Intermediate Perl

Dave Cross
Magnum Solutions Ltd
dave@mag-sol.com

What We Will Cover

- Types of variable
- Strict and warnings
- References
- Sorting
- Reusable Code
- Object Orientation

What We Will Cover

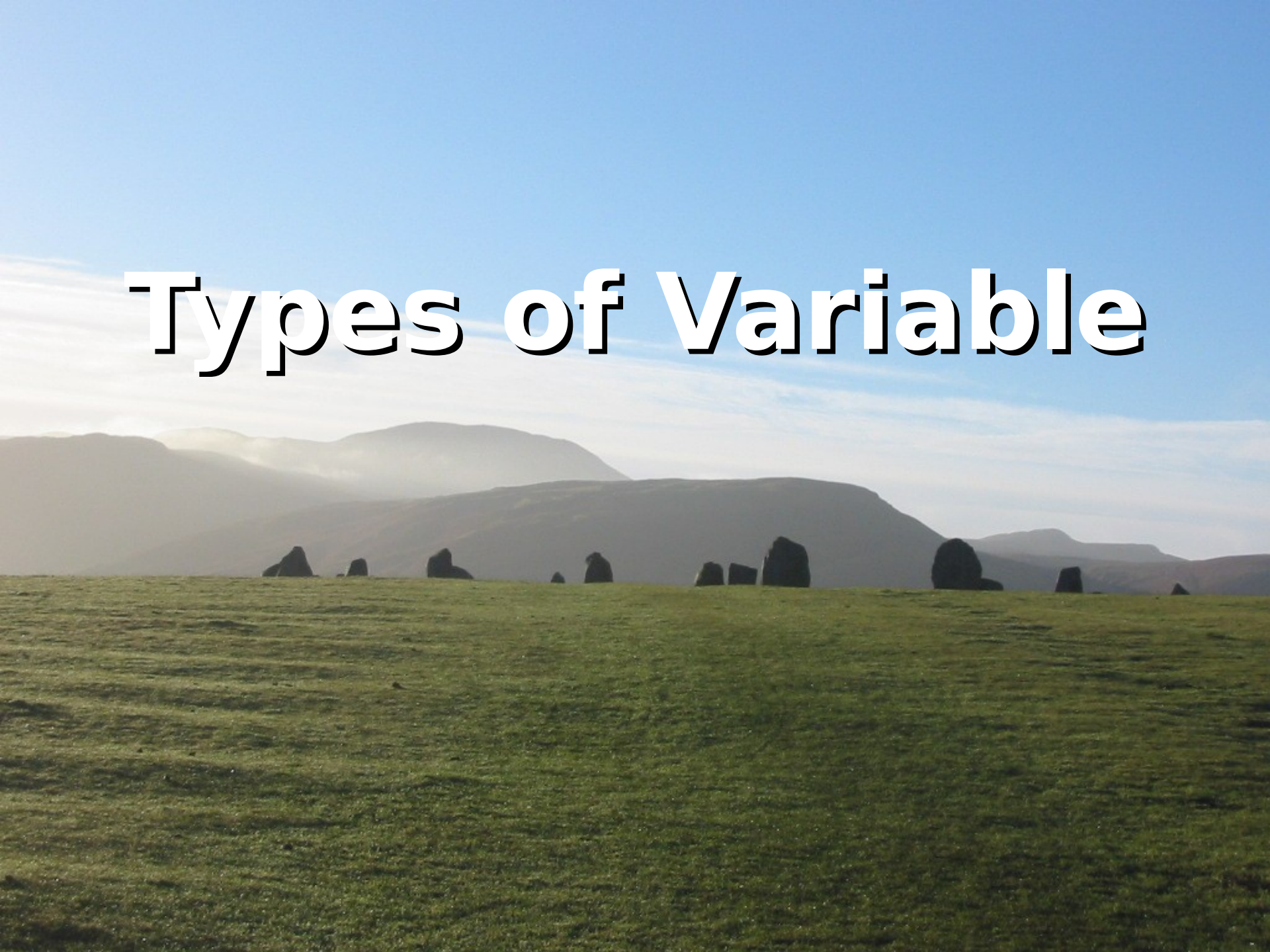
- Testing
- Dates and Times
- Templates
- Databases
- Further Information

Schedule

- 09:45 – Begin
- 11:15 – Coffee break (15 mins)
- 13:00 – Lunch (60 mins)
- 14:00 – Begin
- 15:30 – Coffee break (15 mins)
- 17:00 – End

Resources

- Slides available on-line
  - http://mag-sol.com/train/public/2010-04/ukuug
- Also see Slideshare
  - http://www.slideshare.net/davorg/slideshows
- Get Satisfaction
  - http://getsatisfaction.com/magnum


Types of Variable

Types of Variable

- Perl variables are of two types
- Important to know the difference
- Lexical variables are created with `my`
- Package variables are created by `our`
- Lexical variables are associated with a code block
- Package variables are associated with a package

Lexical Variables

- Created with `my`
- `my ($doctor, @timelords, %home_planets);`
- Live in a pad (associated with a block of code)
  - Piece of code delimited by braces
  - Source file

