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.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);$_=~sQhift(c);$_=~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=splIit;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 1.13 of Perl Training Australia's "Object Oriented Perl" training manual.

Table of Contents

List of Tables

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

Perl is a cross-platform computer language which runs successfully on approximately 30 different operating systems. However, as each operating system is different this does occasionally impact on the code you write. Most of what you will learn will work equally well on all 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 5.8.8, 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 appear *like this*


Notes and tips appear offset from the text like this.


Advanced sections 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.

 Readme sections suggest pointers to more information which can be found in the *Programming Perl* book, system documentation such as manual pages and websites.

 Caution sections 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* (1999) 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 basic 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.

Best practice regarding object oriented programming with Perl has evolved in the years since Conway's "Object Oriented Perl" book was released. Many of these changes are reflected in *Perl Best Practices* (2005) also by Damian Conway.

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 vending 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* headlight (or two) but the car is not a headlight. 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.

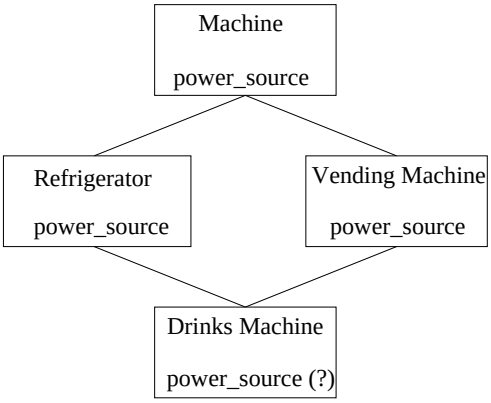
Multiple inheritance

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.

Ten rules for when to use OO

This section is based on Dr Damian Conway's rules with commentary by Perl Training Australia. To read Dr Damian Conway's own commentary, read Chapter 11 of his book: *Perl Best Practices*.

You should use OO:

1. *When your design is large, or is likely to become large.*
2. *When data is aggregated into obvious structures, especially if there's a lot of data in each aggregate.*

   A person's name is often not a good candidate for an object. A string of any length is still only a single piece of information. If however, you wish to associate other information with the name, such as age, address, phone number (enough data for example that you'd think to use a hash), then you have a good case for an object.

   ```
   my %person1 = (
           name => "Julien Gileson",
           age => 42,
           address => "42 Halloway St"
   );
   ```

3. *When types of data form a natural hierarchy that lets us use inheritance.*

   Inheritance allows us to have a common set of operations shared by all similar objects, while still allowing specific operations for each specialised type. For example, one may have a class that tests the functionality on a website, and another that spiders a company's local intra-net. Both classes share a common base concerning access and parsing of web-pages.

4. *When operations on data varies on data type.*

   Many similar types of data have similar operations. For example music stored on CD is a different kind of data than music stored on disk in mp3 or ogg vorbis format. However, although the details are different for all, all these kinds of music can be played. An OO model allows code similar to the following, without needing to be concerned about the underlying format:

   ```
   foreach my $song ($cd_track, $mp3, $ogg) {
           $song->play();
   }
   ```

5. *When it's likely you'll have to add data types later.*

   Consider the above example. If our specification says that we want something which will handle mp3 files and music from CDs, we could code a solution to handle these two exact cases. However, as requirements tend to evolve over time it may become likely that we'll need to handle ogg vorbis, midi, iTunes, and other formats too.

   A carefully designed OO model can allow us to accommodate these future requirements without significant changes to our application.

6. *When interactions between data is best shown by operators.*

   Perl's object model allows operators to be overloaded so that things work the way you'd like them to. For example:

   ```
   $volume += 5;
   ```

   is traditionally only meaningful if $volume is a number. However, with overloading, `$volume` can represent the actual state of our speakers, and numerical adjustments can directly increase or decrease the energy output of the speakers.

7. *When implementation of components is likely to change, especially in the same program.*

   Object oriented programming assists in data encapsulation. A class should be treated as a black box by the code which uses it. Defining clear interfaces allows the internals of the class to be changed as many times as necessary without affecting the rest of the project.

   Thus if your class implements access to some data in a file, then the data's format could change from being in plain-text, to a flat-file database, to a full relational database without the rest of the application needing to change.

8. *When the system design is already object-oriented.*

9. *When huge numbers of clients use your code.*

   In addition to encapsulation, object oriented code encourages modularisation. By breaking your project up into separate modules it becomes easier to maintain and re-use. A change to a class should not change any application code, making bug-fixes and improvements more straightforward.

10. *When you have a piece of data on which many different operations are applied.*

    Consider a piece of data representing sound. This sound can be mixed, adjusted, run backwards, have echo effects added and many other kinds of operations. By wrapping it in a Perl object we can associate these operations with this kind of data. The resulting code is often easier to read, understand, and maintain.

11. *When the kinds of operations have standard names (check, process, etc).*

    The traditional way of handling identical names for operations on unrelated data is to use the data type in the subroutine name. For example.

    ```
    sub process_person { ... }

    sub process_payment { ... }

    sub process_music { ... }
    ```

    Objects side-step this issue by associating the operation with the data allowing all of the subroutines to have the same name, but be in different name spaces. So:

    ```
    Person::process(); Payment::process(); Music::process();
    ```

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. External Files and Packages

In this chapter...

In this chapter we'll discuss how we can split our code into separate files. We'll discover Perl's concept of packages, and how we can use them to make our code more robust and flexible.

Splitting code between files

When writing small, independent programs, the code can usually be contained within a single file. However there are two common occurrences where we would like to have our programs span multiple files. When working on a large project, often with many developers, it can be very convenient to split a program into smaller files, each with a more specialised purpose. Alternatively, we may find ourselves working on many programs that share some common code base. This code can be placed into a separate file which can be shared across programs. This saves us time and effort, and means that bug-fixes and improvements need to be made only in a single location.

Require

Perl implements a number of mechanisms for loading code from external files. The most simplest of these is by using the `require` function:

```
require 'file.pl';
```

Perl is smart enough to make sure that the same file will not be included twice if it's required through the same specified name.

```
# The file is only included once in the following case:
require 'file.pl';
require 'file.pl';
```

Required files *must* end with a true value. This is usually achieved by having the final statement of the file being:

```
1;
```

Conflicts can occur if our included file declares subroutines with the same name as those that appear in our main program. In most circumstances the subroutine from the included file takes precedence, and a warning is given.

We will learn how to avoid these conflicts later in this chapter when we discuss the concept of *packages*.

The use of `require` has been largely deprecated by the introduction of modules and the `use` keyword. If you're writing a code library from scratch we recommend that you create it as a module. However, `require` is often found in legacy code and is a useful thing to understand.

Any code in the file (except for subroutines) will be executed immediately when the file is required. The `require` occurs at run-time, this means that Perl will not throw an error due to a missing file until that statement is reached, and any subroutines inside the file will not be accessible until after the `require`.

Variables declared with `my` are not shared between files, they are only visible inside the block or file where the declaration occurs. To share packages between files we use *package variables* which are covered later in this chapter.

The use of modules (which we will learn about later) allows for external files to be loaded at compile-time, rather than run-time.

Use strict and warnings

Perl pragmas, such as `strict` and `warnings` are lexically scoped. Just like variables declared with `my`, they last until the end of the enclosing block, file or eval.

This means that you can turn strict and warnings on in one file without it influencing other parts of your program. Thus, if you're dealing with legacy code, then your new libraries, modules and classes can be strict and warnings compliant even though the older code is not.

Example

The use of `require` is best shown by example. In the following we specify two files, `Greetings.pl` and `program.pl`. Both are valid Perl programs on their own, although in this case, `Greetings.pl` would just declare a variable and a subroutine, and then exit. As we do not intend to execute `Greetings.pl` on its own, it does not need to be made executable, or include a shebang line.

Our library code, to be included.

```
# Greetings.pl
# Provides the hello() subroutine, allowing for greetings
# in a variety of languages.  English is used as a default
# if no language is provided.

use strict;
use warnings;
use Carp;

my %greeting_in = (
        en      => "Hello",
        'en-au' => "G'day",
        fr      => "Bonjour",
        jp      => "Konnichiwa",
        zh      => "Nihao",
);
```

```
sub hello {
        my $language = shift || "en";

        my $greeting = $greeting_in{$language}
                or croak "Don't know how to greet in $language";

        return $greeting;
}

1;
```

Our program code.

```
#!/usr/bin/perl -w
# program.pl
# Uses the Greetings.pl file to provide another hello() subroutine
use strict;

# Get the contents from Greetings.pl
require "Greetings.pl";

print "English: ",    hello("en"),   "\n";       # Prints "Hello"
print "Australian: ", hello("en-au"),"\n";       # Prints "G'day"
```

FindBin

We can use `FindBin` to locate the directory of our Perl script. This can then be used, if required, to specify locations of required files stored relative to that script.

```
use FindBin qw($Bin);

# require the example.pl in the same directory as our script
require "$Bin/example.pl";

# require the other.pl file in the files/ directory
require "$Bin/files/other.pl";
```

`FindBin` can also be used to get the basename of the Perl script (`$Script`), a canonical path (with all links resolved) (`$RealBin`) and the script name plus canonical path (`$RealScript`).

Exercises

1. Create a file called `MyTest.pl` Define at least two subroutines; `pass` and `fail` which print some amusing output. Make sure that it uses `strict`.
2. Test that your code compiles by running **perl -c MyTest.pl**. (The **-c** tells Perl to check your code).
3. Create a simple Perl script which requires `MyTest.pl` and calls the functions defined within.

Introduction to packages

The primary reason for breaking code into separate files is to improve maintainability. Smaller files are easier to work with, can be shared between multiple programs, and are suitable for dividing between members of large teams. However they also have their problems.

When working with a large project, the chances of naming conflicts increases. Two entirely different files may have two different subroutines with the same name; however it is only the last one loaded that will be used by Perl. Files from different projects may be re-used in new developments, and these may have considerable name clashes. Multiple files can also make it difficult to determine where subroutines are originally declared, which can make debugging difficult.

Perl's *packages* are designed to overcome these problems. Rather than just putting code into separate files, code can be placed into independent packages, each with its own namespace. By ensuring that package names remain unique, we also ensure that all subroutines and variables can remain unique and easily identifiable.

A single file can contain multiple packages, but convention dictates that each file contains a package of the same name. This makes it easy to quickly locate the code in any given package.

Writing a package in Perl is easy. We simply use the `package` keyword to change our current package. Any code executed from that point until the end of the current file or block is done so in the context of the new package.

```
# By declaring that all our code is in the "Greetings" package,
# we can be certain not to step on anyone else's toes, even if
# they have written a hello() subroutine.

package Greetings;

use strict;
use warnings;

my %greeting_in = (
        en      => "Hello",
        'en-au' => "G'day",
        fr      => "Bonjour",
        jp      => "Konnichiwa",
        zh      => "Nihao",
);

sub hello {
        my $language = shift || "en";

        my $greeting = $greeting_in{$language}
                or die "Don't know how to greet in $language";

        return $greeting;
}

1;
```

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.

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`).

The scoping operator

Being able to use packages to improve the maintainability of our code is important, but there's one important thing we have not yet covered. How do we use subroutines, variables, or filehandles from other packages?

