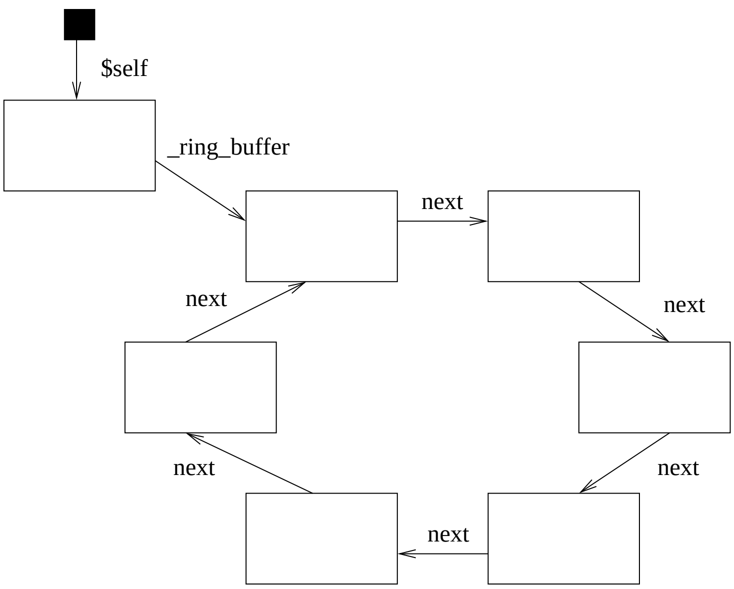
Most objects don't require destructor functions, nothing special needs to be done when a program doesn't need a particular `PlayingCard` anymore. However, some objects *do* require special treatment. For example, if I have an object which is controlling a modem or serial terminal, I might want to make sure that the device gets reset back to a known state upon my program finishing with it. If I have an object which is keeping a cache of information, I might want to write that cache to the disk for speedy access next time. As we've seen above, if an object contains circular references in its data, we want to break those references to make sure that the clean-up done by the garbage collector is complete.

When an object's reference count hits zero, the `DESTROY` method (if it exists) is invoked on it, before that object's memory is reclaimed. This gives the object one last chance to clean itself up before disappearing. The `DESTROY` method gets the object as its first argument, just like any other object method. If no `DESTROY` method exists, then the object's memory is reclaimed as per any other Perl variable.

Destructors also get called in a separate phase when the interpretor is exiting, before other variables are cleaned. This is to ensure that destructors which perform important tasks like saving data to disk get a chance to do so.

Writing a destructor is easy. Let's pretend our object has a ring buffer as one of its attributes. Before the object is destroyed we wish to break the circle of references:

Figure 9-1. Object has a ring buffer



To do this all we have to do is remove a single link from the circle. We can do this with a destructor like the following:

```
sub DESTROY {
        my ($self) = @_;

        # Break one of the references in our ring-buffer, so that
        # it will be cleaned up properly.
```

```
        delete($self->{_ring_buffer}->{next});

        # Okay, the Perl garbage collector will take care of the
        # rest.  Bye!

}
```

Destructor functions are free to do whatever they like, although you should have a good reason if you do anything other than the required cleanup that's expected. Remember that destructors are called are not only called during the normal running of your program, but also when your program is exiting. Assumptions about database connections, open files, and the like may not be valid.

Exercises

1. Write a destructor function for the `PlayingCard` class. Have it print a message to `STDERR` whenever a card is discarded.
2. Run one of your previous scripts that makes use of the `PlayingCard`s and observe its behaviour.

Other uses for destructors

Destructors have a lot of use outside of simple clean-up. Destructors can be used to cleanly close connections to servers or clients, or a convenient place to log information about an object's usage. More importantly, destructors are a convenient place to *serialise* an object, that is, to turn it into a form suitable for storage and later retrieval.

The example below demonstrates a class which uses the `Cache::FileCache` module to create objects which are persistent across processes.

```
# Naive persistent object class.  This assumes that only one
# process will be using an object at any given time.  No locking
# or checking is done to ensure that double-update or race
# conditions are avoided.

package Persistent;

use Cache::FileCache;   # Could be any Cache::Cache module, TMTOWTDI

our $cache = Cache::FileCache->new();

# If the argument "name" is passed, and the object already exists
# in our cache, we skip all initialisation and create the object
# directly.

sub new {
        my ($class, %args) = @_;
        my $this;

        # Grab our object from the cache, if it exists.
        if ($args{name}) {
                $this = $cache->get($args{name});
                return $this if $this;
        }

        # Otherwise, proceed with regular initialisation.
        $this = bless({},$class);
```

```
        $this->_init(%args);
        return $this;
}

# Stuff the object into our cache before releasing its memory.
sub DESTROY {
        my $this = shift;
        $cache->set($this->name,$this);
}
```

It's now possible for us to inherit from the `Persistent` class, and provided we use the inherited constructor and provide a `name` argument during creation, our objects will be made persistent across processes.

Group Exercises

1. The `Persistent` class has a problem when multiple processes may wish to use the same object at the same time. What solutions may exist to solve this problem?
2. The `Persistent` class has other problems, in addition to those mentioned in the previous exercise. What are these problems? How may they be solved?

Chapter summary

- Destructor functions are called upon an object when the reference count to that object has dropped to zero.
- Destructors allow us to clean up after our object, so if our object controls a modem it might set it to a known state before leaving, or if our object is the last outside reference to a loop of other objects, it might break the loop so that those objects can also be destroyed.
- Destructors can be used for other things that need to occur before an object is destroyed, such as logging of statistics dealing with that object, or serialising the object for later use.
- Under most situations, explicit destructor functions are not required.

Chapter 10. Inheritance

In this chapter...

One of the great virtues of object oriented programming is the ability to easily take existing objects and extend them to meet new requirements. The primary means to achieve this is via *inheritance* which we'll talk about here.

So what is inheritance in Perl?

If you've used another object oriented language, you're probably already quite familiar with the idea of inheritance. Even if you haven't, I suspect that you've grasped the idea by now. Inheritance is a way of extending the functionality of a class by *deriving* a more specific sub-class from it.

Let's take people, for example. There are lots of different classes of people, we might have `Gardeners`, `ChessPlayers`, `Cyclists`, and so on. A person may be all of these classes at once, but each of them provides a very different set of behaviours. When something is a member of two or more classes at once, we call this *multiple inheritance*.

In addition to there being many different types of people, some classes will be more specialised than others. For example, a `PerlTrainer` *is a* `Trainer` *is a* `Teacher`. So a `PerlTrainer` is a special type of `Trainer`, which in turn is a special type of `Teacher`. This means that a `PerlTrainer` can do everything that a `Trainer` can do, as well as a few tricks of their own.

Inheritance in Perl is a much more relaxed affair than inheritance in other programming languages. In fact, inheritance in Perl is nothing more than a way of specifying where to look for methods. I'll repeat that, because those with prior object oriented experience will be muttering in disbelief right now. *Inheritance in Perl is nothing more than a way of specifying where to look for methods.* That's it, end of story, nothing more to see here. Move along please.

Attributes do not get inherited. Ancestral constructors and destructors do not get called. Compile-time consistency checks on interfaces or abstract methods do not happen. Actually, that's not completely fair. None of those things happen unless you *want* them to happen. Having a choice is a powerful (and sometimes dangerous) thing, but in Perl the choice is yours to make.

Since the only thing out-of-the-box inheritance affects is method calls, we'll discuss that before going any further.

Method dispatch

The process of finding which method to call is known as *method dispatch*, and different programming languages will handle it in different ways. Perl looks for methods using a depth-first, left-to-right search of the tree of ancestors.

The ancestors of a class are found by looking at the `@ISA` array. Since this is a package variable, this is one of the few times when you *do not* want to use `my`. Instead, you should declare the variable with the `our` keyword (in 5.6.0 and above), or using the `use vars` pragma (in any version of Perl).


A better explanation of the message dispatch mechanism is explained in Conway's book, pages 169-171.

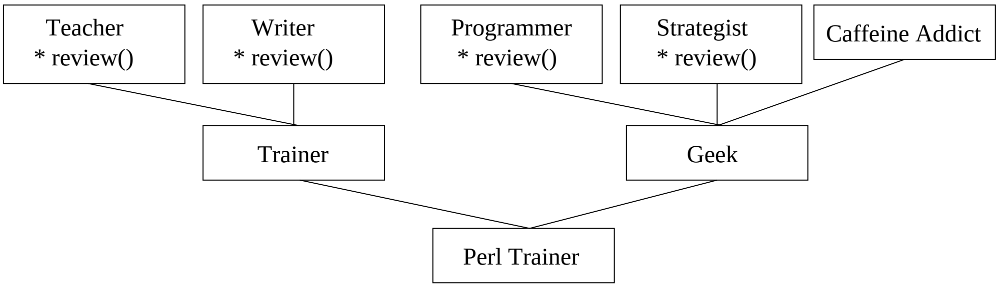
This is best demonstrated with an example. It's all really quite simple.

```
package PerlTrainer;
our @ISA = qw/Trainer Geek/;  # 'our' syntax only valid in >= 5.6.0

package Trainer;
use vars qw/@ISA/;            # Portable across all versions of Perl 5.
@ISA = qw/Teacher Writer/;

package Geek;
our @ISA = qw/Programmer Strategist CaffeineAddict/;
```

Figure 10-1. The PerlTrainer inheritance tree



When a method call is made (say `review`) on an object in the `PerlTrainer` class, the classes are searched in the following order:

- `PerlTrainer`
  - `Trainer`
    - `Teacher`
    - `Writer`
  - `Geek`
    - `Programmer`
    - `Strategist`
    - `CaffeineAddict`

until the method is found.

As soon as the method is found (in this case, at the `Teacher` class), it's called immediately, and cached so that Perl doesn't need to go through all that hard work again. It's important to note here that you always end up with the first available method in the left-most inheritance chain. Conway aptly refers to this as the "left-most ancestor wins".

Directed dispatch

There will be times (and we'll see them later this chapter) that we want to start the dispatch mechanism at a particular place in the class tree. In these cases we use a special syntax for method calls, like this:

```
$trainer->Teacher::instruct(@args);
```

This directs Perl’s method dispatcher to begin looking for the `instruct` method in the `Teacher` class and calls this on the `$trainer` object. If the method isn’t found on the `Teacher` class, then the parent classes of `Teacher` will be searched, and so on. If neither the `Teacher` class nor any of its ancestors have an `instruct` method this will cause a run-time error.

Directed dispatch allows us to call the `Geek review` method as follows:

```
$trainer->Geek::review(@args);
```

Since the `Geek` class does not implement it’s own `review` method the dispatch mechanism would traverse the inheritance tree of the `Geek` class and find the `Programmer review` method and call that.

Using this notation, it’s actually possible to specify a class of which your object is not a member. This mainly has uses in invoking *pseudo-classes*, which have particular side-effects. We’ll see more on this next chapter.