Lexical Variables

- Only visible within enclosing block
- ```
  while (<$fh>) {
     my $data = munge($_);
  }
  # can't see $data here
  ```
- "Lexical" because the scope is defined purely by the text


  ```

Packages

- All Perl code is associated with a package
- A new package is created with package
  - `package MyPackage;`
- Think of it as a namespace
- Used to avoid name clashes with libraries
- Default package is called main

Package Variables

- Live in a package's symbol table
- Can be referred to using a fully qualified name
  - `$main::doctor`
  - `@Gallifrey::timelords`

Package Variables

- Package name not required within own package
- ```
  package Gallifrey;
  @timelords = ('Doctor', 'Master',
                'Rani');
  ```
- Can be seen from anywhere in the package (or anywhere at all when fully qualified)


  ```

Declaring Package Vars

- Can be predeclared with `our`
- `our ($doctor, @timelords, %home_planet);`
- Or (in older Perls) with `use vars`
- `use vars qw($doctor @timelords %home_planet);`

Lexical or Package

- When to use lexical variables or package variables?
- Simple answer
  - Always use lexical variables
- More complete answer
  - Always use lexical variables
  - Except for a tiny number of cases
- http://perl.plover.com/local.html

local

- You might see code that uses local
- `local $variable;`
- This doesn't do what you think it does
- Badly named function
- Doesn't create local variables
- Creates a local copy of a package variable
- Can be useful
  - In a small number of cases

local Example

- `$/` is a package variable
- It defines the input record separator
- You might want to change it
- Always localise changes
- ```
  {
    local $/ = “\n\n”;
    while (<FILE> ) {
      ...
    }
  }
  ```



  ```


Strict and
Warnings

Coding Safety Net

- Perl can be a very loose programming language
- Two features can minimise the dangers
- `use strict` / `use warnings`
- A good habit to get into
- No serious Perl programmer codes without them

use strict

- Controls three things
- `use strict 'refs'` – no symbolic references
- `use strict 'subs'` – no barewords
- `use strict 'vars'` – no undeclared variables
- `use strict` – turn on all three at once
- turn them off (carefully) with `no strict`

use strict 'refs'

- Prevents symbolic references
- Using a variable as another variable's name
- ```
  $what = 'dalek';
  $$what = 'Karn';
  # sets $dalek to 'Karn'
  ```
- What if 'dalek' came from user input?
- People often think this is a cool feature
- It isn't


  ```

use strict 'refs' (cont)

- Better to use a hash
- ```
  $what = 'dalek';
  $alien{$what} = 'Karn';
  ```
- Self contained namespace
- Less chance of clashes
- More information (e.g. all keys)


  ```

use strict 'subs'

- No barewords
- Bareword is a word with no other interpretation
- e.g. word without $, @, %, &
- Treated as a function call or a quoted string
- `$dalek = Karn;`
- May clash with future reserved words

use strict 'vars'

- Forces predeclaration of variable names
- Prevents typos
- Less like BASIC - more like Ada
- Thinking about scope is good

use warnings

- Warns against dubious programming habits
- Some typical warnings
  - Variables used only once
  - Using undefined variables
  - Writing to read-only file handles
  - And many more...

Allowing Warnings

- Sometimes it's too much work to make code warnings clean
- Turn off use warnings locally
- Turn off specific warnings
- ```
  {
    no warnings 'deprecated';
    # dodgy code ...
  }
  ```
- See perldoc perllexwarn


  ```

References

Introducing References

- A reference is a bit like pointer in languages like C and Pascal (but better)
- A reference is a unique way to refer to a variable.
- A reference can always fit into a scalar variable
- A reference looks like `SCALAR(0x20026730)`

Creating References

- Put \ in front of a variable name
  - `$scalar_ref = \$scalar;`
  - `$array_ref = \@array;`
  - `$hash_ref = \%hash;`
- Can now treat it just like any other scalar
  - `$var = $scalar_ref;`
  - `$refs[0] = $array_ref;`
  - `$another_ref = $refs[0];`

Creating References

- `[ LIST ]` creates anonymous array and returns a reference
-
```
$aref = [ 'this', 'is', 'a', 'list'];
$aref2 = [ @array ];
```
- `{ LIST }` creates anonymous hash and returns a reference
-
```
$href = { 1 => 'one', 2 => 'two' };
$href = { %hash };
```


Creating References

- ```
  @arr = (1, 2, 3, 4);
  $aref1 = \@arr;
  $aref2 = [ @arr ];
  print "$aref1\n$aref2\n";
  ```
- Output
  ```
  ARRAY(0x20026800)
  ARRAY(0x2002bc00)
  ```
- Second method creates a **copy** of the array


  ```

Using Array References

- Use `{$aref}` to get back an array that you have a reference to
- Whole array
- `@array = @{$aref};`
- `@rev = reverse @{$aref};`
- Single elements
- `$elem = ${$aref}[0];`
- `${$aref}[0] = 'foo';`

Using Hash References

- Use `{$href}` to get back a hash that you have a reference to
- Whole hash
- `%hash = %{$href};`
- `@keys = keys %{$href};`
- Single elements
- `$elem = ${$href}{key};`
- `${$href}{key} = 'foo';`

Using References

- Use arrow (->) to access elements of arrays or hashes
- Instead of `${$aref}[0]` you can use `$aref->[0]`
- Instead of `${$href}{key}` you can use `$href->{key}`

Using References

- You can find out what a reference is referring to using ref
- ```
  $aref = [ 1, 2, 3 ];
  print ref $aref; # prints ARRAY
  ```
- ```
  $href = { 1 => 'one',
            2 => 'two' };
  print ref $href; # prints HASH
  ```


Why Use References?

- Parameter passing
- Complex data structures

Parameter Passing

- What does this do?
- ```
  @arr1 = (1, 2, 3);
  @arr2 = (4, 5, 6);
  check_size(@arr1, @arr2);

  sub check_size {
     my (@a1, @a2) = @_;
     print @a1 == @a2 ?
              'Yes' : 'No';
  }
  ```



  ```

Why Doesn't It Work?