Perl provides a *scoping operator* in the form of a pair of adjacent colons. The scoping operator allows us to refer to information inside other packages, and is usually pronounced "double-colon".

```
require "Greetings.pl";

# Greetings in English.
print Greetings::hello("en"),"\n";

# Greetings in Japanese.
print Greetings::hello("jp"),"\n";

# This calls the hello() subroutine in our main package
# (below), printing "Greetings Earthling".
print hello(),"\n";

sub hello {
        return "Greetings Earthling";
}
```

Calling subroutines like this is a perfectly acceptable alternative to exporting them into your own namespace (which we'll cover later). This makes it very clear where the called subroutine is located, and avoids any possibility of an existing subroutine clashing with that from another package.

Occasionally we may wish to change the value of a variable in another package. It should be very rare that we should need to do this, and it's not recommended you do so unless this is a documented feature of your package. However, in the case where we do need to do this, we use the scoping operator again.

```
use Carp;

# Turning on $Carp::Verbose makes carp() and croak() provide
# stack traces, making them identical to cluck() and confess().
# This is documented in 'perldoc Carp'.

$Carp::Verbose = 1;
```

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`.

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.

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 globally accessible. These global variables are called *package variables*, and as their name suggests they live inside their current package. The preferred way to create package variables, under Perl 5.6.0 and above, is to declare them with the `our` statement. 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. Change your `MyTest.pl` file to include a package name `MyTest`.
2. Update your program to call the MyTest functions using the scoping operator.
3. Create a package variable `$PASS_MARK` using `our` inside `MyTest.pl` which defines an appropriate pass mark.
4. In your Perl script, create a loop which tests 10 random numbers for pass or fail with reference to the `$PASS_MARK` package variable. Print the appropriate `pass` or `fail` message.
5. Print out the version of the `Cwd` module installed on your training server. The version number is in `$Cwd::VERSION`. (You will need to `use Cwd;` first.)
6. Look at the documentation for the `Carp` module using the **perldoc Carp** command. This is one of Perl's most frequently used modules.

Answers for the above exercises can be found in `exercises/answers/MyTest.pl` and `exercises/answers/packages.pl`.

You may receive the following warning during this exercise:

```
Name "MyTest::PASS_MARK" used only once: possible typo at ....
```

This occurs because perl does not load your external file until the `require` statement executes at run-time. However perl generates warnings about possible typos at compile time, before it loads your required file. We can avoid this by insisting perl load the external file at compile-time.

```
BEGIN { require 'MyTest.pl'; };
```

Modules (which we'll discuss next chapter) provide a cleaner and more extensible syntax for loading code at compile time.

Chapter summary

- A package is a separate namespace within Perl code.
- A file can have more than one package defined within it.
- The default package is `main`.
- 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`
- 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).
- Package variables can be declared with the `our` keyword. This allows them to be accessed from inside other packages.
- The `require` keyword can be used to import the contents of other files for use in a program.
- Files which are included using `require` must end with a true value.

Chapter 4. Modules

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. We will also investigate how to write our own modules.

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 Data::Dumper`).
- 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 looking 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);
use File::Copy qw(copy);
```

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, and the `File::Copy` module for just the `copy` subroutine.

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***.

When compiling your script, Perl first does a full pass over your code looking for `use` statements. This means that Perl will load and compile all of your modules - no matter where they appear in your code - before it starts to compile your code.

If you want to load a module depending on some condition, you can use the `if` pragma. See **perldoc if** for more information.

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.

Exercise

1. 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.

Where does Perl look for modules?

Perl searches through a list of directories that were determined when Perl was 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. This can be done either in the shebang line, or on the command-line. For example:

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

- 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.

You can use the `lib` pragma with directory paths that don't exist. This allows you to have different include paths for each of your development, testing and deployment environments all included in your code.

```
use lib "c:/myperl/projectA/lib";
use lib "/usr/local/bin/projectA/lib";
```

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 other modules that were manually installed on your system by using **perldoc perllocal**. Generally this file only lists other modules that were installed by hand, or using one of the CPAN installers (more on this later). Modules installed through your operating system's packaging system may not appear in **perldoc perllocal**.

To find a complete list of modules available on your system, regardless of how they were installed, read the documentation provided by **perldoc -q installed**.

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. You can also use **perldoc -l** ***module_name*** to locate a particular module, and **perldoc -m** ***module_name*** to view the source of a 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

CPAN provides more than 15,000 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. This may vary between operating systems. On Unix and Unix-like operating systems the process is usually:

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

For ActiveState Perl installations (which includes many Microsoft Windows machines) the PPM (Programmer's Package Manager) can be used. PPM provides a graphical interface for downloading and installing modules, but not all CPAN modules are available via PPM, and not all PPM distributions are as up-to-date as their CPAN counterparts.

On newer versions of ActiveState Perl for Windows, and on all versions of Strawberry Perl, the **cpan** shell works identically as it does under Unix, and will locate and install an appropriate version of **make** if required.

Some times you may not be able to, or may not wish to, install CPAN modules in their default path. In this case you can provide a flag to the `Makefile.PL` program instructing it on your preferred top level directory. For example:

```
perl Makefile.PL PREFIX=/home/sue/perl/ LIB=/home/sue/perl/lib
```

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 `PREFIX` directive specifies the general location for files to be installed, and the `LIB` directive indicates that all modules should be kept together, instead of using separate directories for architecture dependent, core, and other modules.

Provided you're using a recent release of the CPAN shell, you can configure it to always install modules in a local directory using the following commands:

```
o conf makepl_arg "PREFIX=/home/sue/perl LIB=/home/sue/perl/lib"
o conf commit
```

 For coverage on installing modules on various operating systems read **perldoc perlmodlib**. If you want to distribute your own modules read **perldoc perlnewmod**.

 If you need to distribute your code to systems which aren't guaranteed to have all your required modules, and which may not even have Perl installed, PAR is the solution. PAR is the Perl Archiver and allows you to package up your code, all its data files, its modules and Perl itself if necessary and can be used to create a stand-alone executable. For more information see the PAR documentation (http://search.cpan.org/perldoc?PAR).

Finding quality CPAN modules

There are a lot of modules on the CPAN, and sometimes deciding which module is the best task for a job can be a little challenging. Luckily, there is a wealth of information to help you.

Firstly, many CPAN modules have ratings. After doing a search, any module with a star rating next to it has reviews about that module that can be viewed. A module with a large number of high-rating reviews is an indication of a quality module.

The CPAN Testing Service (http://cpants.perl.org/) maintains some automatically generated metrics on common quality indicators such as packaging structure, test suites, and dependencies.

There are lists of community-recommended modules on both perl.net.au (http://perl.net.au/wiki/Recommended_Perl_Modules) and the Perl 5 Wiki (http://www.perlfoundation.org/perl5/index.cgi?recommended_cpan_modules). These include links to further resources.

Finally, online communities such as PerlMonks (http://perlmonks.org/), or local Perl Mongers (http://pm.org/) chapters can often provide personal module recommendations. PerlMonks in particular has an extensive archive of module discussions and tutorials.

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 `used` 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.

The best way to get started with writing your own module is to use `Module::Starter`. Read its documentation on CPAN (http://search.cpan.org/perldoc?Module::Starter) and our mini-tutorial in our Perl Tips (http://perltraining.com.au/tips/2008-10-15.html).

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

To document your modules so that `perldoc` can provide information about them, read **perldoc perlpod** and **perldoc perlpodspec**.

Use versus require

Perl offers several different ways to include code from one file into another. `use` is built on top of `require` and has the following differences:

- Files which are `used` are loaded and executed at compile-time, not run-time. This means that all subroutines, variables, and other structures will exist before your main code executes. It also means that you will immediately know about any files that Perl could not load.
- `use` allows for the import of variables and subroutines from the used package into the current one. This can make programming easier and more concise.
- Files called with `use` can take arguments. These arguments can be used to request special features that may be provided by some modules.

Both methods:

- Check for redundant loading, and will skip already loaded files.
- Raise an exception on failure to find, compile or execute the file.

- Translate `::` into your systems directory separator.

Where possible `use` and Perl *modules* are preferred over `require`.

Warnings and strict

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.

Your modules should always use strict.

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

Exercise

This exercise will have you adapt your `MyTest.pl` code to become a module. There's a list at the end of this exercise of things to watch out for.

1. Create a directory named `p5lib`.
2. Move your `MyTest.pl` file into your `p5lib` directory and rename it to `MyTest.pm`.
3. Make sure `MyTest.pm` uses `strict` and `warnings`.
4. Test that your module has no syntax errors by running **perl -c MyTest.pm**.
5. Change your Perl script from before to use the `lib` pragma in order to find your module. (`use lib 'p5lib';`)
6. Change your Perl script to `use` your module. Check that everything still works as you expect.
7. Add a print statement to your module (outside any subroutines). This should be printed when the module is loaded. Check that this is so.

Answers can be found in `exercises/answers/p5lib/MyTest.pm` and `exercises/answers/modules.pl`

Things to remember...

The above exercises can be completed without reference to the following list. However, if you're having problems, you may find your answer herein.

- A module is a file that contains a package of the same name.
- Perl modules must return a true value to indicate successful loading. (Put `1;` at the end of your module).
- To use a module stored in a different directory, add this directory to the `@INC` array. (Put `use lib 'path/to/modules/'` immediately before your `use MyTest;` line.
- To call a subroutine which is inside a module, you can access it via the double-colon. Eg: `MyModule::test();`

Exporter

Perl modules often make subroutines and variables available to the calling code. Some do this by default such as `File::Copy` (which places the subroutines `copy` and `move` directly into the main namespace) and others do this on request such as `CGI`.

It is generally considered bad form to export things by default as they can easily interfere with symbols already in the main package. In object oriented code it is very rare to need to export anything, as your object already has access to all of its methods.

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.

Writing your own documentation

A module is not complete without the documentation. If you used `Module::Starter` to generate your starting files, much of the scaffolding will have been written for you. In either case, it's good to have an understanding of how it all works.

All of Perl's documentation is written in a format called Plain Old Documentation (POD). POD is deisnged to be a simple, clean documentation language. It uses three kinds of instructions.

- Text paragraphs which the formatter can reformat if required. This is the default form of documentation.
- Code paragraphs which the formatter must not reformat. These are signalled by starting each line in the paragraph with at least one white space character.
- Command paragraphs. These must start in the first column, be preceeded and followed by a blank line and start with an equals character (`=`).

```
package My::Module;
use strict;

# Generally your code goes up here. Your POD may be at the bottom
# (as in this example), but you can interleave your POD and your code
# if you so wish.  Interleaving POD and code can make it easier to keep
# your subroutine documentation up-to-date.

# Any command paragraph (which starts with an equals) causes Perl to
# consider all lines to be POD until it sees the =cut directive.

=head1 NAME

My::Module - Show an example of Perl documentation.

=head1 SYNOPSIS

    use My::Module qw(example pirate);

    example($foo, $bar);
    pirate($yarr, $avast);

=head1 DESCRIPTION

This documentation is an example.  It shows different headings,
code paragraphs, and even lists!  Here's a list now:

=over 4

=item *

Bullet lists are easy.

=item *

See L<perlpod> to see how to do numbered lists and definition lists.

=back

=head1 SUBROUTINES

=head2 example

    my $baz = example($foo, $bar);

Each function should have its own heading and a brief description of what
it does, its arguments and what it returns.

=head2 pirate

    my $booty = pirate($yarr, $avast);

It be a fine day for writing subroutine documentation 'pon the high seas.