Dispatch via subroutine reference

There’s a third way of invoking methods in Perl, and that’s directly with a subroutine reference. It’s possible to get a reference to a subroutine in a few ways (for example, `\&foo` gives us a reference to the subroutine named `foo`). Given such a reference, it can be called as a method directly, without involving the method dispatcher at all. This is very fast:

```
my $subref = \&Strategist::review;
$trainer->$subref(@args);
```

Similar to the directed dispatch above, it’s possible to call methods that don’t exist on the object in this fashion. We’ll see some uses for this notation later on when we examine the `can()` universal method.

Be careful, the following line of code:

```
my $subref = \&Strategist::review();
```

calls the `Strategist::review` subroutine, and then takes a reference from what it returns. When making subroutine references, we have to make sure that we do not include parentheses.

Exercises

1. You can find the source for the `PerlTrainer` and related classes in your `exercises/lib` directory. Write a small script to `use` the `PerlTrainer` class, create a `PerlTrainer` object, and call the `review` method on it. Which class’ method gets called?
2. Invoke the `review` method but direct the dispatcher to start looking in the `Geek` class. Which class’ method gets called this time?
3. Obtain a reference to the the `review` method in the `Strategist` class by using `my $subref = \&Strategist::review`. Use it to call the method directly on your `PerlTrainer` object.

Constructors and inheritance

You'll remember that we mentioned that attributes do not get inherited in Perl, nor does every constructor get called when an object is created. This is different to most other object oriented languages.

In Perl, a constructor is just another (class) method, except it returns a newly blessed object. When we call `PerlTrainer->new()`, Perl does the left-most inheritance search, and calls the first (and only the first) constructor that it finds. With our example above, if `PerlTrainer` had no constructor, but `Trainer` did, then that would be called.

It's here that we finally see why it's so important to bless an object into the class that was passed as the constructor's first argument. When we call `PerlTrainer->new()` we want a `PerlTrainer` object, and this is what the constructor is passed, even if it's the `Trainer` or `CaffeineAddict` constructor that eventually gets called.

What we haven't yet discussed is how to ensure all constructors get the chance to properly initialise an object. Sure, the `Trainer` constructor will correctly initialise attributes for `courses_taught` and `notes_revised`, but is unlikely to even know about `blood_caffeine_level`.

In Perl, the most common solution is to separate the object *construction* from the object *initialisation*. This all happens internal to the class, of course. We don't want users writing code like this:

```
my $trainer = PerlTrainer->new;
$trainer->init(name => "Paul Fenwick");
```

That would just be asking for trouble. Rather, the constructor function should call the initialisation function itself.

So, what's the big deal about splitting creation from initialisation? Why bother in the first place if the user doesn't see nor care about it? Let's take the following example:

```
package PerlTrainer;

use vars qw/@ISA/;

@ISA = qw/Trainer Geek/;

# Constructor method.  Just creates the hash and then passes
# the work off to the initialiser method.

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

        $this->_init(@args);

        return $this;
}



# Initialiser method, does all the hard work.
sub _init {
        my ($this, %args) = @_;

        # Initialise the object for all of our base classes.

        $this->Trainer::_init(%args);
        $this->Geek::_init(%args);
```

```
        # Class-specific initialisation.
        $this->{_perl_courses} = $args{courses} || [];

        # Return the initialised object.
        return $this;
}
```

Our `_init` function first calls the `_init` functions on its base classes, and then does its own class-specific initialisation. In this way, all the classes get a chance to do whatever work is needed on the newly created object. If we had called constructor methods, we would get back many different objects, when we only want to be working with one.

Universal methods

We'll return to initialisers shortly, but first we'll introduce two very special methods that exist on all objects. Those methods are `isa()` and `can()`.

The isa() method

As we've seen, the ancestry of an object can be very long and involved, and sometimes it can be rather tricky to determine if an object has inherited from a certain class. Looking through an object's `@ISA` array is a naive approach, but doesn't let us examine grandparents or great-grandparents.

As you can imagine, checking all the way up the class hierarchy is far from trivial. Luckily for us, there's a *universal method* called `isa()`, which we can use to determine if an object *isa* member of a particular class.

```
my $trainer = PerlTrainer->new(name => "Paul Fenwick");

print "Paul is a geek.\n"         if $trainer->isa("Geek");
print "Paul is a hairdresser.\n"  if $trainer->isa("HairDresser");
```

Assuming the hierarchy for the `PerlTrainer` class that we discussed before, this will print that Paul is a geek, but not mention hairdressers at all.

The `isa()` method caches its return values, so if you change inheritance of a class that has objects in existence that you've already called `isa()` on, then you might get unexpected results. Of course, if you're changing the inheritance of classes at run-time, you should be expecting the unexpected. :)

The `isa()` method can be found in more detail on pages 178-179 of Conway's book.

The can() method

The other universal method that we'll talk about is `can()`, which tells us whether or not a particular object *can* call the method supplied.

A common use of `can` is to call a method only if it exists. For example, an object might have a `display` method that prints its contents in a human readable form. Given a list of objects we want to print, we could do this:

```
foreach my $obj (@list) {
        if ($obj->can("display")) {
                $obj->display;
        } else {
                print $obj;
        }
}
```

Or, more concisely:

```
foreach my $obj (@list) {
        $obj->can("display") ? $obj->display : print $obj;
}
```

The `can` method has more uses than you think. It's particularly useful because it returns a reference to the method if it exists. This makes it handy if you need a particular functionality but you're not certain what it may be called on the object you're dealing with. In the example below, we search through a series of likely methods for converting our object into a string of suitable form for saving in a file or handing to another process.

```
# Convert our object into a string for storage.

my $freezer = $obj->can("freeze")    ||
              $obj->can("store")     ||
              $obj->can("serialise") ||
              $obj->can("serialize") ||
              die "Cannot serialise object\n";

my $frozen_obj = $obj->$freezer();
```

Of course we could use `Data::Dumper` to serialise our object. Data::Dumper stringifies perl data structures suitable for printing or "eval"ing later. You can learn more about Data::Dumper by checking out **perldoc Data::Dumper**.

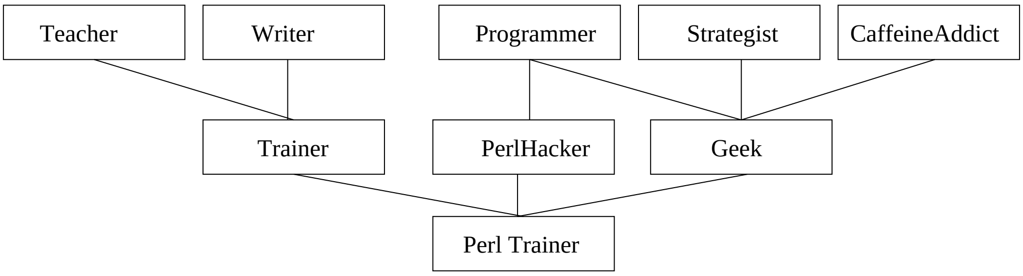
Problems with initialisers

Separating constructors from initialisers to let all the classes in an inheritance hierarchy have a chance at initialising the object is extremely useful. However it is not without traps for the unwary. We cover these next.

Initialisers and diamond inheritance

Let's take our `PerlTrainer` example again and extend it a little. A `PerlTrainer` isn't just any sort of `Programmer`, they're a `PerlHacker`.

Figure 10-2. Adding PerlHacker to the inheritance tree



Here we have two classes (`Geek` and `PerlHacker`) that share a common ancestor (`Programmer`). Now, this can cause some interesting problems, particularly at object creation time. Using the techniques that we've just discussed, the `Programmer`'s `_init` function would be called twice when a `PerlTrainer` is created. This *might* be okay, but what if the `_init` function keeps a record of how many `Programmers` get created? A `PerlTrainer` would end up getting counted twice! Fortunately, provided we are using hashes for our objects, a solution is close at hand:

```
package Programmer;

sub _init {
        my ($self, %args) = @_;
        return if $self->{_init}{Programmer}++;

        # remainder of initialisation...
}
```

The first time this is called, `$self->{_init}{Programmer}` does not exist, so the initialisation is run. The post-increment operator (`++`) ensures that the attribute gets created and set to a true value. The next time (if there is a next time) the initialisation is called, we can tell that we're experiencing a sort of inherited *deja vu*, and skip the initialisation that we've already performed.

Indeed, this code can be generalised even further to protect ourselves against mistyping our own package name (which can happen if you're dealing with long names like `Person::Employee::Technical::SysAdmin::UNIX::FreeBSD`), like this:

```
sub _init {
        my ($self, %args) = @_;

my $PACKAGE = __PACKAGE__;
        return if $self->{_init}{$PACKAGE}++;
        # ...
}
```

`__PACKAGE__` is a magic symbol that always evaluates to the current package, and is generally preferable to writing the package name itself. This is particularly the case if your code is likely to be cut'n'pasted, or if the package name might change, or if it's 3am in the morning with an important client demonstration the next day.

In the example above we copy the value of `__PACKAGE__` to a variable and use that in our hash lookup. This is because of Perl's rules about hash keys, which says that bare words in hash lookups are *always* assumed to be strings. We don't want Perl to look up the literal string `__PACKAGE__` in the hash but rather the result of evaluating `__PACKAGE__` first.

Damian Conway has an example on page 175 of his book which also illustrates this point.

Changing parents

It's possible that during the course of your class' development, you might end up inheriting from a few new classes, or dropping off some old ones. For example, in our last section we added `PerlHacker` as a parent for our `PerlTrainer` class.

Previously, when calling parent initialisers, we needed to list our parent classes *twice*. Once in our `@ISA` array, and once in our `_init` function. That's not a good thing, because as changes are made the two lists might get out of sync. We could end up calling an initialiser on an unrelated class, or forget to call one on a parent class.

Perl provides a special *pseudo-class* named `SUPER`, which signals to Perl's dispatch mechanism to look for the first available method above our current class, and call that:

Let's take the following example:

```
package PerlHacker;
use vars qw/@ISA/;

@ISA = qw/Programmer/;

sub _init {
        my ($self, %args) = @_;

        # Call my parent's _init function.
        $self->SUPER::_init(%args);

        # Class-specific initialisation.
        $self->{_perlmonks_level} = $args{pm_level} || 4;
        $self->{_modules_maintained} = $args{modules} || [];

        return $this;
}
```

Notice that we're using named parameter passing in this code. This is especially useful in inherited situations as we can use values from the parameters that we need and pass the rest to our parent constructors in case they can use them.

Note that using `SUPER` here acts differently than just an alias to `Programmer`. If a class has more than one parent, `SUPER` will search all of them in the regular depth-first, left-to-right fashion, until it finds the required method (or throws an exception if none can be found).

Unfortunately, `SUPER` only calls the first method it finds. So for any class that uses multiple inheritance, and wishes to call initialisers on all of its parents, `SUPER` just isn't suitable.