- `my (@a1, @a2) = @_;`
- Arrays are combined in `@_`
- All elements end up in `@a1`
- How do we fix it?
- Pass references to the arrays

Another Attempt

```
@arr1 = (1, 2, 3);
@arr2 = (4, 5, 6);
check_size(\@arr1, \@arr2);

sub check_size {
   my ($a1, $a2) = @_;
   print @$a1 == @$a2 ?
            'Yes' : 'No';
}
```


Complex Data Structures

- Another good use for references
- Try to create a 2-D array
- ```
  @arr_2d = ((1, 2, 3),
             (4, 5, 6),
             (7, 8, 9));
  ```
- @arr_2d contains
  `(1, 2, 3, 4, 5, 6, 7, 8, 9)`
- This is known a *array flattening*


  ```

Complex Data Structures

- 2D Array using references
- `@arr_2d = ([1, 2, 3],`
  `[4, 5, 6],`
  `[7, 8, 9]);`
- But how do you access individual elements?
- `$arr_2d[1]` is ref to array (4, 5, 6)
- `$arr_2d[1]->[1]` is element 5

Complex Data Structures

- Another 2D Array
- `$arr_2d = [[1, 2, 3],`
  `[4, 5, 6],`
  `[7, 8, 9]];`
- `$arr_2d->[1]` is ref to array (4, 5, 6)
- `$arr_2d->[1]->[1]` is element 5
- Can omit intermediate arrows
- `$arr_2d->[1][1]`

More Data Structures

- Imagine the following data file
- ```
  Jones,Martha,UNIT
  Harkness,Jack,Torchwood
  Smith,Sarah Jane,Journalist
  ```
- What would be a good data structure?
- Hash for each record
- Array of records
- Array of hashes


  ```

More Data Structures

- Building an array of hashes
- ```
  my @records;
  my @cols =
   ('s_name', 'f_name', 'job');

  while (<FILE>) {
    chomp;
    my %rec;
    @rec{@cols} = split /,/;
    push @records, \%rec;
  }
  ```



  ```

Using an Array of Hashes

```
foreach (@records) {
   print "$_->{f_name} ",
         "$_->{s_name} ".
         "is a $_->{job}\n";
}
```


Complex Data Structures

- Many more possibilities
  - Hash of hashes
  - Hash of lists
  - Multiple levels (list of hash of hash, etc.)
- Lots of examples in “perldoc perldsc” (the data structures cookbook)


Sorting

Sorting

- Perl has a sort function that takes a list and sorts it
- `@sorted = sort @array;`
- Note that it does not sort the list in place
- `@array = sort @array;`

Sort Order

- The default sort order is ASCII
- ```
  @chars = sort 'e', 'b', 'a', 'd', 'c';
  # @chars has ('a', 'b', 'c', 'd', 'e')
  ```
- This can sometimes give strange results
- ```
  @chars = sort 'E', 'b', 'a', 'D', 'c';
  # @chars has ('D', 'E', 'a', 'b', 'c')
  ```
- ```
  @nums = sort 1 .. 10;
  # @nums has (1, 10, 2, 3, 4,
  #            5,  6, 7, 8, 9)
  ```



  ```

Sorting Blocks

- Can add a "sorting block" to customise sort order
- `@nums =`

```
    sort { $a <=> $b } 1 .. 10;
```

- Perl puts two of the values from the list into `$a` and `$b`
- Block compares values and returns -1, 0 or 1
- `<=>` does this for numbers (`cmp` for strings)

Sort Examples

- Other simple sort examples
- `sort { $b cmp $a } @words`
- `sort { lc $a cmp lc $b } @words`
-
```
sort { substr($a, 4)
       cmp substr($b, 4) } @lines
```


Sorting Subroutines

- Can also use a subroutine name in place of a code block
- `@words = sort dictionary @words;`

```
sub dictionary {
   # Don't change $a and $b
   my ($A, $B) = ($a, $b);
   $A =~ s/\W+//g;
   $B =~ s/\W+//g;
   $A cmp $B;
}
```


Sorting Names

- ```
  my @names = ('Rose Tyler',
               'Martha Jones',
               'Donna Noble',
               'Amy Pond');

  @names = sort sort_names @names;
  ```
- Need to write sort_names so that it sorts on surname and then forename.


  ```

Sorting Names (cont)

```
sub sort_names {
   my @a = split /\s/, $a;
   my @b = split /\s/, $b;

   return $a[1] cmp $b[1]
     or $a[0] cmp $b[1];
}
```


More Complex Sorts

- ```
  sub sort_names {
     my @a = split /\s/, $a;
     my @b = split /\s/, $b;

     return $a[1] cmp $b[1]
       or $a[0] cmp $b[0];
  }
  ```
- Can be inefficient on large amounts of data
- Multiple splits on the same data