=head1 AUTHOR

Perl Training Australia

=cut

1;
```

We can then render the POD part of this file, using **perldoc** just as we did to read Perl's normal documentation. **perldoc My/Module.pm**:

```
My::Module(1)        User Contributed Perl Documentation       My::Module(1)


NAME
       My::Module - Show an example of Perl documentation.

SYNOPSIS
           use My::Module qw(example pirate);

           example($foo, $bar);
           pirate($yarr, $avast);

DESCRIPTION
       This documentation is an example.  It shows different headings, code
       paragraphs, and even lists!  Here's a list now:

       o   Bullet lists are easy.

       o   See perlpod to see how to do numbered lists and definition lists.

SUBROUTINES
       example

           my $baz = example($foo, $bar);

       Each function should have its own heading and a brief description of
       what it does, its arguments and what it returns.

       pirate

           my $booty = pirate($yarr, $avast);

       It be a fine day for writing subroutine documentation 'pon the high
       seas.

AUTHOR
       Perl Training Australia


perl v5.8.8                       2009-03-19                     My::Module(1)
```


To make sure your POD is valid, you can use the **podchecker** which comes standard with perl.


To learn more about POD command paragraphs read **perldoc perlpod** and our Perl Tip (http://perltraining.com.au/tips/2005-04-06.html).

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 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.

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*. 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 name into which the object will be blessed. 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.

Starting our hash keys with underscores doesn't guarantee that programs which use this class won't access our hash directly and forgo any integrity checks our accessor functions may provide. Even worse, there is nothing in place to prevent the occasion where one part of our code uses the hash key value with an underscore and another part uses the hash key value without! We can all too easily end up with two hash keys where we meant only to have one!

If you wish to make it much harder to accidently create extra hash keys, you might be interested in lockable hashes. These allow you to lock the hash keys so that creation of new keys is not possible. For further information about these read **perldoc Hash::Util**.

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"); # Don't do this
```

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).

PlayingCard in full

Here's the `PlayingCard` class in full.

```
package PlayingCard;            # Our class name
use strict;
use warnings;

# 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()`.

Chapter 6. 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:

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

the `interests()` 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 `interests()` function above would be called thus:

```
interests(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.

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

sub interests {
        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";
}
```

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`. This also demonstrates the use of `Params::Validate` to check arguments and return a hash with defaults applied.

```
package DrinksMachine;
use strict;
use warnings;
use Carp;
use Params::Validate qw(validate :types);

# 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 $1,
#                        50 x 50c, 50 x 20c, 30 x 10c, 10 x 5c.
#
#  'location' is a required field, and has no default.
sub new {
        my ($class, %args) = @_;

        # The line above is enough to get our named parameters,
        # but we're going to use Params::Validate to perform some
        # basic input checking and set defaults.
        %args = validate(
                %args,
                {
                        desired_temperature => { type => SCALAR, default => 4 },
                        price           =>  { type => SCALAR, default => 1.20 },
                        starting_change => {
                                type    => ARRAYREF,
                                default => [ 0, 0, 50, 30, 10 ]
                        },
                        drinks => {
                                type    => ARRAYREF,
                                default =>
                                  [qw(cola orange lemonade squash water)],
                        },
                        location => { type => SCALAR },
                }
        );
        return bless(\%args,$class);
}
```

Exercises

1. Modify your `Coin` class so that its constructor uses named parameters rather than positional parameters.
2. Modify one of your earlier programs to use your changed Coin class.

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 7. 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. Games involving cards are extremely common, and while the rules for each game change, the cards themselves usually share a common set of attributes.

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. Improve the PlayingCard class such that its constructor and existing methods work as decided in the exercises above.
2. Add any other methods that you and your group decided upon.
3. Create a program that `use`s your module. Have it generate a deck of cards (without jokers), shuffle the deck, and print the first five cards. Use the `List::Util shuffle` function for this.
4. 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 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 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. However one should pause and give thought whether or not this is meaningful.

It's easy to catch an incorrect call to a class method by using the `ref` function:

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

        ref($class) and croak "Class method 'new_deck' called on object";

        # ...
}
```

The `ref` function returns the class of the object passed to it, or false if given something other than a reference. This can also be useful when writing `clone` constructors, which are intended to make a copy of the object upon which they are called.

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 been aware, there's no need to explicitly allocate or free memory in Perl. When we create a variable, the memory is allocated automatically. When we assign elements to an array or hash, those structures are extended as needed. When we put data into a scalar, the capacity of that scalar grows as required. All of this saves on both 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 immediately 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 trick Perl's garbage collection into never releasing memory if you're using circular references. For example, the following situation will leak memory:

```
{
        my ($x, $y);
        $x = \$y;        # $x refers to $y.
        $y = \$x;        # $y refers back to $x.
}
```

even after `$x` and `$y` 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 data structures will never get collected (although they're cleaned up when 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.

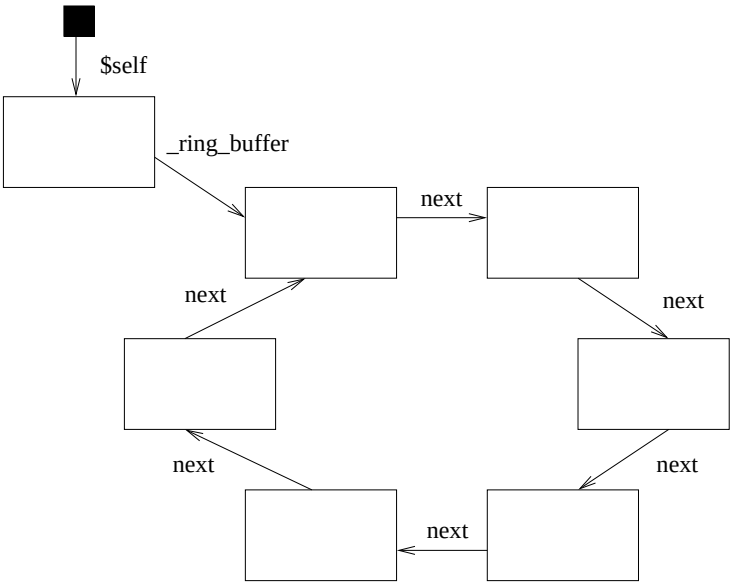
Most objects don't require destructor functions, nothing special needs to be done when a program doesn't need a particular `PlayingCard` anymore. However, some objects *do* require special treatment. For example, if I have an object which is controlling a modem or serial terminal, I might want to make 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

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

# If the argument "name" is passed, and the object already exists
# in our cache, we skip all initialisation and retrieve 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;
}

# Insert 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?

Weak references

An alternative to using destructors for memory management is to instead make use of Perl's *weak references* system. A weak reference is not considered when examining the reference count for an object, and does not prevent that object from being released by Perl's garbage collection system.

Weakening a reference is achieved by using `Scalar::Util`'s `weaken()` function, and can be tested for by using the `isweak()` function from the same module.

```
use Scalar::Util qw(weaken);

my $greet_ref;

{
        my $greeting = "Hello World\n";
        $greet_ref = \$greetings;       # $greet_ref starts a strong reference.

        weaken($greet_ref);             # $greet_ref is now a weak reference.
}

# $greeting is cleaned-up, and $greet_ref set to undef,
# as no strong reference exist to it.
```

Weak references are ideal when two objects have a relationship with each other, but only one is likely to be referenced from the main program. As an analogy, a car has an engine, and that engine needs to be connected to numerous parts of the car. This is a perfect example of a circular reference, which each object (car and engine) needing to be able to access the other. By making sure that the engine has a weak reference to the car, we can ensure that the engine is destroyed when the car is destroyed, without any extra programming required.

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 is discussed in this chapter.

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, you've probably 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. That's worth repeating, because those with object oriented experience from other languages may be surprised. *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


More information about the message dispatch mechanism is explained in Conway's book, pages 169-171.

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).

Alternately we can avoid manually altering the `@ISA` array by using the `base` pragma. This pragma also ensures that the appropriate module is loaded; which is something that editing `@ISA` does not.

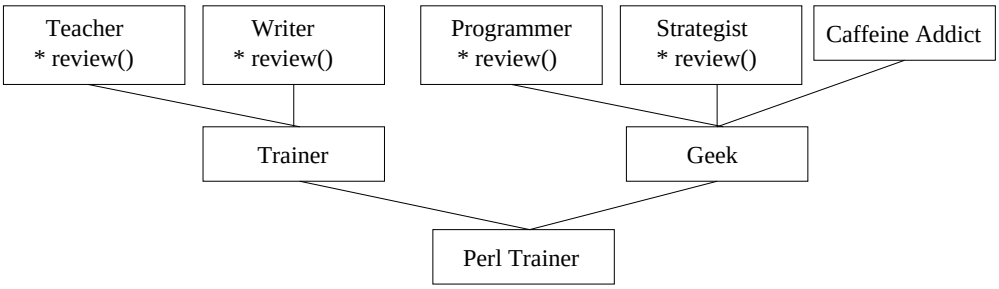
These options are best demonstrated with an example. It's all really quite simple.

```
package PerlTrainer;
our @ISA = (Trainer Geek);    # 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;
use base qw(Programmer Strategist CaffeineAddict);  # >= 5.6.0 again
```

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 some 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 its 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 `exercises/lib/PerlTrainer.pm`. 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;
our @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 $self = {};
        bless($self,$class);

        $self->_init(@args);

        return $self;
}
```

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

        # Initialise the object for all of our base classes.
        $self->Trainer::_init(%args);
        $self->Geek::_init(%args);

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

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

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 at an object's `@ISA` array is a naive approach, and 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.

`isa()` can also be called as a regular function. The main application of this is when we have a scalar that may not even be an object. Calling a method on a regular scalar results in an exception from Perl, whereas calling `UNIVERSAL::isa()` with an unblessed scalar does not. Two arguments (the scalar and the class) must be passed to `isa()` when used in this way.

```
print "This thing is a $class\n" if UNIVERSAL::isa($thing,$class);
```

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**.

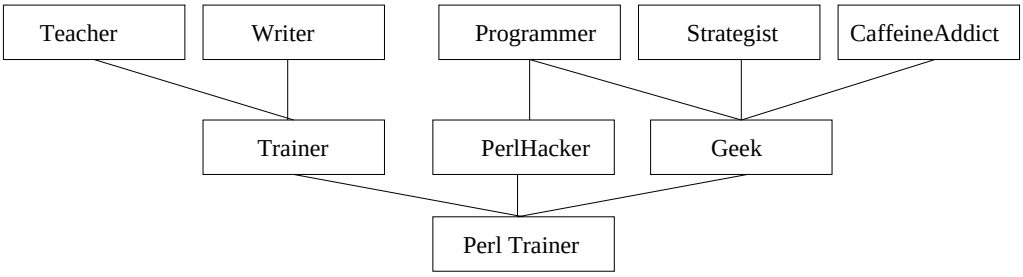
Problems with initialisers

Separating constructors from initialisers to allow all the classes in an inheritance hierarchy to initialise 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`). This can cause some interesting problems, particularly at object creation time. Using the techniques that we've just discussed, the `Programmer::_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;
our @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 $self;
}
```

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...