One way to get around this is to ignore `SUPER` entirely, and walk our `@ISA` array, and determine when we should be calling the method in question...

```
package PerlHacker;
use vars qw/@ISA/;

@ISA = qw/Programmer/;

sub _init {
        my ($this, %args) = @_;
```

```
        # Call my parents' _init functions.
        foreach my $parent (@ISA) {
                $parent_init = $parent->can("_init");
                $this->$parent_init(%args) if $parent_init;
        }


        # Class-specific initialisation.
        $this->{_perlmonks_level} = $args{pm_level} || 4;
        $this->{_modules_maintained} = $args{modules} || [];

        return $this;
}
```

That code needs a bit of explaining. You'll recall that `$parent->can("_init")` checks to see if the given parent (or one of *its* parents) can handle a call to `_init`, and if so, returns a reference to that method. We then call that method directly (using `$this->$parent_init(%args)`). Calling a method directly with a subroutine reference is very fast, and means we don't need to fire up the dispatcher a second time (once for `can()` and a second time for the method call itself). This also has the advantage that if a given branch of the ancestor tree doesn't have an `_init` function for whatever reason, we don't try to call it.

If you find the above is hard to grasp, or think that it's an awful lot of effort to do what *should* be a very simple thing, then you're right. And there is a better way which involves firing up the dispatcher mechanism where it left off. We'll talk about that next[1].

The SUPER package is discussed in further detail in Conway's book on pages 183 and 184.

What is a pseudo-class?

A pseudo-class is a class which cannot instantiate an object, and which should not be inherited. Rather, it exists so that it can be invoked for its side-effects. One use of pseudo-classes is to control the dispatch mechanism. We've seen the SUPER pseudo-class, but there are others such as NEXT which we'll be covering shortly.

The PerlTrainer class in full

Here's how all we've learnt so far fits together:

```
package PerlTrainer;

use vars qw/@ISA/;
use Trainer;
use PerlHacker;
use Geek;

@ISA = qw/Trainer PerlHacker Geek/;

# Constructor method.  Just creates the hash and then passes
# the work off to the initialiser method.
```

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

        $this->_init(@args);

        return $this;
}

# Initialiser method, does all the hard work.
sub _init {
        my ($this, %args) = @_;

        # Initialise the object for all of our base classes.

        foreach my $parent (@ISA) {
                $parent_init = $parent->can("_init");
                $this->$parent_init(%args) if $parent_init;
        }

        # Class-specific initialisation.
        $this->{_perl_courses} = $args{courses} || [];

        # Return the initialised object.
        return $this;
}
```

Exercises

1. Derive a `Coin::Weighted` class from the `Coin` class. These coins behave as regular `Coins`, however heads comes up 60% of the time instead of 50%.
2. Modify your statistics coin program to create one `Coin` coin and one `Coin::Weighted` coin (instead of 2 `Coin` coins). Run it and check that the statistics match what you expect.
3. Create a second coin program which creates a `Coin` object and a `Coin::Weighted` object.
    a. Use `isa` to check whether they are both `Coins`.
    b. Use `isa` to check whether they are both `Coin::Weighteds`.

       An answer for this can be found in `exercises/answers/isa.pl`.
    c. Are the results as you expect?
4. Add the following functions to your `Coin::Weighted` class:
    - `set_weight` which sets the amount of favour the coin shows to heads.
    - `get_weight` which returns the current value of favour the coin shows to heads.
5. Create 10 each of `Coin` coins and `Coin::Weighted` coin and put them into an array. Randomise the array using

   ```
   @array = sort {rand() <=> 0.5} @array;
   ```

and then walk over it setting the weight of each of the Coin::Weighted coins to 90%.

Use the `can()` method to ensure you don't call `set_weight` on a `Coin` coin and then use the returned subroutine reference to call the function.

An answer for this can be found in `exercises/answers/can.pl`.

6. Since we can now set the value of our `Coin::Weighted` coin's weight you will have had to make a decision as to how that weight is initially set. Pull this initialisation that you've done out into an `_init` function and change your `Coin` constructor function to call `_init` on the object if that method exists.
7. Your `Coin::Weighted` class should no longer need to have a separate constructor function. Remove this if you've created one.

Chapter summary

- Inheritance (in Perl) is nothing more than a way of specifying where to look for methods.
- When looking for a method called on our object that that object does not define, Perl will do a depth-first, left-to-right search of the tree of ancestors.
- We can instruct Perl where to start its search by qualifying a method with a parent (or other) class name, for example `$trainer->Teacher::review()`.
- To ensure that our code is easy to inherit from we ought to do our initialisation for our object inside a separate initialise function. This is called `init` by convention.
- The `isa()` method automatically exists on all objects and allows us to determine whether that object is a member of a particular class.
- The `can()` method also automatically exists on all objects and allows us to determine whether that object can call a given method.
- If we want to call each of our parent constructors for our object we can loop through our `@ISA` array.
- The SUPER pseudo-class tells Perl to look for the first available method above our current class and call that.

Notes

1. Pun not intentional

Chapter 11. Redispatching method calls

In this chapter...

In the `PerlTrainer` example in the previous chapter, we demonstrated how Perl's dispatch mechanism might find the method `review`. In this chapter we look at what happens if it finds the wrong one.

Pass it on please

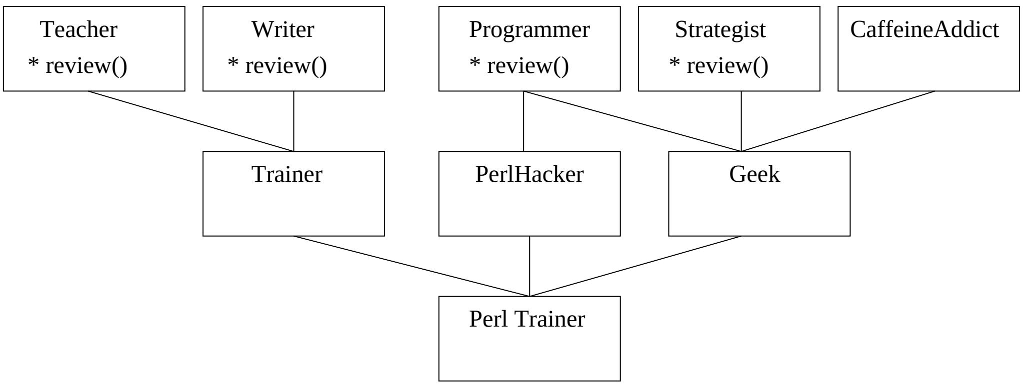
What happens if a number of ancestors have the method that's just been requested, and we end up calling the wrong one? For example:

```
my $person = PerlTrainer->new(name => "Paul Fenwick");

# Get our person to review some Perl code.
$person->review(language => "Perl",program=>"hello.pl");
```

Let's suppose that all of the `Teacher`, `Writer`, `Programmer` and `Strategist` ancestors provide a `review` method. In this case, the method from `Teacher` will *always* be called, due to the "left-most ancestor wins" rule. It's fairly obvious that we wanted the method from `Programmer`, but Perl has no way of knowing that.

Figure 11-1. Classes providing the review method



Now, we could explicitly tell Perl which method we meant, by qualifying it with a classname:

```
$person->Programmer::review(language => "Perl", program => "hello.pl");
```

But as a user of the class we shouldn't need to know nor care what the inheritance structure of a `PerlTrainer` is. In fact, it would be very bad if we did this, since `PerlTrainer` might change at some point to provide a much more appropriate method than the one inherited from `Programmer`.

In cases like this, when a class receives a method call that was obviously meant for someone else, or when we want to see what other members of the hierarchy might think, there's a way to drop back into the message dispatch mechanism and call the next method along. We need to use a special module to do this, and that module is unsurprisingly called `NEXT`.

```
use NEXT;
package Teacher;

sub review {
        my ($this, %args) = @_;

        unless ($args{assignment} or $args{exam}) {
                # Gosh darn, the method dispatcher gave us the call
                # that was meant for someone else.  Throw it back.

                return $this->NEXT::review(%args);
        }

        # Review our literature here.
}
```

The `NEXT` module defines a *pseudo-class* which allows message dispatch to continue on where it left off. In the case of our `PerlTrainer` example, the dispatch mechanism would trek back down the `Trainer` branch of the ancestor-tree, and up the `Writer` class, which also defines a `review` method, which is then called.

The `Writer` class can also choose to re-invoke the method dispatcher with another call via the `NEXT` pseudo-class. In this case, the dispatcher would backtrack to the `Trainer` class again, down even further to `PerlTrainer` and then up through `PerlHacker` to `Programmer`.

If `Programmer` uses `NEXT` to pass on the method, we backtrack down to the `PerlTrainer` class again, and then back up to `Programmer` a *second* time via the `Geek` class.

Hmm, that makes sense in a way, but isn't what we want in this (or many other) situations. Luckily, `NEXT` has a way for us to specify that we should skip over methods that we've already seen:

```
                return $this->NEXT::DISTINCT::review(%args);
```

If we use `NEXT::DISTINCT` then the redispatch mechanism would skip over the `Programmer` class (which it had already seen before), and land the call into the `Strategist` class.

You should always use `NEXT::DISTINCT` unless you're sure that you want parent methods that are multiply-inherited to be called multiple times.

Exercises

1. Change your `PerlTrainer` classes from your `exercises/lib` to superficially distinguish between review calls. For example `Teachers` may expect both "student" and "paper" as arguments, whereas `Writers`s my expect "novel", or "book", or "whole_lifes_work", and so on. Use `NEXT` in each class to make sure that inappropriate calls are passed on.
2. Write a script that uses the `PerlTrainer` classes and makes calls to review various things. Check that the call is getting through to the appropriate class.
3. What happens if you call the review method with arguments that none of the parent classes expect?

Optional redispatch

So, what happens if `Strategist` decides to fire up the redispatcher and send the call on its way? There's no class after `Strategist` which can handle a call to `review`. Does Perl throw an exception or something?

No. It goes away. Since nobody wants the method call, Perl arranges for it to return the undefined value (or the empty list, in list context) and silently leaves it at that.

Isn't that bad? We asked the dispatcher to pass the method on, and it just ignored it? You're going to tell me that's a feature, right? Yup, that's most certainly a feature.

You see, one of the best uses of the `NEXT` pseudo-class is in initialisers and destructors, where we want to call our parent(s) methods and then add a little bit of our own work. If our parents don't have the methods to call, we want to ignore that and continue, rather than bail out.

So, we can replace the rather ugly:

```
# Call my parents' _init functions.
foreach my $parent (@ISA) {
        $parent_init = $parent->can("_init");
        $this->$parent_init(%args) if $parent_init;
}
```

with the much more elegant:

```
$this->NEXT::DISTINCT::_init(%args);
```

Not only is that shorter, it's *much* more clear about what's needed. It also avoids the problems of initialisers being called twice in the case of multiple inheritance. In destructors it's just as easy:

```
sub DESTROY {
        my ($this) = @_;

        # Do my own clean-up here.

        $this->NEXT::DISTINCT::DESTROY;

}
```

It's difficult to recommend `NEXT` enough for this sort of work.

You can read more about the `NEXT` pseudo-class by using **perldoc NEXT**.

Mandatory redispatch

Back to our original example, with the `review` method. Here we want to pass on the method call, but if nobody else is willing to take it we want to complain loudly. Having the code below failing silently is probably not acceptable.