  ```

More Efficient Sorts

- Split each row only once
- `@split = map { [ split ] } @names;`
- Do the comparison
- `@sort = sort { $a->[1] cmp $b->[1]`
  `or $a->[0] cmp $b->[0] } @split;`
- Join the data together
- `@names = map { join ' ', @$_ }`
  `@sort;`

Put It All Together

- Can rewrite this as
- ```
  @names = map  { join ' ', @$_ }
     sort { $a->[1] cmp $b->[1]
       || $a->[0] cmp $b->[0] }
     map  { [ split ] } @names;
  ```
- All functions work on the output from the previous function in the chain


  ```

Schwartzian Transform

- ```
  @data_out =
     map  { $_->[1] }
     sort { $a->[0] cmp $a->[0] }
     map  { [func($_), $_] }
      @data_in;
  ```
- Old Lisp trick
- Named after Randal Schwartz


  ```

Reusable Code

Why Write Modules?

- Code reuse
- Prevent reinventing the wheel
- Easier to share across projects
- Better design, more generic

Basic Module

```
use strict;
use warnings;

package MyModule;

use Exporter;
our @ISA = ('Exporter');
our @EXPORT = ('my_sub');

sub my_sub {
  print "This is my_sub\n";
}

1;
```


Using MyModule.pm

- use MyModule;

```
# my_sub is now available
# for use within your
# program

my_sub();
# Prints "This is my_sub()"
```


Explaining MyModule.pm

- Much of MyModule.pm is concerned with exporting subroutine names
- Subroutine full name
  - `MyModule::my_sub()`
- Exporting abbreviates that
  - `my_sub()`

Packages Revisited

- Every subroutine lives in a package
- The default package is `main`
- New packages are introduced with the `package` keyword
- A subroutine's full name is `package::name`
- Package name can be omitted from within same package
- Like family names

Using Exporter

- The module Exporter.pm handles the export of subroutine (and variable) names
- Exporter.pm defines a subroutine called `import`
- `import` is automatically called whenever a module is used
- `import` puts references to our subroutines into our caller's symbol table

How Exporter Works

- How does MyModule use Exporter's `import` subroutine?
- We make use of inheritance
- Inheritance is defined using the `@ISA` array
- If we call a subroutine that doesn't exist in our module, then the modules in `@ISA` are also checked
- Therefore `Exporter::import` is called

Exporting Symbols

- How does import know which subroutines to export?
- Exports are defined in `@EXPORT` or `@EXPORT_OK`
- Automatic exports are defined in `@EXPORT`
- Optional exports are defined in `@EXPORT_OK`

Exporting Symbol Sets

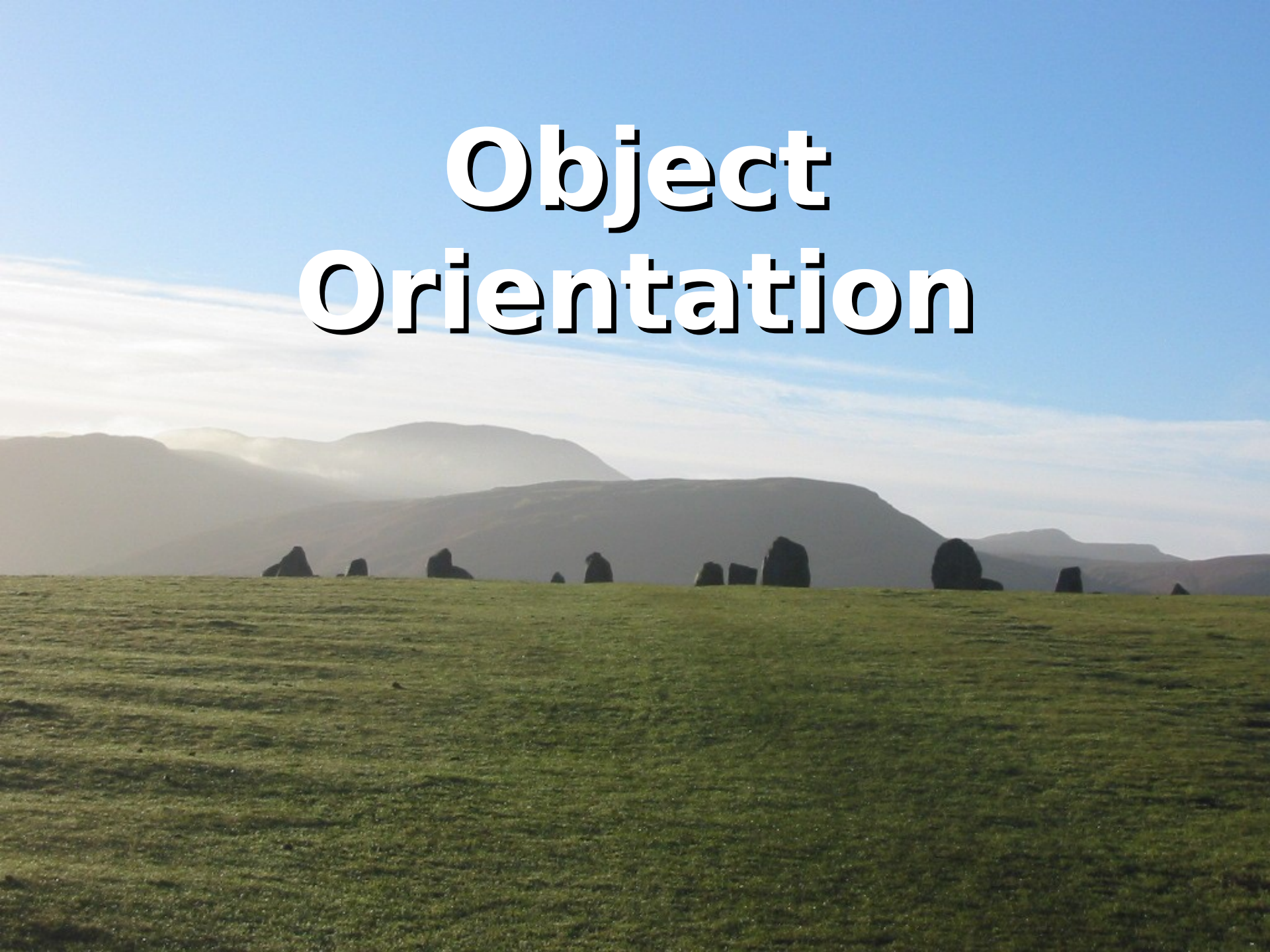
- You can define sets of exports in `%EXPORT_TAGS`
- Key is set name
- Value is reference to an array of names
- ```
  our %EXPORT_TAGS =
    (advanced => [ qw( my_sub
                       my_other_sub ) ];

  use MyModule qw(:advanced);
  my_sub();
  my_other_sub();
  ```



  ```

Why Use @EXPORT_OK?

- Give your users the choice of which subroutines to import
- Less chances of name clashes
- Use `@EXPORT_OK` in preference to `@EXPORT`
- Document the exported names and sets

Exporting Variables

- You can also export variables
- ```
  @EXPORT_OK = qw($scalar,
                  @array,
                  %hash);
  ```
- Can be part of export sets
- Any variables you export must be package variables


