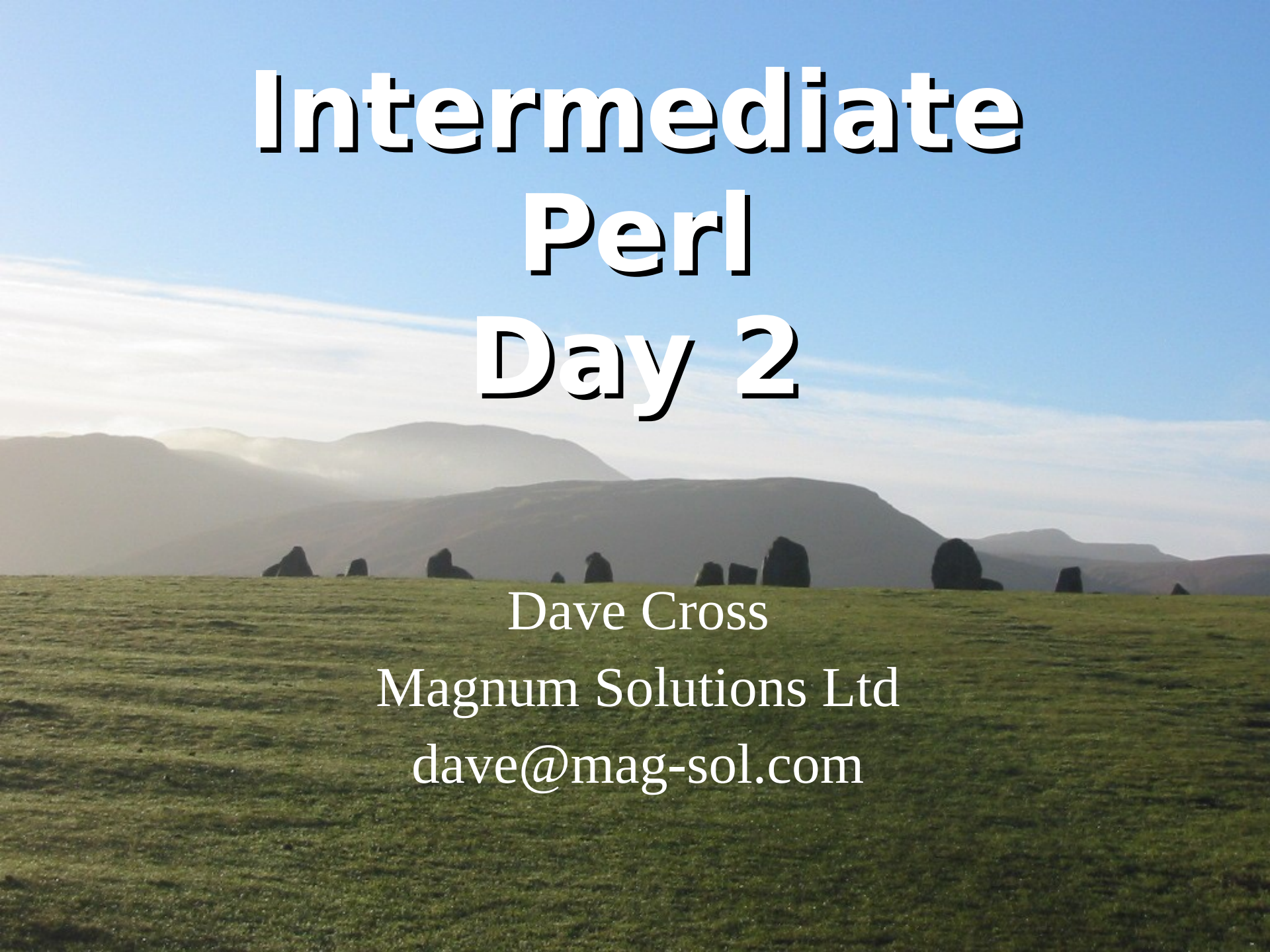


Intermediate Perl
Day 2

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

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

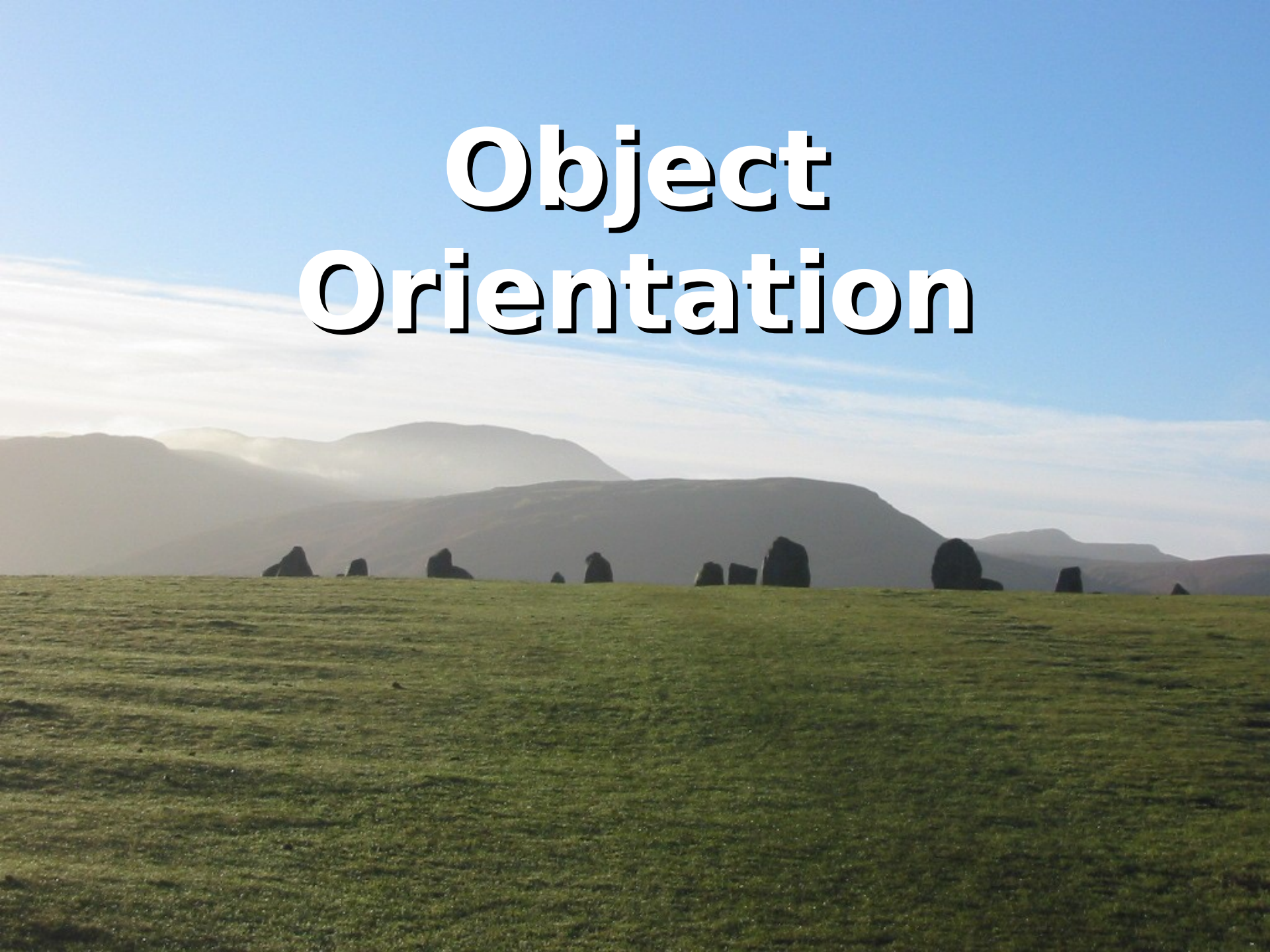
- Slides available on-line
  - http://mag-sol.com/train/public/2012-02/ukuug
- Also see Slideshare
  - http://www.slideshare.net/davorg/slideshows
- Get Satisfaction
  - http://getsatisfaction.com/magnum

What We Will Cover

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

What We Will Cover

- Object Orientation
- Testing
- Dates and Times
- Templates
- Databases


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 5 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)

OO Example

- Write a simple class with three or four methods
- Write a program which uses your class


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

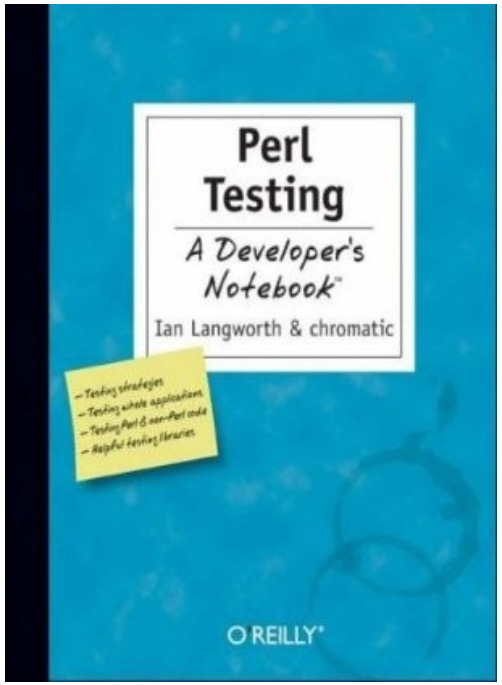
- More possibilities for test output
  - TAP::Harness::Color
  - TAP::Formatter::HTML
  - TAP::Formatter::JUnit
- Make sense of your test results

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...



{MAGNUM SOLUTIONS LIMITED}

Testing Examples

- Write a test script for the class that you wrote in the previous exercise


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;
  # 2012-02-22T12:06:07
  say $dt->ymd;
  # 2012-02-22
  say $dt->hms;
  # 12:06:07
  ```
  ```

Using DateTime

- ```
  use DateTime;

  my $dt = DateTime->new(year  => 2012,