```
my $trainer = PerlTrainer->new(name    => "Paul Fenwick",
                               drinks  => [qw/coffee tea cola/]);

$trainer->review(quilt => $quilt_pattern);
```

`PerlTrainers` usually know nothing about quilting [1], so rather than this method call fall into a hole and disappear, we'd like it to throw an exception that it couldn't do the required task. Enter `NEXT::ACTUAL`.

`NEXT::ACTUAL` works identically to `NEXT`, except that it throws an exception if we try redispatching when no further methods exist to try the call against.

Let's see it in action:

```
use NEXT;
package Writer;

sub review {
        my ($this, %args) = @_;

        unless ($args{book} or $args{notes} or $args{article}) {
                # Gosh darn, the method dispatcher gave us the call
                # that was meant for someone else.  Throw it back.

                return $this->NEXT::ACTUAL::review(%args);
        }

        # Review our literature here.
}
```

And yes, it's possible (and recommended) to use both `NEXT::ACTUAL` and `NEXT::DISTINCT` together:

```
use NEXT;
package Writer;

sub review {
        my ($this, %args) = @_;

        unless ($args{book} or $args{notes} or $args{article}) {
                # Gosh darn, the method dispatcher gave us the call
                # that was meant for someone else.  Throw it back.

                return $this->NEXT::DISTINCT::ACTUAL::review(%args);
        }

        # Review our literature here.
}
```

Yes, `NEXT::ACTUAL::DISTINCT` works as well (and behaves exactly the same) as `NEXT::DISTINCT::ACTUAL`.

Exercises

1. Add mandatory dispatch to your review methods. What happens now if you call the `review` method with arguments that none of the parent classes expect?

Problems with NEXT

Unfortunately, `NEXT` isn't the solution to all our problems. The most common issue you will experience with `NEXT` is when you're working with third-party classes. Proper operation of `NEXT` relies upon each method (particularly constructors/destructors) invoking `NEXT` at the appropriate point. If you're inheriting from a third-party class that doesn't do this, then you have a problem.

If you're only inheriting from a single `NEXT`-ignorant class, then making that your rightmost ancestor may solve your problem, as there will be no requirement for it to try and re-dispatch any method call outside of its own inheritance tree.

Another issue that arises with `NEXT` is that in the case of diamond-inheritance, it's possible for child initialisers to be called before their parents. If this causes problems, then it can be avoided by using the following code construct:

```
sub _init {
        my ($self,@args) = @_;
        $self->NEXT::_init(@args);
        my $package = __PACKAGE__;
        return if $self->{_init}{$package}++;
        # ... initialisation occurs here.
}
```

It's important to use regular `NEXT` and not `NEXT::DISTINCT` for this to work correctly.

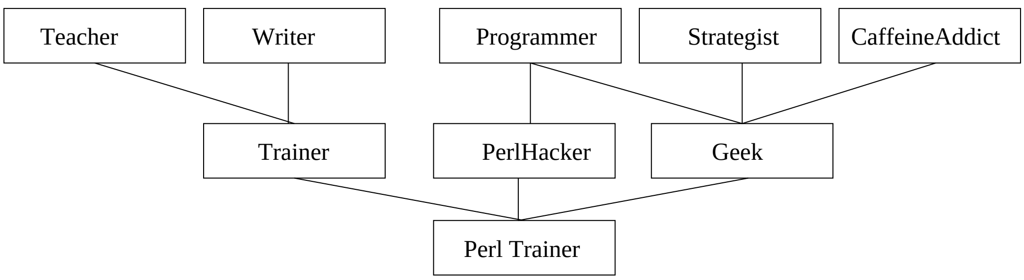
More recent versions of the `NEXT` module provide a new pseudo-class to overcome these problems, called `EVERY`. We'll cover this new pseudo-class in the next section.

Using EVERY to call all methods

The `NEXT` pseudo-class is most useful when we want methods to have the ability to redispatch a method call. However it has some shortcoming when we wish to use it for initialisation and destructors, which the ordering of method calls can also be quite significant, and we wish to process methods in a strict 'parent before child' or 'child before parent' order.

In order to accommodate these situations, Perl has the `EVERY` pseudo-class, for when we wish to call every method in a class hierarchy, rather than just the next one. `EVERY` does not use the 'leftmost ancestor' routine of `NEXT` and the in-built Perl dispatch mechanism. Instead `EVERY` works on a breadth-first search. Let's see an example using our `PerlTrainer` class:

Figure 11-2. The PerlTrainer hierarchy



Using `EVERY` would result in classes being called in the following order:

- `PerlTrainer`
- `Trainer`
- `PerlHacker`
- `Geek`

- `Teacher`
- `Writer`
- `Programmer`
- `Strategist`
- `CaffeineAddict`

Of particular note is that `Geek` is called before `Programmer`, even though `Programmer` appears 'first' in the `PerlTrainer` inheritance hierarchy. `EVERY` guarantees that all child classes will be called before their parents.

Ensuring that child classes are called before parents is very useful for destructor methods, where usually the derived class needs to do tidy-up before its parents. However what of the case of constructors, where we want parent classes to do initialisation first? For this, we use the `EVERY::LAST` pseudo-class.

`EVERY::LAST` will call every method in a given inheritance hierarchy, but in the reverse order to `EVERY`. As such, parents are guaranteed to be called before their child classes.

Using EVERY and EVERY::LAST in practice

When using `NEXT`, each method either begins or ends with a call to the next method. However when using `EVERY` and `EVERY::LAST`, a single call executes all the methods in a given hierarchy. As such, the use of `EVERY` and `EVERY::LAST` requires a different, and often simpler, approach to coding.

Constructors

Let's consider the humble constructor. In our constructors we usually want to ensure that all of our parent classes do their initialisation first before we do our own. This allows us to overwrite values that our parents have set rather than the other way around. To achieve this our constructor will often look like this:

```
sub new {
        my ($class,@args) = @_;
        my $this = bless({},$class);
        $this->_init(@args);    # Call my _init method.
        return $this;
}

sub _init {
       my ($this, %args) = @_;
       # Initialise my parents
       $this->SUPER::_init(%args);      # or walk through @ISA

       # Perform my initialisation here.
}
```

Unfortunately, in the case of diamond inheritance, this can mean that a parent is initialised twice, once for each child. Using `NEXT::DISTINCT` doesn't help either as it can result in a child class doing its initialisation prior to one of its parents.

When using `EVERY::LAST` only a single call is made, from the constructor itself:

```
use NEXT;

sub new {
        my ($class,@args) = @_;
        my $this = bless({},$class);
        $this->EVERY::LAST::_init(@args);       # Call every _init
        return $this;
}

sub _init {
       my ($this, %args) = @_;

       # No need to call my parents.

       # Perform my initialisation here.
}
```

The call to `EVERY::LAST` guarantees that every `_init` method will be called, starting with the parent classes and then moving to the children. No child class will be called before all of its parents are called, and each method will only be called once.

Destructors

In our destructors we usually want to ensure that the child classes' destructors are called prior to their parents. This allows the child class to write out their changes to the database before the parent disconnects, for example.

When we call a method via the `EVERY` pseudo-classes this method is called on each parent class as well as the current class, if it exists. As a result, in the previous example, we were able to call the `_init` method for our class without having to explicitly specify it. However this means that we don't want to call our `DESTROY` method via a call to `EVERY::DESTROY` as this would result in our `DESTROY` method calling itself (and its parents) in an infinitely recursive loop.

The easiest way to solve this issue is simply to have a single, inherited `DESTROY` method, which dispatches the call to methods that do all the hard work, but have a different name:

```
use NEXT;

sub DESTROY {
        my ($this) = @_;
        $this->EVERY::_destroy;
}

sub _destroy {
        my ($this) = @_;
        # All the real clean-up occurs here.
}
```

Each parent class now defines its own `_destroy` method (if required) instead of a `DESTROY` method. The `DESTROY` method is defined in a single class and ensures that all `_destroy` methods are called appropriately.

Exercises

1. Modify your `exercises/lib/PerlTrainer.pm` so that the `PerlTrainer` class has a `call_test` which calls `EVERY::test`.

2. Write a program which instantiates a `PerlTrainer` object and calls its `call_test` method.
3. Either modify your `call_test` method to use `EVERY::LAST` or add an additional method which calls every `test` method in reverse.
4. Call this method on your `PerlTrainer` object.

Chapter summary

- Perl's redispatcher always calls the first method it finds of the requested name, in its depth-first, left-to-right search.
- If we wish to ask Perl to find the next method by that name we can use `NEXT`.
- If we want to insist that Perl find another method or die we can use `NEXT::ACTUAL`.
- If we want to avoid calling a function twice due to diamond inheritance we can use `NEXT::DISTINCT`.
- Unfortunately `NEXT` doesn't solve all of our problems with multiple inheritance in Perl
- `EVERY` provides a way of calling a number of methods in one go. It overcomes the problems associated with `NEXT` when dealing with constructors and destructors.

Notes

1. Actually, some `PerlTrainers` know quite a bit about quilting. See this PerlMonks node (http://perlmonks.org/index.pl?node_id=72270) for that knowledge put to good use.

Chapter 12. Abstract Classes

In this chapter

This chapter discusses abstract classes, where they're useful and how to create them.

Abstracting

Sometimes you'll encounter a situation where it's advantageous for many different objects to share a similar behaviour, but this common behaviour does not constitute a proper object in itself. In this case, we want abstract classes. These classes can be inherited from, but not instantiated into objects.

Let's take an example to demonstrate. You should all be familiar with the game of chess. All the pieces share common attributes, such as their colour and position, and common behaviours like being able to move and take. However there's no such thing as a generic chess piece. We can write an abstract chess piece class like this:

```
package Games::Chess::Piece;
use Carp;
use NEXT;

# can_move, can_take, and get_name must be over-ridden by the
# child class.

sub can_move {
        croak "Abstract method called";
}

sub can_take {
        croak "Abstract method called";
}

sub get_name {
        croak "Abstract method called";
}

# Both move() and take() will need to be updated when we understand
# how our pieces fit together on the board.  Currently they're unaware
# of other pieces.

sub move {
        my $this = shift;
        my $location = shift;
        unless ($this->can_move($location)) {
                croak "Cannot move ".$this->get_name()." to $location";
        }
        $this->set_location($location);
}

sub take {
        my $this = shift;
        my $location = shift;

        # We need a check here to ensure an opposing piece is in the
        # location specified.

        unless ($this->can_take($location)) {
```

```
                croak $this->get_name()." cannot take piece at $location";
        }
        $this->set_location($location);

        # Do whatever is needed to make the opposing piece disappear.
}

sub get_location { return $_[0]->{_location}; }
sub get_position { return $_[0]->get_location(); }
sub get_colour   { return $_[0]->{_colour}; }

# For our American friends...
sub get_color    { return $_[0]->get_colour(); }

# set_location really should ensure that the location is actually on
# the board at the very least
sub set_location {
        my ($this, $location) = @_;
        $this->{_location} = $location;
}

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

        bless($this,$class);

        $this->_init(@args);

        return $this;
}

# The _init function does specific initialisation for this class.
sub _init {
        my ($this, %args) = @_;

        $this->NEXT::UNSEEN::_init(%args);

        # Naively assign colour and location attributes.
        # These should be checked for validity.
        $this->{_colour}   = $args{colour}   || $args{color};
        $this->set_location( $args{location} || $args{position} );

        return $this;
}
```

Now, let’s look at that in more detail, shall we? The first three methods the class defines, `can_move`, `can_take` and `get_name` simply return errors. That doesn’t seem like a particularly useful thing to do when someone tries to move a piece, is it? The reason we do this (as the comments suggest) is that these methods are place holders for child classes to override with something more useful.

Such place holders are called *abstract methods*. In our case, every chess piece has different rules on moving and taking, so while we want to make sure these methods exist, we also want to make sure they’re defined properly for the piece at hand. When we have a class that we want others to inherit, but shouldn’t be used to create objects in its own right, we call it an *abstract class*.

Now, let’s derive a class from this abstract one that we’ve already built.