  ```

Writing Modules The Easy Way

- A lot of module code is similar
- Don't write the boilerplate yourself
- Copy from an existing module
- Or look at Module::Starter


Object
Orientation

OO Programming

- Traditional programming has subroutines acting on methods
- OO inverts this
- Classes contain methods which define their actions
- Objects are instances of classes
- Perl has an OO system bolted on
- Best of both worlds

Object Oriented Perl

- An Object is just a module that obeys certain extra rules
- Three rules of Perl objects
  - A Class is a package
  - An Object is a reference (usually to a hash)
  - A Method is a subroutine
- `bless` tells a reference what kind of object it is

A Simple Object

```
package MyObject;

sub new {
   my $class = shift;
   my $name = shift;

   my $self = { name => $name };

   return bless $self, $class;
}
```


A Simple Object (cont)

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

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

1;
```


Using MyObject.pm

- ```
  use MyObject;
  my $obj =
    MyObject->new('Dave');

  print $obj->get_name;
  # prints 'Dave'

  $obj->set_name('David');
  print $obj->get_name;
  # prints 'David'
  ```



  ```

Moose

- Easier OO Perl
- Moose is on CPAN
- Based on Perl 6 OO
- Well worth investigating

Moose Example

```
package MyModule;
use Moose;

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

1;
```


Further Information

- perldoc perlboot
- perldoc perltoot
- perldoc perlobj
- perldoc perlbot
- perldoc Moose (if it is installed)
- *Object Oriented Perl* (Conway)


Testing

Testing

- Never program without a safety net
- Does your code do what it is supposed to do?
- Will your code continue to do what it is supposed to do?
- Write unit tests
- Run those tests all the time

When to Run Tests

- As often as possible
- Before you add a feature
- After you have added a feature
- Before checking in code
- Before releasing code
- Constantly, automatically

Testing in Perl

- Perl makes it easy to write test suites
- A lot of work in this area over the last eight years
- Test::Simple and Test::More included in Perl distribution
- Many more testing modules on CPAN

Simple Test Program

- ```
  use Test::More tests => 4;

  BEGIN { use_ok('My::Object'); }

  ok(my $obj = My::Object->new);

  isa_ok($obj, 'My::Object');

  $obj->set_foo('Foo');

  is($obj->get_foo, 'Foo');
  ```



  ```

Simple Test Output

- ```
  $ prove -v test.t
  test....
  1..4
  ok 1 - use My::Object;
  ok 2
  ok 3 - The object isa My::Object
  ok 4
  ok
  All tests successful.
  Files=1, Tests=4,  0 wallclock secs
  ( 0.02 usr  0.00 sys +  0.05 cusr  0.00
  csys =  0.07 CPU)
  Result: PASS
  ```