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

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

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

        return $self;
}
```

That code needs a bit of explaining. You'll recall that `$parent->can("_init")` checks to see if the given parent (or one of *its* parents) can handle a call to `_init`, and if so, returns a reference to that method. We then call that method directly (using `$self->$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.

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 and EVERY 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 base 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 $self = {};
        bless($self,$class);

        $self->_init(@args);

        return $self;
}

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

        # Protect against multiple inheritance
        my $PACKAGE = __PACKAGE__;
        return if $self->{_init}{$PACKAGE}++;

        # Initialise the object for all of our base classes.
        foreach my $parent (@ISA) {
                my $parent_init = $parent->can("_init");
                $self->$parent_init(%args) if $parent_init;
        }

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

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

Exercises

1. Derive a `Coin::Weighted` class from the `Coin` class. These coins behave as regular `Coin`s, however heads comes up 80% of the time instead of 50%. For now, don't worry if your statistics don't work with the new class.
2. Create a second coin program which creates a `Coin` object and a `Coin::Weighted` object.
   a. Use `isa` to check whether they are both `Coin`s.
   b. Use `isa` to check whether they are both `Coin::Weighted`s.

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

   c. Are the results as you expect?
3. Add the following functions to your `Coin::Weighted` class:

a. `set_weight` which sets the amount of favour the coin shows to heads.

b. `get_weight` which returns the current value of favour the coin shows to heads.

4. Create 10 each of `Coin` coins and `Coin::Weighted` coin and put them into an array. Randomise the array using the following code:

   ```
   use List::Util qw(shuffle);

   @array = shuffle @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`.

5. 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.

6. Your `Coin::Weighted` class should no longer need to have a separate constructor function. Remove this if you've created one.

7. 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.

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.

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

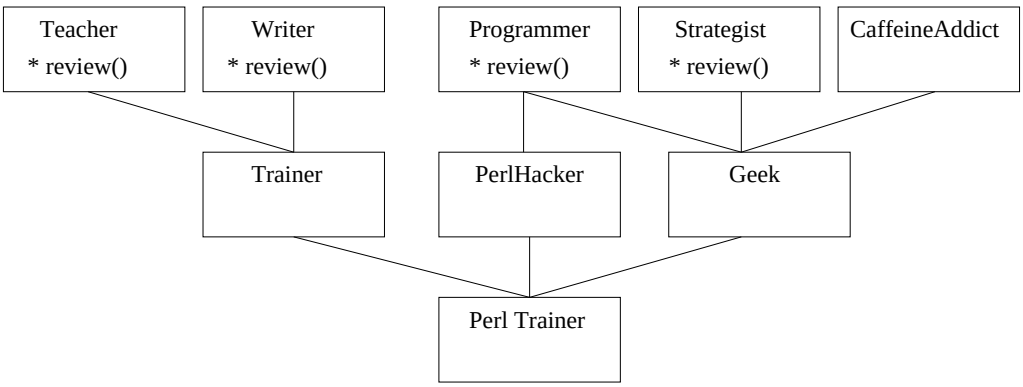
What happens if a number of ancestors have the method that's just been requested, and we end up calling the wrong one? For example:

```
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`.

```
package Teacher;
use NEXT;

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 `Writerss` 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:

```
package Writer;
use NEXT;

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:

```
package Writer;
use NEXT;

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.
}
```

NEXT::ACTUAL::DISTINCT works 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 initialisers to be called such that a child class completes initialisation before its parent.

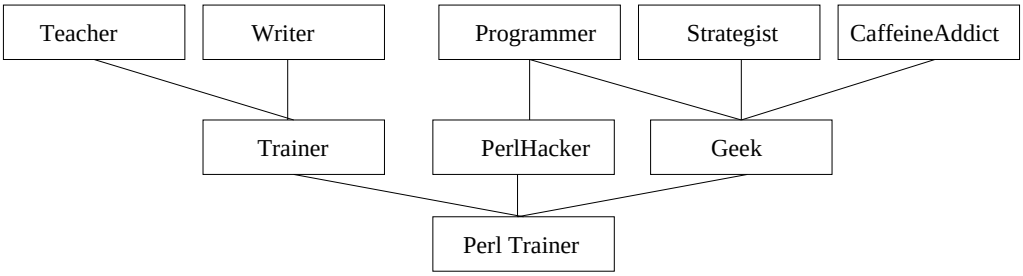
More recent versions of the `NEXT` module provide a new pseudo-class to overcome these problems, called `EVERY`. We'll cover this new pseudo-class in the next section.

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 where 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` or 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 is incomplete.  It does not check that its
# argument is meaningful.

sub set_location {
        my ($this, $location) = @_;
        $this->{_location} = $location;
}

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

        bless($this,$class);

        $this->EVERY::LAST::_init(@args);

        return $this;
}

# The _init function does specific initialisation for this class.
sub _init {
        my ($this, %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;

our @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 {
        my ($this,@args) = @_;
        return $this->can_move(@args);
}

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 (either directly or indirectly). 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 its 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.

Considerable thought needs to be given before creating `UNIVERSAL` methods. These affect all classes, including those that you did not write. Before writing any `UNIVERSAL` method, one should seriously consider if an alternative and more encapsulated approach may exist.

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. Moose - a postmodern object system for Perl 5

In this chapter...

Although it is possible to write Object Oriented code in Perl, as we've seen it can be a little laboured. Moose provides a solution to this by stealing ideas from Perl 6 and other object oriented languages and is rapidly becoming the de-facto standard for OO in Perl. This chapter shows many of the powerful features of using Moose for Object Oriented programming in Perl.

What is Moose?

Moose is a *complete* object system for Perl 5. It provides a simple and consistent syntax for attribute declaration, object construction and inheritance, without the need to understand how those things are implemented. Furthermore, Moose allows you to separate the use of your data from its underlying representation, thus you need not know whether Moose is using a hash, array or something else in which to store your data. This allows the programmer to focus on "what" the code is doing, rather than "how".

One chapter alone cannot hope to cover the full extent of what Moose has to offer. Other resources to return to are:

- The Moose is Flying (part 1) by Randal Schwartz
  http://www.stonehenge.com/merlyn/LinuxMag/col94.html
- The Moose is Flying (part 2) by Randal Schwartz
  http://www.stonehenge.com/merlyn/LinuxMag/col95.html
- Moose Manual http://search.cpan.org/perldoc?Moose::Manual
- Moose Cookbook http://search.cpan.org/perldoc?Moose::Cookbook

The basics

Using Moose gives us more easy accessors, clever classes, roles, types and coercions and loads of other cool stuff. However the rules we learned from our first object still apply.

A class is just a package

To create a class in Perl we create the package and use the `Moose` module:

```
package PlayingCard;
use Moose;

# we now have a Moose-based class
1;
```

That's it! Moose also turns on `strict` and `warnings` for us, so we don't have to do that ourselves, although nothing bad happens if we turn them on explicitly.

Classes in Moose can be used straight away, as it establishes a constructor for us automatically. Admittedly our class isn't very useful yet, but we can still create simple objects if desired:

```
use PlayingCard;

my $card = PlayingCard->new();          # works!
```

If you need your constructor to do more than just create an object with the given attributes then Moose provides some handy construction hooks: `BUILDARGS` and `BUILD`. To read more about these read `Moose::Manual::Construction`.

Methods are just subroutines

Just as in regular OO Perl, we add subroutines to our class in order to define object methods:

```
package PlayingCard;
use Moose;

sub show_card {
        my $self = shift;

        ....
}

1;
```

We then call them on our object, just as we did before:

```
use PlayingCard;

my $card = PlayingCard->new();          # works!

# Print out the card's value
print $card->show_card();
```

Attributes, basic types and accessors

One of the huge improvements that Moose provides is an easy and straight-forward way to handle accessors. We declare these with an English-like syntax and Moose handles everything else.

```
# Create our class (this also creates our constructor)
package PlayingCard;
use Moose;

# Add the suit attribute
has 'suit' => (
        is  => 'ro',    # Once card is created, suit won't change
        isa => 'Str',   # Suits will be strings
        required => 1,  # This attribute must be defined
);
```

```
# Add the value attribute
has 'value' => (
        is  => 'ro',    # Once card is created, value won't change
        isa => 'Str',   # Values will be strings
        required => 1,  # This attribute must be defined
);

# Returns the card name
sub show_card {
        my $self = shift;

        return $self->value() . " of " . $self->suit();
}

no Moose;       # turn off Moose-specific scaffolding

1;
```

Once we have created these, we now have our full PlayingCard object. We can make a whole bunch of cards, show their values, put them in a deck or whatever:

```
use PlayingCard;

my $card1 = PlayingCard->new(
        value => "ace",
        suit  => "hearts",
);

my $card2 = PlayingCard->new(
        value => "king",
        suit  => "spades",
);

print $card1->show_card();
print $card2->show_card();

# trying to change a value is not allowed
$card1->suit("diamonds");       # ERROR
```

Read-write or read-only

When we declare an attribute we *must* specify whether whether it `is` read-only (`ro`) or read-write (`rw`). Failing to do this will not cause a warning, but will result in the attribute not being usable which is a hard bug to track down. An attribute which is read-only can only be set when the object is created, while attributes which are read-write may have their value changed as required.

```
has 'suit' => (

        # Suit is read-only

        is  => 'ro',
        isa => 'Str',
);
```

In the case of our card, it doesn't make sense for a card to change either its suit or its value after creation time. However, in a game where cards can be face-up or face-down, it does make sense for these to change:

```
has 'visible' => (

        # visible is read-write
        is  => 'rw',
        isa => 'Bool',
);
```

Don't forget to specify whether your attribute is read-only or read-write.

Types

Moose provides a number of basic attribute types which we can use for our class attributes. These form a simple hierarchy, where any more specialised type may be used in the place of a more generalised type (for example you can use an integer (Int) instead of a number (Num) and either can be used where a string is expected. You can see this hierarchy by reading `perldoc Moose::Util::TypeConstraints` and it is also reproduced below:

```
Any
Item
    Bool
    Maybe[`a]
    Undef
    Defined
Value
    Str
        Num
            Int
        ClassName
        RoleName
Ref
    ScalarRef
    ArrayRef[`a]
    HashRef[`a]
    CodeRef
    RegexpRef
    GlobRef
        FileHandle
    Object
```

Any type followed by a type parameter `['a]` can be parameterised. Thus you can write:

```
ArrayRef[Int]    # an array of integers
HashRef[CodeRef] # a hash of str to CODE ref mappings
Maybe[Str]       # value may be a string, may be undefined
```

You can also create your own types and constraints, which we will cover later.

These are *not* a strong typing system for Perl 5. These are *constraints* to ensure that attributes for Moose-based objects contain what you expect them to.

Accessors

Once you have declared your attributes in your class, you automatically receive accessor functions for them. These are named by the attribute name and if your attribute is marked as read-write (`rw`) then you can use them to change the value as well:

```
my $card = PlayingCard->new(
        value      => "two",
        suit       => "clubs",
        decoration => "flowers",
);

print $card->suit();            # prints "clubs";

$card->decoration("money");     # decoration is now money
```

You can tell Moose to name your accessors differently if you want:

```
has 'decoration' => (
        is     => 'rw',
        isa    => 'Str',
        reader => 'get_decoration',
        writer => 'set_decoration',
);
```

Then later we call these as required:

```
$card->set_decoration("money"); # decoration is now money
print $card->get_decoration();  # prints "money"
```

Moose will still check the type constraints and run any triggers with methods named like this.

BUILDARGS and BUILD

For many objects, we would like to run code when an object is created. This may be to attach to a resource, run validation checks, or merely fill in some sensible defaults. Moose allows us to do this by defining our own `BUILDARGS` and `BUILD` methods.

The `BUILDARGS` method, if defined, is executed *before* the object is constructed. It's useful for tweaking our arguments before they hit Moose. One very common use of `BUILDARGS` is to support object construction with a non-hash syntax. For example, we may allow our playing cards to be constructed with a single argument (eg: 'Kh' for the king of hearts).