```
use Games::Chess::Piece;

package Games::Chess::Piece::Rook;

use vars qw/@ISA/;
```

```
@ISA = qw/Games::Chess::Piece/;

# Check to see if the piece can move to a particular location.
# This doesn’t currently check for intervening pieces.
sub can_move {
        my ($this,$newloc) = @_;
        my $oldloc = $this->get_location;

        # Rooks can move along files (columns, for non-chess players)...
        return 1 if (substr($oldloc,0,1) eq substr($newloc,0,1));

        # ...and rows.
        return 1 if (substr($oldloc,1,1) eq substr($newloc,1,1));

        return 0;
}

# Rooks take the same way that they move.  Only pawns have odd
# behaviour here.  Note this doesn’t check to ensure we’re taking
# a piece of the opposite colour.
sub can_take { return shift()->can_move(@_); }

sub get_name { return "Rook" };
```

As you can see, we only needed to define the `can_move`, `can_take` and `get_name` methods to build ourselves a rook class. The creation of the piece and common functions to check its location and colour have been handled for us by the parent.

You may have noticed that we found the rook’s location by calling `$this->get_location`, rather than accessing it directly with `$this->{_location}`. Why was that? Surely it’s faster to fetch the value directly from the hash, rather than going through a method call. Well, it is, but there’s a price to pay for it. As long as we call the `get_location` method, the internals of how that information is stored can change, and provided the method returns the same information we don’t need to worry about it. Imagine if we wanted to store the location packed into a single byte for more compact storage -- we’d need to update each and every piece [1] (knights, rooks, kings, queens, pawns and bishops) and change every instance where we accessed the location to instead unpack that byte into an appropriate form. Object oriented methodology doesn’t just exist to protect the users of a class, it exists to protect the writers of a class as well.

The *Class::Virtual* and *Class::Virtually::Abstract* classes can be used to automate the creation of virtual methods.

You can read more about these classes using **perldoc Class::Virtual** and **perldoc Class::Virtually::Abstract**.

Now that we have all these chess pieces to play with, we can go on to our next topic, which is *polymorphism*.

We can force a class to be abstract by not providing a constructor method for that class. Note that this is not the same as not providing initialisation, since that may be essential. If no constructor method exists for a class then a user of the class must explicitly bless their object into the class themselves. Hopefully they’ll think about that first.

In our example, we *do* provide a constructor, as it saves us needing to write a constructor for each child class. The primary purpose of an abstract class is to provide useful functionality to any classes that inherit it.

Group Exercise

1. Earlier we described airplane modelling. Of the classes you defined which were abstract? Which methods were abstract?

Chapter summary

- Abstract classes are usually set up to ensure that their child classes definitely have a particular interface.
- Abstract methods ought to be overridden by each child class (and bad things should happen if they are not).

Notes

1. Chess purists will no doubt complain that pawns are pawns, not pieces. However you shouldn't change your code just because a chess purist tells you to.

Chapter 13. Polymorphism

In this chapter...

As we discussed in the introduction, *polymorphism* is the ability for objects to react differently to the same message, depending upon their class. There are many instances where polymorphism is useful. For example, we may be managing a fleet of vehicles, and the form to print when requested to `print_registration_form` is likely to be different for a motor-bike compared to a tow-truck.

Using polymorphism

Let's take our chess pieces that were introduced in the last chapter, as they are an excellent example of where polymorphic behaviour is useful. When we try to move a piece, we want to be told if that move is valid or not, and the way in which a piece can move varies from piece to piece. Rather than having to use different methods depending upon the piece we're dealing with, (`can_move_bishop()`, `can_move_rook()`, etc) we can use the same method call, and trust the piece to do the right thing. Let's see an example:

```
use Games::Chess::Piece::Rook;
use Games::Chess::Piece::Bishop;

my $rook = Games::Chess::Piece::Rook->new(colour=>"white",location=>"a1");
my $bishop = Games::Chess::Piece::Bishop->new(colour=>"black",location=>"h1");

my @pieces = ($rook, $bishop);

foreach my $piece (@pieces) {
        print "The ",$piece->get_name()," at ",$piece->get_location()
        if ($piece->can_move("a5")) {
                print " can move to a5\n";
        } else {
                print " cannot move to a5\n";
        }
}
```

That's a somewhat contrived example, but it shows off polymorphism very well. There are three separate places where polymorphic behaviour was used. `$piece->get_name()`, `$piece->get_location()`, and `$piece->can_move()`. If you think they look just like regular method calls, then you're absolutely right.

Inheritance vs interface polymorphism

Broadly speaking, there are two main types of polymorphism. What we've seen so far is an example of *inheritance polymorphism*. All of our chess pieces share a common ancestor, and so we know that they all share a common set of methods (such as `get_location()` and `can_move()`) which that ancestor class defines.

What happens if we want polymorphic behaviour with objects that don't share anything in common? This is an instance of *interface polymorphism*. Our objects aren't related, but they all share some common interface which allows them to be treated in a polymorphic way.

Sometimes it's important to know if an object has a particular method. To do this, you'll want to cast your mind back to the `can()` universal method, which exists on all objects.

```
# Check if our object can refresh itself on the screen.

if ($obj->can("refresh")) {
        $obj->refresh;
} else {
        # Refresh the object manually....
}
```

In Perl, there's no requirement that your classes declare allegiance to a particular interface specification to be polymorphic, it just has to declare the appropriate method that it's expected to provide.

Adding default methods and the UNIVERSAL class

Sometimes it would be nice to have a method exist for all of our objects, and perhaps for some objects we didn't write. We can (and should) write these methods into our classes, but we may not be able to change the sources of classes we don't own. Fortunately for us, all objects inherit from the `UNIVERSAL` class. Which is why we are able to call the universal methods `can` and `isa` on them.

This means that we can add default methods to the `UNIVERSAL` class, if necessary, and be confident that all objects will now have access to that method. For example:

```
sub UNIVERSAL::to_string
{
        return $_[0];
        # or if we've used Data::Dumper
        # return Dumper($_[0]);
}

        # print out my object types all my objects
foreach my $object ($gardener, $pitchfork, $shovel, $car, $cat, $dog)
{
        print $object->to_string();
}
```

If the object has it's own `to_string` method then that will be called in preference to the `UNIVERSAL` method. If it does not, we can feel certain that we won't receive any run-time errors, as the `UNIVERSAL` method will be used instead.

More on inheritance polymorphism

The most common form of polymorphism is via inheritance, and this warrants a little further discussion as Perl's approach may differ from other object oriented languages that you've used in the past.

In Perl, when a method is called upon an object, it is always dispatched according to the class to which the object belongs, not the class of the current subroutine. This means that if you have a `Chess::Piece::Bishop` object, then `$bishop->get_name()` will always start searching in the `Chess::Piece::Bishop` class.

There are some object oriented languages (such as C++) where in some instances the object is treated as being in one of its base classes, regardless of its actual type. If you want this behaviour in Perl you can have it:

```
$bishop->Chess::Piece::get_name()
```

Using the directed method syntax above (which was covered in the chapter on *Inheritance*), we start the dispatch in the `Chess::Piece` class (which in this case will almost certainly generate an exception about invoking an abstract method).

Exercises

1. Write a script that creates both a `PlayingCard` object and a `Coin` object. In your script define a `UNIVERSAL to_string` subroutine which uses `Data::Dumper` to print out the content of that object. Call this subroutine on both objects.
2. Add a separate `to_string` subroutine in `PlayingCard` which prints out the card's value and suit. Call `to_string` on both objects and see what happens.
3. Add a `to_string` subroutine in `Coin` which prints out the coin's current state. Call `to_string` on both objects and see what happens. These are instances of interface polymorphism.

Chapter summary

- Polymorphism is the term for different classes behaving in different ways when given the same message.
- Inheritance polymorphism is where a set of classes have the same interface because they all share a common ancestor.
- Interface polymorphism is where a set of classes have the same interface because they have agreed to.

Chapter 14. Practical Exercise - the Game of Chess

Required reading

The game of chess has simple and easy-to-understand rules. Although the strategy involved in the game can be quite complex, it is possible to learn the basic rules with only a few minutes of reading. We're going to use this game as the basis of a number of practical exercises using our Object Oriented Perl knowledge.

The basic rules of chess are explained very clearly on the website of the US Chess Federation, in their "Let's Play Chess" (http://www.uschess.org/beginners/letsplay.php) primer. At the very least you will need to know the names of the pieces and how they move, so take a few minutes to read over this information now.

For these exercises, we will use *algebraic notation* to denote the location of the pieces on the board. Algebraic notation assigns every file (column) a number between *a* and *h*, and every rank (row) a number between 1 and 8. A good introduction to algebraic notation can be found on the US Chess Federation's "How to Read and Write Chess" (http://www.uschess.org/beginners/read/) page. You may wish to take a moment to read it now.

We'll actually be using a simplified version of algebraic notation for these exercises. Rather than writing a shorthand involving just the piece and final location (eg, Bc4 - Bishop to c4), we'll instead just list both the starting and ending squares for the move (eg, f1-c4 - the piece at f1 moves to c4).

Group Questions

In this exercise we will create a number of related classes that implement a set of chess pieces. We would like our pieces to be able to know their name, colour, and position. We'd also like our chess pieces to be able to tell if they can move to, or take a piece in, a particular square.

1. Chess pieces have a co-ordinate in two dimensions, their *row* (rank) and *column* (file). Discuss the best way to store this information.
2. There will be some behaviour which is common to all chess pieces. For example, we should be able to ask any piece for its colour or location. Discuss what methods we might want common to all chess pieces. In what way might we guarantee that these methods exist on all pieces?

   Determine what these methods will be called, what arguments they'll take and in which order.
3. Consider further methods that will make the chess pieces more usable. For example, rather than just `get_name` to get a piece's name, would it be useful to also have a method which reports a piece's name, colour, and location?
4. There are six different types of chess pieces (rooks, knights, bishops, kings, queens and pawns). We'll be implementing these pieces as part of the remaining exercises. Volunteer to implement at least one of these pieces for the group. We'll make sure that everyone in the group will be working on at least one piece.

Individual Exercises

You may do this section in pairs if you desire.

1. Create a `Chess::Piece` abstract class, and make sure that it implements all the virtual methods that were decided upon in the group exercises above. Your trainer may provide you with a starting point. Verify the class doesn't generate any errors when run with **perl -wc Piece.pm**.
2. Create a `Chess::Piece::Rook`, or one of the pieces you volunteered to create for the group. Make sure it inherits from `Chess::Piece`. Use the `chess-tester.pl` program that your trainer will supply to test that the piece can be created, moved, and displayed.
3. Update the `chess-tester.pl` program to create two or more different types of pieces, and let the user take turns in moving them about the board.

Group Discussion

1. Are there any problems with how the pieces currently behave? Why is this? How might they be solved?
2. Let's say that we create a `Chess::Board` class, that implements a chess board which can have pieces. What sort of relationships need to exist between the pieces and the board? Does this solve any of the problems we've discovered above?

Chapter 15. Operator overloading

In this chapter...

In this chapter we will briefly discuss Perl’s operator overloading mechanism, and how we can use it improve code readability and extend the usefulness of objects we create.

This topic is covered in much greater detail in Chapter 10 of Damian Conway’s book (Object Oriented Perl), or by using **perldoc overload**.

What is operator overloading?

Operator overloading is the process of taking standard arithmetic, comparison, and other operators, and changing their behaviour to act differently based upon the objects they are dealing with.

Operator overloading has the potential to make programs easy to read and write, and provide concise and intuitive ways of manipulating objects. For example, if we had a class which represented numbers in Roman Numerals, it would make perfect sense to be able to perform all the regular arithmetic operations on those objects.

On the other hand, operator overloading can turn your program into an incomprehensible minefield of obscure errors and unexpected problems. Overloading `eq` so that we can write:

```
if ($card eq "hearts")
```

rather than:

```
if ($card->get_suit eq "hearts")
```

may seem quite intuitive, but overloading `cos` to mean "cut once shuffled" is certainly not.

Perl allows you to overload a great many things, including things that you may not expect, like constants. This chapter will show you how to overload a few simple operators. It is not a complete guide to operator overloading.

Overloading stringification

The most useful operator to overload is Perl’s *stringification* operator, commonly written as `q{""}` (or more perversely as "\"\""). This isn’t a real operator per se, rather it’s an operation that is performed whenever your object gets used in a string context, such as being used as a hash-key, being printed, being concatenated, or having a string comparison (`eq`, `le`, `ge`, etc) operator applied.

Without overloading the stringification operator, Perl objects are just plain ugly (and unhelpful!) when they’re printed. For example, one of our chess-pieces when printed might produce this:

```
Chess::Piece::Bishop=HASH(0x80f62ac)
```

While it’s correct that we have an object of the specified type, and it is built upon a hash, that’s not particularly useful to most mortals. Wouldn’t it be better if instead it would print:

```
black bishop at e3
```

We can do all this (and more) using Perl's `overload` pragma. Here's how:

```
package Chess::Piece;

# Overloading is inherited, so we only need to define this on
# our base, abstract Chess::Piece.

use overload (
q{""} => "as_string",
);

sub as_string {
        my ($this) = @_;

        return join(" ", $this->colour, $this->name, "at", $this->location);
}
```

The `overload` pragma takes a list of directives, in the form of operator and method pairs. You will have noticed that we wrote the method name as a string. Since operator overloading is inherited by subclasses, specifying the name as a string indicates to Perl that it should search the class hierarchy for an appropriate method. If we specified the method as a subroutine reference, that subroutine would be invoked directly.

In our example above, whenever we used the chess-piece as a string (including when printed, concatenated, or used as a hash-key), its `as_string` method would be called, and the result of that used as the string.

Inheritance and overloading

There are two ways to provide Perl with methods that are used in overloads. If a string is passed to the `overload` pragma, then Perl looks for a method with that name, starting on the child class and working a leftmost-ancestor wins fashion. This is the preferred way to specify overloads, as it means that a child overriding a parent method does so for both regular and operator-overloaded calls to that method.

It is also possible to provide Perl with a *subroutine reference* to the code to be executed for an overloaded operator. Because this is a code reference, the `overload` pragma cannot tell if it refers to a normal method or an otherwise anonymous subroutine. The result of this is that if child classes want to override the method called for these operators, they must invoke the `overload` pragma again.

Where possible, it's recommended that methods to be used for overloaded operators always be passed by *name*, as this provides the most consistent and useful functionality to child classes.

```
#----------------------------------------
package A;

use overload (
        q(-) => "minus",
        q(*) => "multiply",
        q(+) => \&plus,       # This is a subroutine reference.
);

sub minus    { return $_[0]->{value} - $_[1] }
sub multiply { return $_[0]->{value} * $_[1] }
sub plus     { return $_[0]->{value} + $_[1] }
```

```
#--------------------------------------
package A::B;
our @ISA = qw/A/;               # A::B inherits from A

$self - 2;                      # Calls A::B::minus
$self * 2;                      # Calls A::B::multiply
$self + 2;                      # Calls A::plus

sub minus    { return $_[0]->{b_value} - $_[1] }
sub multiply { return $_[0]->{b_value} * $_[1] }
sub plus     { return $_[0]->{b_value} + $_[1] }

#--------------------------------------
package A::C;
our @ISA = qw/A/;               # A::C inherits from A

use overload (
        q(+) => "plus",         # I want to use my own plus method
);

$self - 2;                      # Calls A::C::minus
$self * 2;                      # Calls A::C::multiply
$self + 2;                      # Calls A::C::plus as explicitly requested

sub minus    { return $_[0]->{c_value} - $_[1] }
sub multiply { return $_[0]->{c_value} * $_[1] }
sub plus     { return $_[0]->{c_value} + $_[1] }

#--------------------------------------
```

If a method is overloaded in several ancestors then the usual inheritance rules work and the left-most ancestor wins.

Exercises

1. Add an overloaded `q{""}` method to your `PlayingCard` class. Have this print out the card's value and suit.
2. Create a deck of cards and use this new overload to print out the card objects without explicitly calling the subroutine.

Overloading comparison operators

The conversion operators (such as `q{""}` above) are invoked with only a single argument, being the object that requires conversion. Most operators, however, are binary operators with two operands, both of which are passed to the required method when that particular operator is used.

In fact, the method receives three arguments -- the object itself, the second operand, and whether or not the object and operand were reversed. The last argument is needed because methods always receive their object first, and we need to be able to distinguish between:

```
if ($obj < 2) { ... }
```

and

```
if (2 < $obj) { ... }
```

which obviously have very different meanings.

The subroutine which handles the overloaded method is expected to return a value that is appropriate to the operator in question. In the case of simple comparison operators, this is just a simple true/false value. In the case of the <=> and `cmp` operators, it is expected to be 1, 0, or -1, depending upon if the first operand is greater than equal to, or less than the second operand respectively.

Let's look at overloading the <=> operator for our `PlayingCard` class.

```
package PlayingCard;
use Carp;

use overload (
                q{""}       =>      "as_string",
                "<=>"      =>      "compare"
             );

sub compare {
        my ($this, $that, $reversed) = @_;
        unless (UNIVERSAL::isa($that, "PlayingCard") {
                croak("Attempt to compare card to non-card");
        }

        ($this,$that) = ($that,$this) if $reversed;

        return ($this->{value} <=> $that->{value});
}
```

As you can see, writing an overload method for a comparison operator isn't that hard. However, there are a *lot* of comparisons in Perl (fourteen, to be exact), and writing a method for every one gets very tedious very quickly. Luckily for us, there's a better way.

Magic auto-generation

In order to save us from the tiresome job of writing a very large number of methods which do essentially the same thing, the `overload` pragma can arrange to do much of the hard work for us. It does this through a process called *magic auto-generation* (yes, that's the technical term).

How it works is quite simple. If I overload a particular operator, the `overload` pragma will figure out whether it can derive any other operators from that, and do so if required. Since the <=> operator can be used to determine if two objects are greater than, less than, or equal to each other, it can be used to magically auto-generate all other numeric comparisons (>, >=, ==, etc). The same holds for `cmp` and string comparisons.

So, let's assume that we overloaded the <=> operator in the `PlayingCard` class above. We can now write code that looks like this:

```
#!/usr/bin/perl -w
use strict;
use PlayingCard;

# Assume we've implemented the deck class method, to return a full
# deck of cards.

my @deck = PlayingCard->deck();

# Shuffle...
```

```
@deck = sort { rand() <=> 0.5 } @deck;

# Deal one card each...

my $my_card   = pop(@deck);
my $your_card = pop(@deck);

# And compare...

if ($my_card > $your_card) {
        print "I win!\n";
} elsif ($my_card < $your_card) {
        print "I lose.\n";
} else {
        print "We draw.  Isn't that nice?\n";
}
```

Overloading using attributes

An alternate way of declaring which subroutines are responsible for overloaded operators is by using the `Attribute::Overload` module. This allows you to define *attributes* on subroutines to indicate they are to be used for overloaded operations.

```
use Attribute::Overload;

sub as_string : Overload("") {
        my ($this) = @_;

        return join(" ",$this->colour,$this->name,"at",$this->location);
}
```

When using `Attribute::Overload` there are a few things to remember:

- The operator name is not quoted or escaped in any way. You should write these as:

  ```
  sub add : Overload(+) { ... }
  sub string : Overload("") { ... }
  ```

- The `Attribute::Overload` module associates a specific subroutine (not a subroutine name) with an overloaded operator. Inherited classes need to explicitly declare which methods are responsible for overloaded operations, otherwise those in the parent class will be used. This behaviour is the same as using subroutine references with the `overload` pragma.

Exercises

1. Declare your `to_string` subroutine on your `Coin` class to have an Overload attribute for "".
2. Create and print a `Coin` object.
3. Create and print a `Coin::Weighted` object.
4. Provide a separate `to_string` subroutine overload for your `Coin::Weighted` class. Create a `Coin` coin and a `Coin::Weighted` coin and print them both.

Chapter summary

- Operators can be overloaded to increase (or decrease) the legibility and intuitiveness of our code.
- We can overload the stringification operator (`q{""}`) to change how our object behaves when it is printed or used as a string.
- We can overload comparison operators to change the way in which objects are compared. We can change other operators to change how our objects behave in other circumstances too.
- The `overload` pragma will *auto-magically generate* overload methods for us when possible. This saves us from having to tediously code them all ourselves.
- The `Attribute::Overload` module can be used to place overload declarations on the subroutines that handle the overloaded operations, rather than with your `use` declarations.

Chapter 16. Exceptions

In this chapter...

We all know that handling errors is important, and the most frequently seen way of handling errors in Perl is to deal with them in the code where they occur. Another approach adopted by many modern languages, including Perl, is to make use of *exceptions*, which allow for errors to be handled in a separate block of code. Proper use of exceptions can improve both readability and correctness of code. In this chapter, we examine exceptions in Perl.

What is an exception?

The Free Online Dictionary of Computing (http://wombat.doc.ic.ac.uk/foldoc/) defines an exception as *"an error condition that changes the normal flow of control in a program"*. Exceptions may be *thrown* by the underlying operating system or language (eg, when trying to write to a closed file, or dividing by zero), or they can be thrown by modules or code to indicate that something exceptional has happened.

An important aspect of an exception is that it can be *caught* and handled. This may involve rolling back a transaction, attempting to perform the operation a different way, ignoring the exception, or printing an error to the user. Uncaught exceptions may kill the program entirely.

Throwing exceptions in Perl

You may have been throwing exceptions in Perl for years, and been unaware that you have been doing so. The following familiar code throws an exception when the file cannot be opened:

```
open(FILE,"< $filename") or die "Cannot open $filename - $!\n";
```

The `die` throws an exception. Normally, these exceptions aren't caught, and so your program dies with an error.

Catching exceptions in Perl

Most people are surprised when they learn that *catch* in Perl is spelled `eval`. Any exception (using `die`) that's thrown inside an `eval` doesn't kill the program, instead it gets placed into the special variable `$@`.

Perl has two very different `eval` constructs, commonly referred to as *string eval* and *block eval*, depending upon the argument which they accept.

String eval takes a string, parses it (and re-parses it every time the `eval` is executed), and executes the resulting code. It's most commonly used for delaying parsing and execution of code until run-time. Because the string in a *string eval* gets re-parsed every time the statement is executed, there's a perception that all `eval` constructs are slow. However this is not the case with *block eval*.

The *block eval* construct takes a block, which is parsed at the same time as the code surrounding it, and executed within the same context as the surrounding code. It comes with no performance penalty, and is used almost exclusively for exception handling. Here's an example:

```
eval {
        my $result = $customer->credit_card->bill($amount);
        do_something_with($result);
}; # Don't forget that semi-colon!

if ($@) {
        # Oh dear, it didn't succeed.
}
```

In the case that something calls `die` or otherwise generates a fatal error, the execution of code will stop and `$@` will be set. In the example above, this would include the circumstance where `$customer`, or the result of any of the chained methods called on `$customer` were undefined, in addition to exceptions generated from those methods.

Inspection of `$@` can be done to determine exactly what sort of exception occurred. In the case of a regular `die` this will contain a string. However it is also possible to die with an *object*, which can make exception handling much cleaner. We'll be discussing this topic in greater detail later in this chapter.

Having Perl throw more exceptions

One of the reasons for using the exception-based paradigm is to free the programmer from having to do error checking at every stage of an operation. Being able to wrap an operation in an `eval` and then test to see if the operation as a whole has failed can result in much cleaner and maintainable code than testing each element individually.

By convention, most Perl functions and modules indicate errors by using return values, rather than throwing an exception. This means we still have to check all of our functions returns and throw the exceptions ourselves, however this checking of every step defeats many of the advantages of using exceptions to begin with. However, there is a way to change Perl's behaviour.

Perl's `use warnings` pragma allows us to escalate mere warnings into full-blown exceptions. Let's examine the following code:

```
eval {
        socket(SOCKET,PF_INET,SOCK_STREAM,$tcp)
                or die "Could not make socket - $!\n";

        setsockopt(SOCKET,SOL_SOCKET,$option,$value)
                or die "Can't setsockopt - $!\n";

        bind(SOCKET,$address) or die "Could not bind socket - $!\n";
        listen(SOCKET,1) or die "Listen failed - $!\n";
        accept(CLIENT,SOCKET) or die "Accept failed - $!\n";

        print CLIENT "Hello, the time is now".localtime()."\n"
                or die "Could not print to socket - $!\n";

        close(CLIENT) or die "Bizarre, could not close - $!\n";
};

# Trivial handling of exceptions.
warn "Connection handling failed - $@" if $@;
```

Sockets require many operations, and there are plenty of places where things could go wrong. As such, our code is littered with `or die "..."` statements. It's easy to forget that these are needed, and they definitely detract from the readability of the code.

```
eval {
        use warnings FATAL => qw(io);   # All I/O warnings are now fatal.

        socket(SOCKET,PF_INET,SOCK_STREAM,$tcp);
        setsockopt(SOCKET,SOL_SOCKET,$option,$value);
        bind(SOCKET,$address);
        listen(SOCKET,1);
        accept(CLIENT,SOCKET);
        print CLIENT "Hello, the time is now".localtime()."\n";
        close(CLIENT) or die "Bizarre, could not close - $!\n";
};

# Trivial handling of exceptions.
warn "Connection handling failed - $@" if $@;
```

By promoting all I/O warnings to errors, we've removed the need for us to check our return values, as trying to perform an erroneous operation, such as setting options on a closed socket, or printing to a closed filehandle, will now result in an exception being thrown.

It's worth noting that a failed `socket` call will not generate an exception, but the action of trying to set options on it will. We have also kept the `or die "...` after our `close`, since failing to close a filehandle does not generate a warning or exception.

Real-world examples of exceptions

Most people don't really begin to appreciate exceptions until they realise that there are *real* modules out there, which handle exceptions very well, and which they're using *every day*. The `DBI` module is just one of these.

The `DBI` module is used to access databases. Almost everyone who's needed to interface with a database in Perl has used `DBI`. If you haven't, then don't worry, the following still contains valuable lessons and examples, and there's a *very* good chance you'll end up using `DBI` sometime during your Perl programming career.

When using `DBI`, a lot of time is spent checking to ensure things are still okay. Did we connect to the database? Did we authenticate? Was that last SQL statement free of errors? Did we get back error-free results? Is the database still there? Large amounts of programming time and readability is spent checking for errors. Here's an example:

```
# Connect to the database, or die.
my $dbh = DBI->connect($dsn,$user,$pass,{AutoCommit => 0})
        or die $DBI::errstr;

# Start a transaction, or die.
$dbh->begin_work or die $dbh->errstr;

# Prepare some SQL, or die.
my $sth = $dbh->prepare($SQL) or die $dbh->errstr;

# Execute the SQL with some bind values, or die.
$sth->execute($customer,$purchase,$number) or die $dbh->errstr;

# Pull out some rows...
while (my $row = $sth->fetchrow_hashref) {
```

```
        # Process each row here.
}

# ... or die (if there was a problem in retrieving rows).
$DBI::err and die $dbh->errstr;

# Commit our transaction, or die.
$dbh->commit or die $dbh->errstr;
```

For every operation involving `DBI` we're manually checking for errors. Some of the more obscure checks (like checking the value of `$DBI::err` after a fetch loop have finished) are easy to forget.

However, `DBI` also has a mode whereby it throws exceptions upon errors, rather than meekly returning a false value. This not only improves readability, but also removes the problem of forgetful programmers not checking their return values.

```
# Connect to the database, using RaiseError to throw exceptions.
my $dbh = DBI->connect(
        $dsn,$user,$pass,
        {AutoCommit => 0, RaiseError => 1,
         PrintError => 0, ShowErrorStatement => 1}
);

$dbh->begin_work;
my $sth = $dbh->prepare($SQL);
$sth->execute($customer,$purchase,$number);

# Pull out some rows....
while (my $row = $sth->fetchrow_hashref) {
        # Process each row here.
}

$dbh->commit;
```

It's worth noting what some of the options we've passed through to `DBI->connect` are doing:

- `AutoCommit => 0` states that we should not automatically commit every statement. This only works on databases that allow transactions.
- `RaiseError => 1` states that any error from DBI should be turned into an exception and thrown. It's the reason why we don't have `or die "..."` scattered throughout our code.
- `PrintError => 0` prevents errors from being printed using `warn`. An error will result in an exception which will be displayed if not caught. If the exception is caught, we may wish to decide for ourselves if it should generate a warning.
- `ShowErrorStatement => 1` means that any exception (or warning) will also contain the SQL that generated the error. This wonderful option takes most of the detective work out of trying to debug *which* bit of SQL is being naughty, and is highly recommended.

As can be seen, having the `DBI` module throw exceptions when required simplifies our error-handling. In this example, we're simply dying with an error if anything goes wrong, with `DBI` automatically arranging for our transactions to be rolled-back in case of error. In many applications involving `DBI`, that's the correct thing to do.

However, we can also use the same code when we wish to handle errors. Let's take the example of a database import. We may have a number of records we wish to import into a database, and some of them may fail. Rather than aborting the entire process, we'd like to note which of these failed, and continue on.

```
my $dbh = DBI->connect(
        $dsn,$user,$pass,
        {AutoCommit => 0, RaiseError => 1,
         PrintError => 0, ShowErrorStatement => 1}
);

my $sth  = $dbh->prepare($SOME_SQL_INSERT_CODE);
my $sth2 = $dbh->prepare($SQL_FOR_SECOND_TABLE);

while (<RECORDS>) {
        my $record = $_;
        eval {
                my ($fields1, $fields2) = process_record($record);
                $dbh->begin_work;
                $sth->execute(@$fields1);
                $sth2->execute(@$fields2);
                $dbh->commit;
        };

        # Error-handling.
        if ($@) {
                eval { $dbh->rollback; };  # Rollback current transaction.

                if ($@ =~ /execute failed:/) {
                        # Hmm, looks like our record had bad data.
                        # We'll log that, and continue onwards.
                        log_record($record);
                } else {
                        # Some other kind of error?  We don't
                        # know how to deal with these, so we'll
                        # re-throw the exception.
                        die $@;
                }
        }
}
```

The above code allows us to process a large number of records, back-out and log the ones which fail, and can easily be expanded to include extra code that may also generate exceptions.

The Error module

CPAN has an `Error` module which provides both syntactic sugar as well as a basis for exception objects for use in Perl. In this section we'll cover the basics of using `Error` and some of the common pitfalls you may encounter.

The `Error` does not come standard with Perl. To use it, you must install it from CPAN first.

Loading the Error module

In order to use the extra syntax provided by `Error`, one needs to call it with the `:try` argument:

```
use Error ':try';
```

Without requesting `:try`, the `Error` class is loaded, but none of the extra syntax is provided for your program.

You can find further documentation on `Error` at CPAN (http://search.cpan.org/~uarun/Error-0.15/Error.pm) or by using **perldoc Error** if it is installed on your system.

Syntax provided by the Error module

The `Error` module provides extra syntax for dealing with exceptions. Here’s an example:

```
use Error ’:try’;

my $CONFIG = "Config.txt";