  ```

Adding Test Names

- ```
  use Test::More tests => 4;
  BEGIN { use_ok('My::Object'); }

  ok(my $obj = My::Object->new,
     'Got an object');

  isa_ok($obj, 'My::Object');

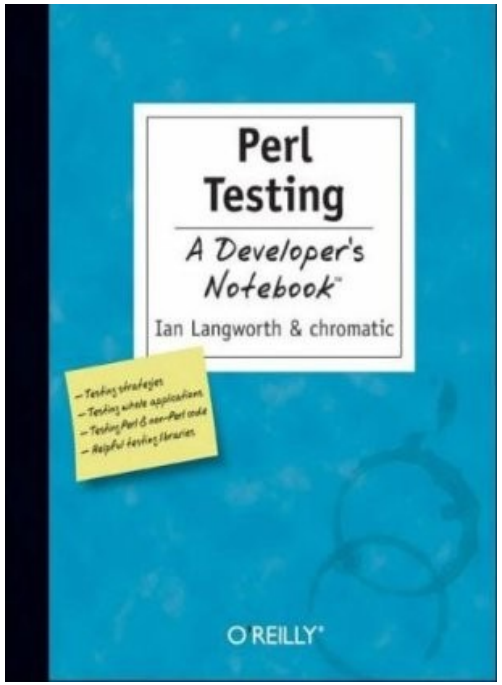
  $obj->set_foo('Foo');

  is($obj->get_foo, 'Foo',
     'The foo is "Foo"');
  ```



  ```

Output With Names

- ```
  $ prove -v test2.t
  test2....
  1..4
  ok 1 - use My::Object;
  ok 2 - got an object
  ok 3 - The object isa My::Object
  ok 4 - The foo is "Foo"
  ok
  All tests successful.
  Files=1, Tests=4,  0 wallclock secs
  ( 0.02 usr  0.00 sys +  0.05 cusr  0.00
  csys =  0.07 CPU)
  Result: PASS
  ```



  ```

Using `prove`

- A command line tool for running tests
- Runs given tests using Test::Harness
- Comes with the Perl distribution
- Command line options
  - -v verbose output
  - -r recurse
  - -s shuffle tests
  - Many more

Test Anything Protocol

- Perl tests have been spitting out “ok 1” and not “ok 2” for years
- Now this ad-hoc format has a definition and a name
- The Test Anything Protocol (TAP)
- See Test::Harness::TAP (documentation module) and TAP::Parser

TAP Output

- More possibilities for test output
  - TAP::Harness::Color
  - Test::TAP::HTMLMatrix
- Make sense of your test results

More Testing Modules

- Dozens of testing modules on CPAN
- Some of my favourites
- Test::File
- Test::Exception, Test::Warn
- Test::Differences
- Test::XML (includes Test::XML::XPath)

More Information

- Perl Testing: A Developer's Notebook (Ian Langworth & chromatic)
- perldoc Test::Tutorial
- perldoc Test::Simple
- perldoc Test::More
- etc...




Dates and Times

Dates & Times

- Perl has built-in functions to handle dates and times
- `time` – seconds since 1st Jan 1970
- `localtime` – convert to human-readable
- `timelocal` (in Time::Local) – inverse of localtime
- `strftime` (in POSIX) – formatting dates and times

Dates & Times on CPAN

- Look to CPAN for a better answer
- Dozens of date/time modules on CPAN
- Date::Manip is almost never what you want
- Date::Calc, Date::Parse, Class::Date, Date::Simple, etc
- Which one do you choose?

Perl DateTime Project

- http://datetime.perl.org/
- *"The DateTime family of modules present a unified way to handle dates and times in Perl"*
- "unified" is good
- Dozens of modules that work together in a consistent fashion

Using DateTime

- ```
  use DateTime;

  my $dt = DateTime->now;
  say $dt;
  # 2001-04-14T15:06:07
  say $dt->dmy, "\n";
  # 2010-04-14
  say $dt->hms, "\n";
  # 15:06:07
  ```



  ```

Using DateTime

- use DateTime;

```
my $dt = DateTime->new(year  => 2010,
                       month => 4,
                       day   => 14);

say $dt->ymd('/'), "\n";
# 2010/04/14
say $dt->month;       # 4
say $dt->month_name; # April
```


Arithmetic

- A DateTime object is a point in time
- For date arithmetic you need a duration
- Number of years, weeks, days, etc

Arithmetic

```
use DateTime;
my $dt = DateTime->new(year => 2010,
                       month => 4,
                       day => 14);

my $two_weeks =
DateTime::Duration->new(weeks => 2);
$dt += $two_weeks;
say $dt;
# 2010-04-28T00:00:00
```


Formatting Output

- ```
  use DateTime;
  my $dt = DateTime->new(year => 2010,
                         month => 4,
                         day => 14);
  say $dt->strftime('%A, %d %B %Y');
  # Wednesday, 14 April 2010
  ```
- Control input format with DateTime::Format::Strptime


  ```

Parsing & Formatting

- Ready made parsers and formatters for popular date and time formats
- DateTime::Format::HTTP
- DateTime::Format::MySQL
- DateTime::Format::Excel
- DateTime::Format::Baby
  - the big hand is on...