```
around BUILDARGS => sub {
    my ($orig, $class, @args) = @_;

    # $orig is Moose's (or our parent's) BUILDARGS
    # subroutine.  This allows us chain together
    # classes that all do their own argument handling.

    if ( @args == 1 and ! ref $args[0] ) {

        # If we only have one argument, and it's not a
        # reference, then extract our card information.

        my ($value, $suit) = $args[0] =~ /^(\w)(\w)$/;
```

```
        return $class->orig(
            value => $value,
            suit  => $suit,
        );

    }
    else {
        return $class->$orig(@args);
    }
};
```

The use of the `around` keyword is explaind more fully later in this chapter.

The `BUILD` method is called immediately after an object is constructed. It's often useful to perform extra validation steps, connect to a database, open a file, or otherwise do work that needs to occur at object construction. For example, in the game of *Pontoon*, tens are removed from the deck.

```
package PlayingCard::Pontoon;
use Moose;
extends 'PlayingCard';

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

    if ($self->value eq 'T') {
        die "Tens are not allowed in Pontoon\n";
    }
}
```

There is no need to call parental `BUILD` methods; in fact, it's a grave mistake to do so. Moose ensures that all `BUILD` methods are called. Parental `BUILD` methods are guaranteed to be called before their children.

Exercise

Implement the Coin class from earlier in Moose. You should only need to make minor changes to the script which uses that class.

Smarter types and coercions

Our existing `PlayingCard` class in Moose doesn't do any sort of validation of the card suit or value. While we could write our own code to manually validate these values, a much more flexible approach is to define our own types for Moose.

```
use Moose::Util::TypeConstraints;

my %String_to_suit = (
        s      => 's',
        spades => 's',
        spade  => 's',
        h      => 'h',
        hearts => 'h',
        heart  => 'h',
```

```
        ...
);

my %String_to_value = (
        ( map { lc($_) => $_ } (2..9, qw(T J Q K A) ) ),
        10    => 'T',
        jack  => 'J',
        queen => 'Q',
        king  => 'K',
        ace   => 'A',
);

subtype 'Suit'
        => as 'Str'
        => where { /^[hdcs]/ };

subtype 'CardValue'
        => as 'Str'
        => where { /^[2-9AKQJT]$/ };

coerce 'Suit'
        => from 'Str'
        => via { $String_to_suit{lc $_} };

coerce 'CardValue'
        => from 'Str'
        => via { $String_to_value{lc $_} };
```

The above code can be placed into any file, provided it is used before any attributes that depend upon it. In the case of our `PlayingCard` class, it could go into the same file as the class definition itself. For a more complex project, where types may be used by multiple classes, it's best to put type definitions into a separate file and `use` it from the classes that need them.

package PlayingCard;

```
use Moose;
use CardTypes;

has 'suit'  => (
        is       => 'ro',
        isa      => 'Suit',
        required => 1,
        coerce   => 1,
);
has 'value' => (
        is       => 'ro',
        isa      => 'CardValue',
        required => 1,
        coerce   => 1
);

no Moose;

1;
```

The `coerce` parameter is required to tell Moose that it's okay to coerce from a compatible type. Without this, Moose won't try to turn our strings into `Suits` and `CardValues`.

Exercise

Update your `Coin` class to provide its own type constraints for which side is up.

Inheritance

In regular Perl, inheritance is handled by either manipulating the `@ISA` array directly; or by using the `base` pragma. Moose abstracts those issues away with the `extends` keyword.

For example, consider the following inheritance tree:

```
Trainer
        PerlTrainer
```

We can create our `Trainer` class as follows:

```
package Trainer;
use Moose;

has 'name'      => ( is => 'rw', isa => 'Str', required => 1 );
has 'subject'   => ( is => 'rw', isa => 'Str', required => 1 );

no Moose;
1;
```

Inheriting from that is pretty easy:

```
package PerlTrainer;
use Moose;
extends 'Trainer';

no Moose;
1;
```

The `extends` keyword will load the class if needed, however it will also *overwrite* any previous inheritance information. Thus for multiple inheritance one must specify all parent classes in the one call to `extends`. For example:

```
extends 'Trainer', 'Geek';
```

At this point our two classes are effectively identical. The `PerlTrainer` class automatically gets the two attributes from `Trainer`. To make things more interesting, let's add a `review` method:

```
package PerlTrainer;
use Moose;
extends 'Trainer';

sub review_results {
        my ($self, $material) = @_;

        if   ( $material =~ /Perl/ ) {
                return "Awesome stuff!";
        }
```

```
        elsif( $material =~ /PERL/ ) {
                return "An abomination!";
        }
        else {
                return "Could be good, dunno.";
        }
}

no Moose;
1;
```

Now, we can call that from within our parent class:

```
package Trainer;
use Moose;

has 'name'      => ( is => 'rw', isa => 'Str', required => 1 );
has 'subject'   => ( is => 'rw', isa => 'Str', required => 1 );

sub review {
        my ($self, $material) = @_;

        return $self->name() . " replies: " .
               $self->review_results($material);
}

no Moose;
1;
```

What if there isn't a `review_results` method? We should add an abstract method to our `Teacher` class just in case:

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

        # Moose gives us confess (like Carp's confess) to complain
        return confess $self. " needs to be able to review materials!";
}
```

Now we can create PerlTrainers and review materials:

```
my $trainer = PerlTrainer->new( name => 'Paul', subject => 'Perl' );

say $trainer->review('Programming Perl, 3rd Ed');
```

While this is a traditional solution, it's not necessarily a good one, as we can still create abstract objects. A much better solution is to use *roles*, which we will learn about next.

Roles

In the previous example, our `Teacher` class is essentially an abstract class. We don't really intend to create a `Teacher` object at any point, instead we intend to add classes to represent each type of trainer we might have. In Moose this becomes a candidate for a *role*.

A role never has any instances, but exists to give functionality to one or more classes. To declare a role in Moose, we `use Moose::Role`. Otherwise, the code is almost identical:

```
package Trainer;
use Moose::Role;
```

```
has 'name'      => ( is => 'rw', isa => 'Str', required => 1 );
has 'subject'   => ( is => 'rw', isa => 'Str', required => 1 );

sub review {
        my ($self, $material) = @_;

        return $self->name(), " replies: ",
                $self->review_results($material);
}

# replace review_results sub with a requires statement:
requires 'review_results';

no Moose::Role;
1;
```

By replacing the abstract method with a requirement that `review_results` exists in the classes which use this role, we can tell at *compile time* whether the method is going to be missing rather than waiting until runtime when the abstract subroutine gets called.

To use a role, rather than extending another class, we use `with` instead of `extends`:

```
package PerlTrainer;
use Moose;
with 'Trainer';

sub review_results {
        my ($self, $material) = @_;

        if   ( $material =~ /Perl/ ) {
                return "Awesome stuff!";
        }
        elsif( $material =~ /PERL/ ) {
                return "An abomination!";
        }
        else {
                return "Could be good, dunno.";
        }
}

no Moose;
1;
```

Roles as a conceptual framework

Roles work well whenever it makes more sense to say our object *does* a particular behaviour rather than it *is a* particular thing. This is handy because a number of very different kinds of things may all have certain similar behaviours. For example, coffee and cola both *do* caffiene, but so does caffienated soap and caffienated lollies If we wanted to represent all of these, it would make sense to just have a `Caffientated` role which each of these `do`, and let coffee and cola also `extend` the `Beverage` class if it exists.

```
package Coffee;
extends 'Beverage';
with    'Caffeine';

package Cola;
extends 'Beverage';
with    'Caffeine';
with    'Carbonated';
```

```
    package Caffeinated::Soap;
    extends 'Soap';
    with    'Caffeine';
```

Once you've wrapped your mind around the idea of roles, you should find that roles make object oriented programming much easier. In fact, it's not unusual to see Moose project which have practically discarded inheritance in favour of compositing roles, as they avoid awkward abstract classes, and provide valuable compile-time checking.

Method modifiers: before, after and around

One of Moose's features is the easy ability to extend existing methods using the new keywords `before`, `after` and `around`. These avoid cumbersome walking of the inheritance tree, as well as allowing roles to modify existing methods.

Using `before` allows us to inject code before the role method is called, `after` allows us to inject code after the role method is called and `around` allows us to do both. This is useful if we want to emit debugging information, log something to a file, pre-open a file, change `@_` or any of many things. In the following case we can demonstrate how these might help us provide some extra debugging information

```
# Logger.pm
package Logger;
use Moose::Role;

sub log {
        my ($self, $fh, $message) = @_;

        print {$fh} $message;
}

no Moose::Role;

1;

##########

# Logger/Debug.pm
package Logger::Debug;
use Moose;
with 'Logger';

before 'log' => sub {
        print STDERR "About to log\n";
};

after 'log' => sub {
        print STDERR "Finished logging\n";
};

around 'log' => sub {
        my $next = shift;
        my $self = shift;

        print STDERR "  Around the call to log\n";
```

```
        # Pass in a handle to STDERR instead
        # Pass in our own message:
        $self->$next( \*STDERR, "    Inside log\n" );

        print STDERR "  Still around the call to log\n";
};

no Moose;

1;
```

Now if we use this in our program:

```
#!/usr/bin/perl -w
use strict;
use warnings;
use autodie;
use Logger::Debug;

open (my $fh, ">", "/tmp/logfile.txt");

Logger::Debug->new->log($fh, "test\n");
```

we will get:

```
About to log
 Around the call to log
   Inside log
 Still around the call to log
Finished logging
```

As you can see, first any `before` modifiers are called, then any `around` modifiers (which should themselves call the desired method but may not) and then finally the `after` modifiers.

In this example our `around` call supplants the original arguments passed to log and instead creates its own. This is useful for debugging and testing, but obviously not useful as a general rule.

`before`, `after` and `around` are Moose methods which take two arguments; the name of the method they are modifying and a subroutine reference containing the modification. As such it is important to remember to complete each method call with a semi-colon.

```
# Wrong, missing final semi-colon (syntax error)
before 'log' => sub { print "before"; }

# Correct
before 'log' => sub { print "before"; };
```

Method resolution order (MRO)

In standard Perl 5 OO, methods are resolved using the rule that the left-most-ancestor wins. Thus the first method found in a depth-first search through your object's hierarchy will be the only method called unless it is using the `NEXT` module to pass the request back to the dispatcher.

One problem with this rule is that in some circumstances a class' method may be called before its subclass has had an attempt. For example if we have the following diamond inheritance:

```
       Programmer
          / \
  PerlHacker  Geek
          \   /
       PerlTrainer
```

Then our methods will be resolves as: PerlTrainer, PerlHacker, Programmer, Geek.

We may however, want to ensure that all subclass methods get called before their superclass and thus have these methods called as: PerlTrainer, PerlHacker, Geek, Programmer.

C3 Method Resolution and Moose

The `mro` pragma (or `MRO::Compat` for Perl 5.6 and 5.8) allows you to choose which method resolution order you would prefer. You can choose between Perl's default (depth-first-search: `dfs`) and C3 (`c3`).