-e $CONFIG or throw Error::Simple("No config file");

try {
        open(FILE,"< $CONFIG")
                or die with Error::Permission(
                        -filename => $CONFIG,
                        -value => $!,
                        -text => "Cannot open $CONFIG - $!\n"
                );
        while (<FILE>) {
                do_some_stuff();
                die with Error::Simple("Oops!") if $some_condition;
        }
}
catch Error::Permission with {
        my $E = shift;
        print STDERR "Permission difficulties with $E->{’-filename’}: ".
                     $E->{’-text’};
}
except {
        my $E = shift;
        return {
                "Error::IO"   => \&handle_io_exception,
                "Error::CPU"  => \&handle_cpu_exception,
                "Error::Acme" => \&handle_acme_exception,
        };
}
otherwise {
        print STDERR "Caught an exception not handled anywhere else\n";
}
finally {
        tidy_up_program();      # Always gets called.
}; # Don’t forget we need a trailing semi-colon.
```

We will now examine each piece of extra syntax in turn.

try BLOCK CLAUSES

The `try` construct is used to enclose a block of code. If no exception is thrown, then try returns the result of the block. If an exception *is* thrown, then the clauses described below are examined and the appropriate action taken.

catch CLASS with BLOCK

This clause allows exceptions of a given CLASS (or its descendants) to be handled with the BLOCK provided. An `Error` object is passed to the BLOCK as the first argument (`$_[0]`). This error can be propagated by calling the `throw` method upon it.

If the `catch` block returns a value then this will be returned by `try`.

except BLOCK

Rather than writing a separate `catch` for every class of error, it's possible to provide a hash mapping classes to subroutines, and this is the purpose of an `except` block.

If an `except` block exists, then it will be passed the error as its first argument. This allows the `except` block to do any necessary preparation. The `except` block should then return a hashref mapping classes to subroutines.

otherwise BLOCK

The `otherwise` block will be called if no `catch` or `except` wishes to deal with the error. Only one `otherwise` can be specified per `try` block.

finally BLOCK

The `finally` block will be called regardless of whether or not the `try` block succeeded or resulted in an exception. It's a useful place to perform any necessary clean-up.

Error objects

The `Error` module also provides an abstract class (also called `Error`) which is a useful base for exception objects. The `Error` module provides a `Error::Simple` class that can be used directly. Any call to `die` with a literal string will be converted into an `Error::Simple` object.

Constructing an Error object

The `Error` object is implemented as a hash and can take the following arguments to its constructor (all optional). In some cases where these defaults are not specified defaults are given:

- `-file` (the name of the file that the error was thrown in)
- `-line` (the line number of the file that the error was thrown from)
- `-text` (the error message)
- `-value` (a numerical value associated with the error, defined when the exception is thrown)
- `-object` (an object which is associated with the exception, defined when it is thrown)

The `-file` and `-line` arguments are automatically filled in with the location where the error was thrown, or are automatically extracted from the `die` message in the case of `Error::Simple` objects.

The `-text`, `-value` and `-object` arguments allow for extra information to be provided about an error, such as an error message, a well-defined error-code, or an object which is associated with the error.

These not are defined when using a simple `die` and are often not set when using an `Error::Simple` object.

If an object is passed to the constructor, then the `Error` class will remember this as the last error associated with that object's class. It can be retrieved with `Error->prior($classname)`.

Error syntax

Any object that inherits from `Error` can be used with the following constructs:

- `throw Some::Error ( ARGS )` will throw an exception. Any arguments will be passed to the class' constructor.
- `with Some::Error ( ARGS )` is syntactic sugar to allow the programmer to write `die with Some::Error ( ... )`. It merely creates an `Error` object and returns it.
- `record Some::Error ( ARGS )` is also syntactic sugar that creates and returns a member of the `Error` class. It's most useful for `Error` classes which log information when they are constructed.

Chapter summary

- Exceptions allow us to handle errors in the code where they occur.
- An exception is an error condition that changes the normal flow of control in a program.
- Exceptions can be caught and handled, or ignored. Uncaught exceptions can kill the program entirely.
- The `die` function can be used to throw a simple exception.
- In Perl exceptions are caught by using the `eval` construct.
- Promoting Perl's warnings to fatal errors can allow us to generate exceptions in large operations which we can then handle correctly.
- Some modules, such as `DBI` allow the programmer to utilise exceptions very well to detect and handle errors.
- The `Error` module provides a more structured and syntactically pleasing way of dealing with exceptions in Perl.

Chapter 17. Conclusion

What you've learnt

Now you've completed Perl Training Australia's Object Oriented Perl module, you should be confident in your knowledge of the following fields:

- Object orientation (in Perl anyway).
- What packages and modules are.
- How to write packages and modules.
- How to write Perl objects.
- How to write constructors, init functions and destructors for your objects.
- How your class can inherit from other classes.
- How you can redispatch method calls that come to your class unintentionally.
- What polymorphism is, and how easy it is in Perl.
- How to overload operators.

Where to now?

To further extend your knowledge of Perl, you may like to:

- Work through any material not included during the course
- Visit the websites in our "Further Reading" section (below)
- Follow some of the URLs given throughout these course notes, especially the ones marked "Readme"
- Join a Perl user group such as Perl Mongers (http://www.pm.org/)
- Join an on-line Perl community such as PerlMonks (http://www.perlmonks.org/)
- Extend your knowledge with further Perl Training Australia courses such as:
    - CGI Programming with Perl
    - Perl Security
    - Database Programming with Perl

  Information about these courses can be found on Perl Training Australia's website (http://www.perltraining.com.au/).

Further reading

Books

- Damian Conway, *Object Oriented Perl*, Manning, 2000. ISBN 1-884777-79-1
- Tom Christiansen and Nathan Torkington, *The Perl Cookbook*, O'Reilly and Associates, 1998. ISBN 1-56592-243-3.
- Joseph N. Hall and Randal L. Schwartz *Effective Perl Programming*, Addison-Wesley, 1997. ISBN 0-20141-975-0.

Online

- The Perl homepage (http://www.perl.com/)
- The Perl Journal (http://www.tpj.com/)
- Perlmonth (http://www.perlmonth.com/) (online journal)
- Perl Mongers Perl user groups (http://www.pm.org/)
- PerlMonks online community (http://www.perlmonks.org/)
- comp.lang.perl.announce newsgroup
- comp.lang.perl.moderated newsgroup
- comp.lang.perl.misc newsgroup
- Comprehensive Perl Archive Network (http://www.cpan.org)

Colophon

```
#!/usr/bin/perl
# Copyright (c) Marcus Post, <marcus@marcuspost.com>
               #              #              #              #
$_=q,my(@f|@c|x$_=q.my(@f|@c|x$_=q.my(@f|@c|x$_=q.my(@f|@c|x$_=q.my(@f|@c|x$
@w);@a=@f=<DAT%@w);@a=@f=<DAT%@w);@a=@f=<DAT%@w);@a=@f=<DAT%@w);@a=@f=<DAT%@
A>;seek(DATA|0!A>;seek(DAT|00!A>;sek((DAT|00!A>;sek((DT|000!A>;sek((DT|000!A
|0);@c=<DATA>;Y|0);@c=<DAA>;;Y|0);c=<<DAA>;;Y|0);c=<<AA>;;;Y|0);c=<<AA>;;;Y|
until(($_=pop(zuntil(($_pop((zuntl(($$_pop((zuntl(($_pop((zzuntl(($_pop((zzu
@c))=~/^_/){};Q@c))=~/^_){};;Q@c)=~/^^_){};;Q@c)=~/^_){};;;Q@c)=~/^_){};;;Q@
unshift(@a|$_)xunshift(@|$_))xunsift((@|$_))xunsift(@|$_)))xunsift(@|$_)))xu
;for(1..3){pri%;for1..3){pri%;for1..3){pri%%;for1..3{pri%%%;for1..3{pri%%%;f
nt(shift(@c));!nt(hift(@c));!nt(hift(@c));;!nt(hift(@));;;!nt(hift(@));;;!nt
}for(@f){my($sY}fr(@f){my($sY}fr(@f){my($$sY}fr(@f){m($$ssY}fr(@f){m($$ssY}f
);split//;$_=sz);split//$_=ssz);slit///$_=ssz);slit//$_=sssz);slit//$_=sssz)
hift(@c);$_=~sQhift(@c);_=~ssQhif(@c));_=~ssQhif(@c))_=~sssQhif(@c))_=~sssQh
/(.{15}).*/\1/x/(.{15})./\1//x/(.15}))./\1//x/(.15}))/\1///x/(.15}))/\1///x/
;@w=split//;fo%;@w=split/;foo%;@wspliit/;foo%;@wspliit;fooo%;@wspliit;fooo%;
r(@_){$w[$s+15!r(@_{$w[$s+15!r(@_{$w[$s+15!!r(@_{$w[$s15!!!r(@_{$w[$s15!!!r(
−$_]=(($w[$s]eY−$_=(($w[$s]eY−$_=(($w[$s]eeY−$_=(($w[$]eeeY−$_=(($w[$]eeeY−$
q"|")?".":$w[$zq"")?".":$w[$zq"")?".":$w[[$zq"")?".":$ww$$zq"")?".":$ww$$zq"
s]);$s++;}for(Qs]);$s++;for((Qs])$s+++;for((Qs])$s+++;fr(((Qs])$s+++;fr(((Qs
1..75){unless(x1..75){uness((x1..5){uuness((x1..5){uunes(((x1..5){uunes(((x1
$w[$_]ne’’){$w%$w[$_]ne’){$ww%$w[_]nee’){$ww%$w[_]nee’){$ww%$w[_]nee’){$ww%$
[$_]=$w[($_−1)![$_]=$w[(_−11)![$_=$ww[(_−11)![$_=$ww[(_−11)![$_=$ww[(_−11)![
];}}print(joinY];}}printjooinY];}prrintjooinY];}prrintooinnY];}prrintooinnY]
""|@w);print"\z""|@w);print"\z""|@w);print"\z""|@w);pint"\zz""|@w);pint"\zz"
n";}print@a;,;#n";}print@a;.;#n";}print@a;.;#n";}prin@a;.;;#n";}prin@a;.;;#n
y!|zY\!%x!,Q!;#y!|zY\!%x!.Q!;#y!|zY\!%x!.Q!;#y!|zY\!%x.Q!;;#y!|zY\!%x.Q!;;#y
s{Q.*\n}[]g;#<>s{Q.*\n}[]g;#<>s{Q.*\n}[]g;#<>s{Q.*\n}[]g;#<>s{Q.*\n}[]g;#<>s
eval;#EndFini!$eval;#EndFini!$eval;#EndFini!$eval;#EndFini!$eval;#EndFini!$e
__DATA__
000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000
000000000000110000001100000000000000000011100000000000000000
000000000001110000011100000000000000000111100000000000000000
000000000011110000011110000000000000001111110000000000000000
000000000011110000011110000000000000001111100000000000000000
000000000011110000011110000000000000001111000000000000000000
000001111111111111111111111110000000001111000000000000000000
000011111111111111111111111100000000000111000000000000000000
000111111111111111111111111000000000000111000000000000000000
000000000011110000011110000000000000000111000000000000000000
000000000011110000011110000000000000000111000000000000000000
000000000011110000011110000000000000000111000000000000000000
000000000011110000011110000000000000000011100000000000000000
000001111111111111111111111110000000000011100000000000000000
000011111111111111111111111100000000000011100000000000000000
000111111111111111111111111000000000000001100000000000000000
000000000011110000011110000000000000000001100000000000000000
000000000011110000011110000000000000000001100000000000000000
000000000011110000011110000000000000000000000000000000000000
000000000011100000011100000000000000000000000000000000000000
000000000011000000011000000000000000000001111000000000000000
000000000000000000000000000000000000000111111000000000000000
000000000000000000000000000000000000000111110000000000000000
000000000000000000000000000000000000000011100000000000000000
000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000
```

The Perl code on the cover was written by Marcus Post. It generates stereograms based upon the information provided in its DATA segment (not shown on the front cover due to space). The output of the script is not only a stereogram, but is also a valid Perl program that is capable of creating new stereograms.

A discussion of the code where it was originally posted can be found on PerlMonks (http://perlmonks.org/index.pl?node_id=118799). More information about Marcus Post and his work can be found on his website (http://www.marcuspost.com/).