Alternative Calendars

- Handling non-standard calendars
- DateTime::Calendar::Julian
- DateTime::Calendar::Hebrew
- DateTime::Calendar::Mayan
- DateTime::Fiction::JRRTolkien::Shire

Calendar Examples

- ```
  use DateTime::Calendar::Mayan;

  my $dt = DateTime::Calendar::Mayan->now;

  say $dt->date; # 12.19.17.4.9
  ```
- ```
  use DateTime::Fiction::JRRTolkien::Shire;

  my $dt =
    DateTime::Fiction::JRRTolkien::Shire->now;

  say $dt->on_date; # Mersday 13 Astron 7474
  ```



Templates

Templates

- Many people use templates to produce web pages
- Advantages are well known
- Standard look and feel (static/dynamic)
- Reusable components
- Separation of code logic from display logic
- Different skill-sets (HTML vs Perl)

Non-Web Templates

- The same advantages apply to non-web areas
- Reports
- Business documents
- Configuration files
- Anywhere you produce output

DIY Templating

- Must be easy - so many people do it
- See perlfaq4
- “How can I expand variables in text strings?”

DIY Templating

- ```
  $text =
  'this has a $foo in it and a $bar';

  %user_defs = (
    foo  => 23,
    bar  => 19,
  );

  $text =~ s/\$(\w+)/$user_defs{$1}/g;
  ```
- Don't do that


  ```

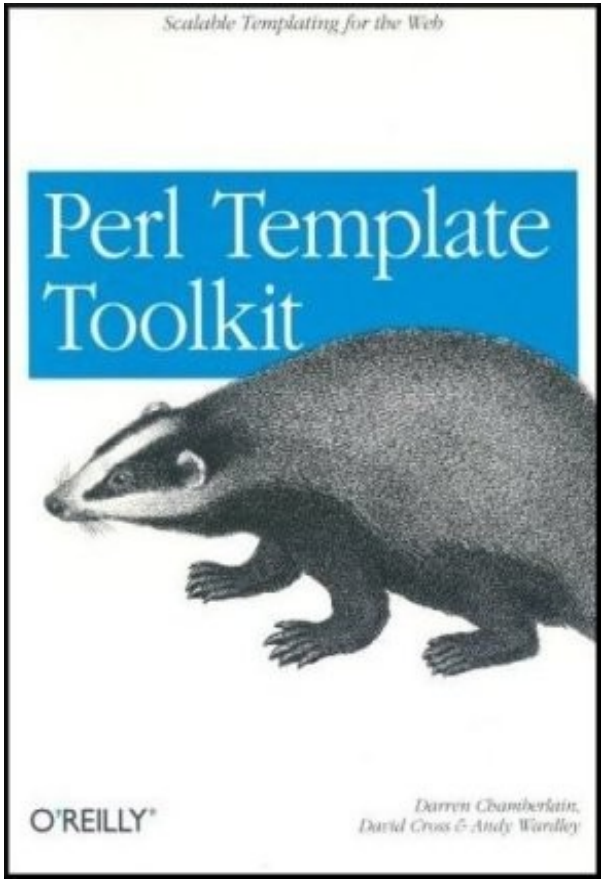
Templating Options

- Dozens of template modules on CPAN
- Text::Template, HTML::Template, Mason, Template Toolkit
- Many, many more
- Questions to consider
  - HTML only?
  - Template language
- I recommend the Template Toolkit

Template Toolkit

- http://tt2.org/
- Very powerful
- Both web and non-web
- Simple template language
- Plugins give access to much of CPAN
- Can use Perl code if you want
  - But don't do that

Good Book Too!



The Template Equation

- Data + Template = Output
- Data + Alternative Template = Alternative Output
- Different views of the same data
- Only the template changes

Simple TT Example

```
use Template;
use My::Object;
my ($id, $format) = @ARGV;
$format ||= 'html';
my $obj = My::Object->new($id)
  or die;
my $tt  = Template->new;
$tt->process("$format.tt",
             { obj => $obj },
             "$id.$format")
  or die $tt->error;
```


html.tt

```
<html>
  <head>
    <title>[% obj.name %]</title>
  </head>
  <body>
    <h1>[% obj.name %]<h1>
    <p><img src="[% obj.img %]" /><br />
    [% obj.desc %]</p>
    <ul>
    [% FOREACH child IN obj.children -%]
      <li>[% child.name %]</li>
    [% END %]
  </body>
</html>
```


text.tt

- ```
  [% obj.name | upper %]

  Image: [% obj.img %]
  [% obj.desc %]

  [% FOREACH child IN obj.children -%]
    * [% child.name %]
  [% END %]
  ```



  ```

Adding New Formats

- No new code required
- Just add new output template
- Perl programmer need not be involved

Equation Revisited

- Data + Template = Output
  - Template Toolkit
- Template + Output = Data
  - Template::Extract
- Data + Output = Template
  - Template::Generate


Databases

Databases

- A large proportion of applications need to talk to databases
- Perl has tools to make this as simple as possible
- DBI is the basis for all modern Perl database access
- You should be using DBI
  - or something based on DBI

How DBI Works

- Program uses DBI.pm
- Create a connection to a particular type of database
- DBD module gets loaded
- DBD translates from DBI API to database specific calls
- DBD translates returned data into Perl data stuctures

Connecting to a DB

- ```
  use DBI;
  my $dbh = DBI->connect(
    "dbi:mysql:$some_stuff",
    $user, $pass
  );
  ```
- “mysql” is the name of the DBD
  - DBD::mysql
- Easy to port a program to another database
- Just change the connection line


  ```

Selecting Data

- Prepare the SQL statement
- ```
  my $sth = $dbh->prepare(
     'select name, genre from artist'
  );
  ```
- ```
  my $sth = $dbh->prepare(
     "select title,
      from   song
      where  artist = '$id'");
  ```
- Check return values (syntax errors)

Selecting Data (cont)

- Execute the statement
- `$sth->execute`
- Still need to check for errors

Selecting Data (cont)

- Fetch the returned data
- ```
  while (my @row =
              $sth->fetchrow_array){
    print "@row\n";
  }
  ```
- Fields are returned in the same order as they are defined in the query