C3 always preserves local precedence ordering. Thus in the example above, C3 ordering gives us the latter choice, where all subclass methods are called before their superclass.

Moose uses C3 method ordering by default.

For more information on C3 and MRO choices, read `perldoc mro` for Perl 5.10 and above or `perldoc MRO::Compat` for Perls 5.6 and 5.8.

Passing things on

The `mro` pragma (but not `MRO::Compat`) provides three useful methods in a similar vein to the `NEXT` module.

`next::method`

> Calls the next method of the same name in the C3 MRO. Throws an exception if there are no more methods of that name.

`next::can`

> Returns a code reference to the next method in the C3 MRO if it exists, `undef` otherwise.

`maybe::next::method`

> Combines `next::method` and `next::can` to ensure that the method is only called if it exists. Similar to writing:
>
> ```
> $self->next::method(@_) if $self->next::can;
> ```

We can use use these methods from within our Moose classes too (so long as we have Perl 5.10 or above).

```
# PerlTrainer.pm
package Programmer;
use Moose;
```

```
sub review {
        my $self = shift;
        print "Programmmer review\n";
        $self->maybe::next::method(@_);
}

no Moose;

package PerlHacker;
use Moose;
extends 'Programmer';

sub review {
        my $self = shift;
        print "PerlHacker review\n";
        $self->maybe::next::method(@_);
}

no Moose;

package Geek;
use Moose;
extends 'Programmer';

sub review {
        my $self = shift;
        print "Geek review\n";
        $self->maybe::next::method(@_);
}

no Moose;

package PerlTrainer;
use Moose;
extends 'PerlHacker', 'Geek';

sub review {
        my $self = shift;
        print "PerlTrainer review\n";
        $self->maybe::next::method(@_);
}

no Moose;

1;
```

and in our calling code:

```
use PerlTrainer;

PerlTrainer->new()->review();
```

This gives us:

```
PerlTrainer review
PerlHacker review
Geek review
Programmer review
```

as expected. If we were certain that our inheritance tree was finalised, we may have used `next::method` instead of `maybe::next::method`.

For Perl 5.10 and later, the use of `mro` is preferred over `NEXT`.

Chapter summary

- Moose is a complete object system for Perl 5.
- Perl's basic OO rules still apply. To create a class, we just create a package. To create a method, we create a subroutine.
- Moose automatically creates constructors for our classes.
- To create an attribute, we use the `has` method. We must remember to specify whether each attribute is read-only or read-write.
- Moose automatically creates get accessors for each attribute.
- Moose comes with a range of basic types which attributes might be and other types can be created.
- We use `extends` to denote inheritance.
- Roles provide a useful alternative to a complex inheritance tree. We use `with` to denote role usage.
- `before`, `around` and `after` allow us to add modifiers to methods.
- Moose uses C3 method resolution order by default. This is achieved by the use of the `mro` pragma on Perl 5.10+ and `MRO::Compat` for Perl 5.6 - 5.8.
- On Perl 5.10+ we can use `mro`'s dispatch methods to pass method calls around correctly.

Chapter 15. 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 16. 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.

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

```
# Package to manage all of my different pet types
package Pet;

# Overload stringification and !
use overload (
        q("")   => "as_string",
        q(++)   => \&praise_pet,     # Subroutine reference!
);

#  Simplistic constructor
sub new {
        return bless {name=> $_[1]}, shift;
}

# Used for stringification (default method)
sub as_string {
        my $self = shift;

        my $string = "$self->{name} is my pet. ";
        if($self->{praise}) {
                $string .= "I have praised $self->{name} $self->{praise} ".
                           "times today.";
        }
        return $string;
}

# Used for ++ (always used!)
sub praise_pet {
        my $self = shift;
        $self->{praise}++;
        return;
}

#-------------------------------------
# Special case of pets, a dog class
package Pet::Dog;

our @ISA = qw(Pet);     # We inherit from Pet.

# Define our own stringify method
sub as_string {
        my $self = shift;

        my $string = "$self->{name} is my dog! ";
        if($self->{praise}) {
                $string .= "I have praised my dog $self->{name} ".
                            "$self->{praise} times today!";
        }
        return $string;
}

# Dogs earn double praise!
sub praise_pet {
        my $self = shift;
        $self->{praise} += 2;  # Good dog!
        return;
}
#-------------------------------------
```

```
#!/usr/bin/perl
# Use my Pet classes;
use strict;
use warnings;
use Pet;
use Pet::Dog;

# Define a generic pet, Sally
my $sally = Pet->new("Sally");

print "$sally\n";
# Sally is my pet.

$sally++;
print "$sally\n";
# Sally is my pet. I have praised Sally 1 times today.

# Define a dog, Fido.
my $dog = Pet::Dog->new("Fido");

print "$dog\n";
# prints  Fido is my dog!

$dog++;
print "$dog\n";
# Fido is my dog! I have praised my dog Fido 1 times today!
```

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;
use List::Util qw/shuffle/;

# Assume we've implemented the deck class method, to return a full
# deck of cards.
my @deck = PlayingCard->deck();

# Shuffle...
@deck = shuffle @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";
}
```

Exercise

1. Overload the <=> operator for your `PlayingCard` object.
2. Revisit your `war` program and make use of this new overload.

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 17. 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 are *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 will 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. In most programs 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 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.

From version 5.10.1, Perl comes standard with the `autodie` module which makes all Perl's built-ins throw exceptions on failure. These changes are lexically scoped. `autodie` can be downloaded from the CPAN for Perl 5.8.0 and above.

```
use autodie;

# Failed calls to open and close will now throw exceptions

open(my $fh, "<", $filename);   # No need for 'or die...'

while (<$fh>) {
        print;
}

close($fh);   # No need for 'or die' here, either.
```

It's also possible to promote Perl's in-built warnings into errors. This means that we can generate exceptions from performing an unwise operation like using an undefined value as if it were defined, or reading/writing to a closed filehandle.

```
eval {
        use warnings FATAL => qw(all);

        print "Give me a number\n";
        my $num = <STDIN>;
        my $num2 = $num + 10;
        print "$num + 10 = $num2\n";
};

if ($@) {
        warn "Input handling failed - $@";
}
```

Using both Perl's `warnings` and `autodie` modules in conjunction can allow us to more cleanly use exception-based code.

You can learn more about having perl built-ins generate exceptions by reading **perldoc autodie** and **perldoc perllexwarn**.

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

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

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

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.

Try::Tiny

Yuval Kogman's excellent `Try::Tiny` module not only provides extra syntactic sugar for handling exceptions, but also fixes many of the issues commonly associated with Perl's exception handling mechanisms.

Using `Try::Tiny` in your programs is easy, and the use of the `try` and `catch` statements is quite intuitive:

```
use Try::Tiny;

my $CONFIG = "Config.txt";

try {
        use autodie;

        open(my $fh,'<',$CONFIG);

        while (<$fh>) {
                process($_);
        }
}
```

```
catch {
        warn "Caught error: $_";
};
```

One can also use a `try` without a corresponding `catch` to ignore exceptions in a given block.

For a more heavyweight and featureful exception handling system, look at the `TryCatch` module available from the CPAN.

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.
- Using Perl's `autodie` pragma allows Perl's built-ins to throw exceptions.
- Some modules, such as `DBI` allow the programmer to utilise exceptions very well to detect and handle errors.
- The `Try::Tiny` module provides both a nicer syntax for exception handling, as well as also solving many edge-cases that can arise.

Chapter 18. 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.
- 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

- Perl Training Australia (http://perltraining.com.au/)
- The Australian Perl Portal (http://perl.net.au/)
- The Perl homepage (http://www.perl.com/)
- Perl Mongers Perl user groups (http://www.pm.org/)
- PerlBuzz news (http://perlbuzz.com/)
- PerlMonks online community (http://www.perlmonks.org/)
- Perl 5 wiki (http://www.perlfoundation.org/perl5)
- Comprehensive Perl Archive Network (http://www.cpan.org)

Appendix A. Inside-out objects

In this chapter...

The de-facto choice for Perl objects has been to use blessed hashes. A hash makes it easy to both add new attributes and access existing ones. Unfortunately, this ease of access is also one of the greatest problems with a hash-based structure. In this chapter we'll cover an alternative Perl object structure known as an *inside-out object*.

Problems with blessed hashes

In a perfect world everyone would obey the rules and only use the documented interface for each class. Unfortunately, the world isn't always perfect. Maybe a developer will bypass the interface to try and squeeze some extra performance out of their code. Maybe they used `Data::Dumper` to inspect your object and wrote their code to extract attributes without *ever* reading your documentation.

When you change your object's implementation, any code that bypasses your documented interface will break. Of course, the miscreant developer who wrote the bad code was fired years ago for not conforming to coding guidelines, but it's *your* changes that just caused the system to break. Even if you can convince your boss that it isn't your fault, it will still be your job to make things work.

The other problem with using hashes comes down to simple typographical errors. Let's pretend that one of your attributes is `address`, but somewhere in your code you make an accidental typo, forgetting a 'd': `adress`:

```
sub get_address {
        my ($this) = @_;
        return $this->{address};
}

sub set_address {
        my ($this, $value) = @_;
        $this->{adress} = $value;       # Oops!
}
```

The above code doesn't result in a warning. Perl is perfectly happy to add a new element to our hash, but since nothing else refers to the key it will never be used, resulting in a difficult to find bug.

Wouldn't it be great if we could have *compile-time* checking of attributes, rather than relying upon run-time checks and the correctness of developers? With *inside-out objects*, we can.

This was the primary problem that pseudo-hashes were designed to avoid. Unfortunately pseudo-hashes slowed everything down, so they were removed.

What is an inside-out object?

Inside-out objects are known by many names, including flyweight objects and inverted indices. Rather than storing all of our attributes inside our single object, we instead have a single hash for

each attribute, and our object has an entry in each hash. The following example demonstrates the differences in structure:

```
# Traditional hash-based objects.

$person1 = { firstname => "Paul",    surname => "Fenwick"    };  # Object 1
$person2 = { firstname => "Jacinta", surname => "Richardson" };  # Object 2
$person3 = { firstname => "Damian",  surname => "Conway"     };  # Object 3

# Inside-out objects.

       # Object 1          # Object 2             # Object 3
%firstname = ( 12345 => "Paul",    23456 => "Jacinta",    34567 => "Damian" );
%surname   = ( 12345 => "Fenwick", 23456 => "Richardson", 34567 => "Conway" );
```

Error checking

Inside-out objects provide excellent error checking, because if we make a mistake in writing an attribute name we receive an error *at compile time*:

```
use strict;
use Class::Std;

my %address;

# ...

sub set_address {
        my ($this, $value) = @_;
        $adress{ident $this} = $value;          # Oops!
}

# Trying to compile the above code results in an error:
# Global symbol "%adress" requires explicit package name at ...
```

Automatic attribute checking is a big improvement in preventing what is otherwise a very common and frustrating bug. However the benefits don't stop there. Inside-out objects provide much better encapsulation than regular hash based objects.

Strong encapsulation

An inside-out object contains none of its own data; instead this has been moved into a series of hashes that are stored inside the class. By ensuring these are declared lexically (using `my %attribute`) we can be sure that nothing outside of the class is able to access these attributes.

Strong encapsulation means that a misguided developer can't bypass our interface and access attributes directly. There's simply no way that external code can access those attributes. They're simply not in scope.

Attribute access

We connect our attributes to our object by using a unique key. Since we're trying to ensure object integrity, our ideal key would be fixed and unchangeable for each object. The simplest solution would be to give each object a sequential number upon generation and mark it as read-only. Unfortunately this would make it very easy for external code to guess possible key values and break

encapsulation. Ideally we want our key to be hard to fake. One solution is to use a module such as `Data::UUID` which generates globally unique identifiers. Another is to realise that every Perl variable already comes with something unique and verifiable -- its memory address.

The `Scalar::Util` module provides us with the `refaddr` function, which returns the memory address pointed to by a given reference. The `Class::Std` module, which we will be examining shortly, provides exactly the same function named `ident` (since the memory address is used as an identifier for the object).

Inside-out playing cards

We now have enough information to build ourselves our very own inside-out object. Let's see our `PlayingCard` turned inside-out:

```
package PlayingCard;
use strict;
use warnings;
use Scalar::Util qw/refaddr/;

# Using an enclosing block ensures that the attributes declared
# are *only* accessible inside the same block.  This is only really
# necessary for files with more than one class defined in them.
{
        my %suit_of;
        my %rank_of;
                                                                #
        sub new {
                my ($class, @args) = @_;
                                                                #
                # This strange looking line produces an
                # anonymous blessed scalar.
                                                                #
                my $this = bless \do{my $anon_scalar}, $class;
                                                                #
                return $this->EVERY::LAST::_init(@args);
        }

        sub _init {
                my ($this, $rank, $suit) = @_;
                                                                #
                # Attributes are stored in their respective
                # hashes.  We should also be checking that
                # $suit and $rank contain acceptable values.
                                                                #
                $suit_of{refaddr $this} = $suit;
                $rank_of{refaddr $this} = $rank;
                                                                #
                return $this;
                                                                #
        }
                                                                #
        sub get_suit {
                my ($this) = @_;
                return $suit_of{refaddr $this};
        }
                                                                #
        sub get_rank {
                my ($this) = @_;
                return $rank_of{refaddr $this};
        }
}
```

\do{my $anon_scalar}

One of the strangest lines in our code contains `\do{my $anon_scalar}`. This odd construct simply declares a lexical variable using `my`. The name of our scalar is irrelevant, since it immediately goes out of scope at the end of the block. Normally this would seem fruitless, but the enclosing `do {}` block returns the last statement evaluated, in our case the freshly created scalar. By taking a reference to this scalar (using the backslash operator) our scalar avoids destruction and lives on without a name.

Note that our scalar itself is completely empty, it doesn't contain anything, and we never use its contents. It exists simply to be blessed into the appropriate class, and for our own code to use its memory address for attribute lookups.

Exercise

1. Write an address entry class (`Address.pm`) using inside-out-objects. The constructor will be called as follows:

   ```
   my $address = Address->new(
                   surname   => $surname,
                   firstname => $firstname,
                   street    => $street_address,
                   postal    => $post_address,
                   phone     => $phone,
   );
   ```

   If the post address is not set, it should be given the same value as the street address. A starter is available in `exercises/lib/Address.pm`);

2. Run the `exercises/lister.pl` program to ensure that your class works as expected.

An answer can be found in `exercises/answers/lib/Address.pm`.

A problem with inside-out objects

Inside-out objects compare favourably with regular objects. They scale better in terms of memory usage, and with minor modifications can be tuned to provide even faster performance, albeit with the loss of some integrity benefits. However you're unlikely to notice these benefits unless it's absolutely critical that your application needs to run very fast or very small. So what's the catch?

Think about what happens when an object is destroyed. With a regular hash-based object the only reference to the object's attributes is lost with the object itself, and Perl handles the clean-up for us. When we have an inside-out object, nothing cleans up the attributes when the object is destroyed. Instead, we have to write our own `DESTROY` method. We also need to worry about making sure our parent and sibling `DESTROY` methods are called as well. If we don't, then our objects will leak memory, and that's bad.

For our `PlayingCard` we would need to add the following, or code like it:

```
use NEXT;

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

sub _destroy {
        my ($this) = @_;
        delete $suit_of{ident $this};
        delete $rank_of{ident $this};

}
```

All our derived classes will need to write their own `_destroy` method to clean up any additional attributes that have been defined.

Inheritance and attributes

An additional advantage of inside-out objects is that each class has its own private area in which to store attributes. This means that derived classes don't need to worry about clashes with parents or siblings, and vice-versa. It also makes it possible, although possibly unwise, for derived classes to have attributes of the same name, but with different values (something which is impossible for standard hash-based objects).

If we do decide to use attributes of the same name in more than one class in our inheritance tree, we need to think about how we will ensure that each class gets the correct value during construction and initialisation. The best way to do this depends on our implementation, and will be discussed further in the next chapter.

Inside-out objects do not avoid the problems associated with multiple methods in the inheritance tree having the same names. Fortunately, we can use `NEXT` in such situations, just as we do with standard hash-based inheritance.

Helper modules

The basic structure of any inside-out object is essentially the same, just as the basic structure for hash-based objects is essentially the same. As such a number of builder modules have been created to remove the repetitive code and make it quicker for you to start writing the real code. Two particularly good modules for inside-out objects are:

- `Class::Std`
- `Object::InsideOut`

We will talk more about the first of these in the next chapter.

Chapter Summary

- Blessed hashes have *weak encapsulation*, allowing for the potential that external code will bypass your interface and access attributes directly.
- Blessed hashes have an additional problem where a typo in a hash lookup can result in a silent error. These bugs can be difficult to catch.
- Inside-out objects provide *strong encapsulation* by using lexically scoped hashes. No data is stored inside the object proper.
- By using lexical hashes inside-out objects provide compile-time attribute checking.
- Inside-out objects require the creation of `DESTROY` methods to avoid memory leaks.
- Inside-out objects can allow parent or sibling classes to have attributes with the same name but distinct values. However correctly initialising these attributes may be challenging.
- Both the `Class::Std` and `Object::InsideOut` modules are excellent helpers for creating your own inside-out objects.

Appendix B. Building classes with Class::Std

In this chapter...

Building a new class is often a repetitive exercise. We need a constructor that somehow produces an object; for inside-out objects we need a destructor to clean things up; for almost all classes we'll need accessors to get and set various attributes. For the bulk of classes, much of this code is going to be practically identical, with simply a few labels changed here and there.

The `Class::Std` module provides a way to avoid a lot of the boring and repetitive work in producing a new class. It provides default constructors, destructors, allows for auto-generation of accessors, and provides a convenient way of tagging attributes. This chapter will cover these aspects of `Class::Std`.

Using Class::Std

When using `Class::Std` we can choose to use as much or as little functionality as needed, however all such classes need to start by loading `Class::Std`:

```
package PlayingCard;
use Class::Std;
```

The `use` definition *must* be under the package line, as `Class::Std` exports extra functions into the current package.

Object creation

We can create a new object just like we create any other object:

```
my $card = PlayingCard->new({ suit => "spades", rank => "ace" });
```

Notice here that we're passing a hash reference rather than just named parameters. This is a requirement of `Class::Std`.

Defining Attributes

Almost all objects will have attributes, and when working with inside-out objects (which `Class::Std` supports) each attribute will have its own separate hash:

```
package PlayingCard;
use Class::Std;

my %suit_of :ATTR;
my %rank_of :ATTR;
```

Note that we've used the special tag `:ATTR` to indicate these hashes are used for storing attributes. One advantage of this approach is that `Class::Std` will now automatically tidy up the data in these hashes when an object is destroyed.

If we're declaring a lot of attributes, then we can use the `:ATTRS` tag like this:

```
my (
        %name_of,
        %address_of,
        %age_of,
        %favourite_food_of,
) :ATTRS;
```

Object construction

We've already covered why it's a good idea to separate object construction from initialisation. The `Class::Std` module provides a standard `new` method that handles construction, meaning we avoid having to remember the strange syntax for creating an anonymous scalar. The `new` method also arranges for all initialisation routines to be called in both parent and sibling classes, meaning we don't need to worry about using `NEXT` to set this up manually.

The default initialisation routine called by `Class::Std`'s `new` method is called `BUILD`. The `BUILD` method is passed the object, the identifier, and a reference to the argument list. Our new PlayingCard now looks like this:

```
package PlayingCard;
use Class::Std;

my %suit_of :ATTR;
my %rank_of :ATTR;

sub BUILD {
        my ($self, $ident, $args_ref) = @_;
        $suit_of{$ident} = $args_ref->{suit};
        $rank_of{$ident} = $args_ref->{rank};
}
```

`Class::Std` actually calls a *second* initialisation routine called `START` if it exists. The difference between the two is that `START` is called after any automatic attribute initialisation has been finished, whereas `BUILD` is called before hand. To help remember the order, remember that first you have to `BUILD` the object (such as a car), before you can `START` it.

It's perfectly appropriate for a class to have both `BUILD` and `START` methods.

Automatic accessors

Writing accessors for our objects can be dull. For our playing card we could write the following:

```
sub get_suit {
        my ($this) = @_;
        return $suit_of{ident $this};
}

sub get_rank {
        my ($this) = @_;
        return $rank_of{ident $this};
}

sub set_suit {
        my ($this, $suit) = @_;
        suit_of{ident $this} = $suit;
}
```

```
sub set_rank {
        my ($this, $rank) = @_;
        $rank_of{ident $this} = $rank;
}
```

However this rapidly gets tedious. All accessors are in exactly the same form, and if we have many attributes we wish to access, then writing all these accessors gets very boring very quickly.

Luckily, we can use `Class::Std` to provide us with automatic accessors. Now we don't need to write any accessors at all! Let's see how.

Read accessors

To create an automatic read accessor we simply pass a `:get` option to the `:ATTR` tag when declaring our attribute:

```
package PlayingCard;
use Class::Std;

my %suit_of :ATTR( :get<suit> ); # Builds get_suit()
my %rank_of :ATTR( :get<rank> ); # Builds get_rank()
```

Read accessors generated with `Class::Std` always begin with `get_`.

Write accessors

For a simple attribute that can be set freely to any value, we can use the `:set` option of `:ATTRS` that allows `Class::Std` to create an automatic write accessor. For example:

```
package PlayingCard;
use Class::Std;

# Builds get_suit(), set_suit()
my %suit_of :ATTR( :get<suit> :set<suit> );

# Builds get_rank(), set_rank()
my %rank_of :ATTR( :get<rank> :set<rank> );
```

Write accessors generated with `Class::Std` always begin with `set_`.

Automatic initialisation

For a simple class where attributes are stored without any sort of validity checking, `Class::Std` can be used to provide automatic initialisation as well. This is also done by adding the `:init_arg` argument to the `:ATTR` tag:

```
package PlayingCard;
use Class::Std;

my %suit_of :ATTR( :get<suit> :set<suit> :init_arg<suit> );
my %rank_of :ATTR( :get<rank> :set<rank> :init_arg<rank> );

# No 'BUILD' required at all!
```

The code above will look for `suit` and `rank` parameters being passed to `new`, and will use them automatically for attribute initialisation. If these attributes are not provided at construction time then `Class::Std` will throw an exception that required parameters have not been set.

If we want to, we can use both `:init_arg`s and `BUILD`. The `BUILD` method is invoked before automatic initialisation occurs, which allows us to validate our arguments are correct. If we find the arguments are not correct we can throw an exception or revert to defaults. We can also use `BUILD` for initialisation of more complex attributes, while leaving trivial initialisation to `:init_arg`.

:name

Writing `:ATTR( :get<attr> :set<attr> :init_arg<attr> )` for each attribute can get a little repetitive. As a result there's a short cut! If we wish to use both automatic accessors and the automatic initialiser for an attribute, we can instead use `:ATTR( :name<attr> )`.