  ```

Other Select Functions

- Other fetch methods are available:
  - `fetchrow_arrayref`
  - `fetchrow_hashref` (keys are column names)
  - `fetchall_arrayref`
  - `fetch` (alias for fetchrow_arrayref)
- Many more added each week

Some Caveats

- If you're using a fetch method that returns an array
  - Never use "select *"
  - For (hopefully) obvious reasons
- If you're using a fetch method that returns a hash
  - Ensure all your columns have (unique) names
  - For (hopefully) obvious reasons

Insert, Update & Delete

- Statements that don't return data can be executed the same way
- ```
  my $sql = "update table1
                set col1 = '$val'
                where id_col = $id";
  my $sth  = $dbh->prepare($sql);
  $sth->execute;
  ```
- But there's a shortcut
- ```
  $rows = $dbh->do($sql);
  ```


Multiple Insertions

- ```
  while (<FILE>) {
     chomp;
     my @data = split;
     my $sql = "insert into tab
                  values ($data[0],
                          $data[1],
                          $data[2]");
     $dbh->do($sql);
  }
  ```
- Recompiles the SQL every time
- Very inefficient


  ```

Binding Data

- Prepare statement once, use many times
- ```
  my $sql = "insert into tab
                values (?, ?, ?)";
  my $sth = $dbh->prepare($sql);
  while (<FILE>) {
     my @data = split;
     bind_param(1, $data[0]);
     bind_param(2, $data[1]);
     bind_param(3, $data[2]);
     $sth->execute;
  }
  ```
- Bonus - binding handles quoting for you


  ```

Binding Data (cont)

- Even easier – extra parameters to execute
- ```
  my $sql = "insert into tab
                values (?, ?, ?)";
  my $sth = $dbh->prepare($sql);

  while (<FILE>) {
     chomp;
     my @data = split;
     $sth->execute(@data);
  }
  ```



  ```

Unnamed Placeholders

- Having unnamed placeholders can get confusing
- ```
  my $sql = 'insert into big_table
             values(
  ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,
  ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?,
  ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)';
  ```
- Good chance of getting the variables in the wrong order
- By the way - there's a basic maintainability error in that SQL


  ```

Bind By Name

- ```
  my $sql = 'insert into big_table
             (id, code, name, addr, email,
              url, ... )
             values (:id, :code, :name,
                     :addr, :email, :url,
                     ... );

  my $sth = $sql->prepare($sql);

  $sth->bind_param(':id', $id);
  $sth->bind_param(':code', $code);
  # etc

  $sth->execute;
  ```



  ```

Even Easier Binding

- Store your data in a hash
-
```
my %data = (id   => 42,
            code => 'H2G2',
            ... );

# and later...
foreach my $col (keys %data) {
  $sth->bind_param(":$col",
                   $data{$col};
}
```


Downsides

- Many DBDs don't support it
- Which is a bit of a bugger
- Oracle does
- So does PostgreSQL (tho' the docs discourage its use)
- Check your DBD documentation
- Email your friendly neighbourhood DBD author

Some Tips

- Make your life as easy as possible
- Don't hard-code connection data
  - Config file, command line options, environment variables
- Send all of your data access through one function
- Store SQL queries externally and reference them by name
- Use named bind parameters if you can

Sample Code

```
my $dbh;

sub run_sql {
  my ($sql_statement, %args) = @_;
  my $sql = get_sql($sql_statement);
  $dbh = get_dbh() unless $dbh;

  my $sth = $dbh->prepare($sql);
  foreach my $col (keys %args) {
    $sth->bind_param(":$col",
                     $args{$col});
  }

  return $sth->execute;
}
```


Not Writing SQL

- Writing SQL is boring
- It's often similar
  - Select the id and name from this table
  - Select all the details of this row
  - Select something about related tables
  - Update this row with these values
  - Insert a new record with these values
  - Delete this record
- Must be a better way

Object Relational Mapping

- Mapping database relations into objects
- Tables (relations) map onto classes
- Rows (tuples) map onto objects
- Columns (attributes) map onto attributes
- Don't write SQL

Replacing SQL

- Instead of
- ```
  SELECT *
  FROM   my_table
  WHERE  my_id = 10
  ```
- and then dealing with the prepare/execute/fetch code


  ```

Replacing SQL

- We can write
- ```
  use My::Object;

  # warning! not a real orm
  my $obj = My::Object->retrieve(10)
  $obj->name('A New Name');
  $obj->save;
  ```
- Or something similar


  ```

ORM on CPAN

- Very many ORMs on CPAN
- Tangram
- Alzabo
- Class::DBI
- DBIx::Class
  - The current favourite
  - Highly recommended

Further Information

- perldoc DBI
- perldoc DBD::*
  - DBD::mysql
  - DBD::Oracle
  - Etc...
- perldoc DBIx::Class

That's All Folks

- Any Questions?