```
package PlayingCard;
use Class::Std;

my %suit_of :ATTR( :name<suit> );
my %rank_of :ATTR( :name<rank> );

# Still no 'BUILD' required!
```

Default values

Regardless of whether or not we use automatic or manual initialisation, we can specify a default value for a particular attribute by using the `:default` switch:

```
package PlayingCard;
use Class::Std;

# Our default suit is spades.

my %suit_of :ATTR( :name<suit> :default<spades> );
```

Note that only literal strings and numbers can be used as default values. This is a limitation in Perl's attribute handling, and not a limitation in `Class::Std`. If you want to use more complex defaults, then you may choose to not use the `:default`, `:name` or `:init_arg` switches; calculate and set your defaults in `BUILD` or `START` instead.

Summary of :ATTR options

Table B-1. Summary of :ATTR options

Option	Description
:init_arg<key>	Automatically initialise this attribute the argument specified.
:default<value>	Use the default value specified for this attribute.

Option	Description
:get<name>	Automatically create a read accessor, prefixed with get_ . For example, :get<flavour> produces an accessor named get_flavour().
:set<name>	Automatically create a write accessor, prefixed with set_ .
:name<name>	Shorthand for setting :init_arg, :get, and :set all at once.

Exercise

1. Create a new class called `Address2.pm` in your `exercises/lib` directory.
2. Create attribute hashes and any further code necessary.

   (Hint, you'll probably need to declare a `:default` value for the postal field and set this to the street value in a `START` block.
3. Change the `exercises/lister.pl` program to use your new Address2 class, and to construct the object properly.

Object destruction

Provided we have made use of the `:ATTR` tag to indicate our attributes, objects built with `Class::Std` do not require any special destructors to free attributes. Of course, sometimes we'll want to write our destructors for other reasons, such as to correctly releasing external resources such as file, database, or network connections.

`Class::Std` will automatically call the `DEMOLISH` method (if it exists) before an object is destroyed. This occurs for all classes in the object hierarchy, starting with the child classes, and walking up to the parents. This avoids the extra work of needing to use `NEXT` from our destructors.

`DEMOLISH` is called before any attributes are removed, so they can still used within the `DEMOLISH` method.

```
package PlayingCard;
use Class::Std;

my %suit_of  :ATTR( :name<suit> );
my %rank_of  :ATTR( :name<rank> );

# Our example start method simply prints a note to
# notify us of object creation.  We show how our
# START method is also passed $ident and $args, even
# though in this example we don't use them.

sub START {
        my ($this,$ident,$args) = @_;
        print "Created card: ",$this->as_string,"\n";
}
```

```
# Returns our card as a string.  Eg, "ace of spades"
sub as_string {
        my ($this) = @_;
        return join(" ",$this->get_rank,"of",$this->get_suit);
}

# Our demolish code is called at object destruction,
# before attributes have been cleared.  Here we just
# print a simple message noting object destruction.
# Clearing of attributes is done automatically by Class::Std.
# We also show that this method is passed ident as its second
# argument, even though we don't use it in this example.

sub DEMOLISH {
        my ($self, $ident) = @_;
        print "Discarded card: ",$self->as_string;

}
```

Debugging features

One of the difficulties that can be encountered when using inside-out objects is that not all common debugging techniques work like we would expect. A common debugging practice is to use `Data::Dumper` to display the contents of an object; however with an inside-out object this method provides no information whatsoever.

`Class::Std` automatically provides a `_DUMP` method that returns a string representation of the object, and is suitable for where we would normally call `Data::Dumper` for debugging. It should be noted that `_DUMP` only displays attributes that have been explicitly tagged with `:ATTR` flags.

Using `_DUMP` is generally not suitable for object serialisation. The special `Class::Std::Storable` module can be used to implement serialisable inside-out objects.

Method traits (overloads)

`Class::Std` provides a simple way to describe the behaviour of our objects when they are treated as particular types, including both basic types (string, numeric, boolean) and as references. For example, we can indicate that our `as_string` method should be used whenever our object is used as a string:

```
# Returns our card as a string.  Eg, "ace of spades"

sub as_string :STRINGIFY {
        my ($this) = @_;
        return join(" ",$this->get_rank,"of",$this->get_suit);
}

# Later, in our main code...

my $card = PlayingCard->new({ suit => "spades", rank => "ace" });
print "I have the $card!\n";
```

It does not make sense to nominate more than one method to be used for these types. Any of the following flags can be added to a method to indicate it should be used for a particular type overloading operation.

Table B-2. Type overloading flags

Option	Description
:STRINGIFY	Use this method when object is treated as a string.
:NUMERIFY	Use this method when object is treated as a number.
:BOOLIFY	Use this method when object is tested as true or false.
:SCALARIFY	Use this method when object is treated as a scalar reference.
:ARRAYIFY	Use this method when object is treated as an array reference.
:HASHIFY	Use this method when object is treated as a hash reference.
:GLOBIFY	Use this method when object is treated as a glob reference.
:CODIFY	Use this method when object is treated as a code reference.

An example of when you might find it useful to use the HASHIFY trait is where your object essentially represents a list of parameters and values which you intend to pass to something like `HTML::Template` or `HTML::FillInForm`. In these cases, it would be nice to be able to pass the object *as if it were a hash* to the appropriate method:

```
$template->params( %$object );
```

Fortunately we can write a method to allow our inside out object to do the right thing when treated as above.

```
# Returns reference to a hash-based representation of our object.

sub as_hash : HASHIFY {
        my ($this) = @_;

        return {
                rank => $this->get_rank,
                suit => $this->get_suit,
        };
}
```

Issues with Class::Std

`Class::Std` is a powerful tool to ensure proper encapsulation with Perl objects. However, it is not without its drawbacks. In order to ensure constructors and destructors are called correctly, `Class::Std` assumes that all parent classes also use `Class::Std`. If this is an issue the similarly featured `Object::InsideOut` module may be a better choice.

Another common complaint about `Class::Std` is more a complaint against inside-out objects in general. A common debugging strategy with Perl objects is to use `Data::Dumper` to print the object out. With inside out objects, the result of this is often of the form:

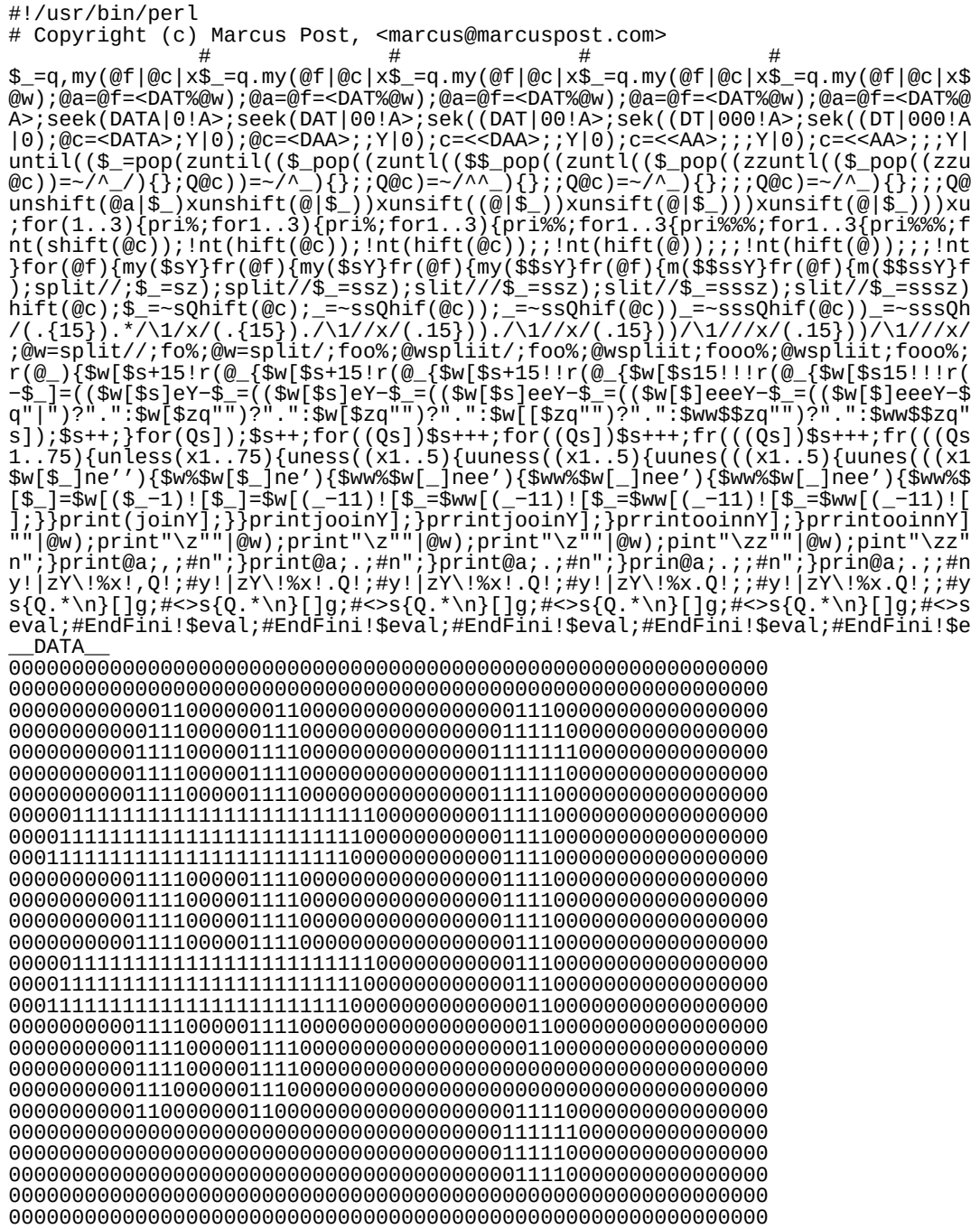
```
$VAR1 = bless( do{\(my $o = undef)}, 'Address' );
```

which is much less useful than the result with traditional hash based structures. This can be overcome by using the `_DUMP` method provided by `Class::Std`.

Chapter Summary

- Class::Std provides a way to avoid a lot of the repetitive work in producing a new class.
- Class::Std requires that we pass in a hash reference of our parameters and values.
- Class::Std marks our attribute hashes with `:ATTR`.
- The `BUILD` method is called to allow any changes to the incoming parameters once the object has been created, it is called before all automatic initialisation.
- The `START` method is called after automatic initialisation has occurred.
- Class::Std will automatically generate read and write accessors as well as perform automatic initialisation if requested. The `:name<>` attribute allows us to set all three.
- Default values (strings and numbers) can be set for each attribute using the `<default<>` attribute.
- Class::Std automatically cleans up the entries in the attribute hashes upon object destruction. Should further work be done, this can be done in the `DEMOLISH` method.
- To print the contents of a Class::Std object in a similar way as Data::Dumper would, use the `_DUMP` method.
- Method traits can be added to methods to provide overloads for your object. For example, the method with the `NUMERIFY` trait is called when the object is treated as a number.
- Class::Std assumes all classes in the inheritance hierarchy use Class::Std.

Colophon



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/).