                         month => 2,
                         day   => 22);

  say $dt->ymd('/');
  # 2012/02/22
  say $dt->month;      # 2
  say $dt->month_name; # February
  ```
  ```

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 => 2012,
                       month => 2,
                       day => 22);

my $two_weeks =
DateTime::Duration->new(weeks => 2);
$dt += $two_weeks;
say $dt;
# 2011-03-07T00:00:00
```

Formatting Output

- ```
  use DateTime;
  my $dt = DateTime->new(year => 2012,
                         month => 2,
                         day => 22);
  say $dt->strftime('%A, %d %B %Y');
  # Wednesday, 22 February 2011
  ```
  ```

Parsing Input

- ```
  use DateTime::Format::Strptime;
  my $p = DateTime::Format::Strptime->new(
    pattern => '%Y-%m-%d',
  );

  my $dt =
    $p->parse_datetime('2012-02-22');

  say $dt->day_name; # Wednesday
  ```
- DateTime::FormatStrptime is not part of the DateTime distribution
  ```

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; # 12.19.19.1.19
  ```
- ```
  use DateTime::Fiction::JRRTolkien::Shire;

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

  say $dt->on_date; # Sunday 13 Solmath 7476
  ```

Date/Time Examples

- Write a program that prints the current date
- Change it so that it accepts an output format as an optional argument
- Write a program which calculates someone's age in years, months, weeks and days


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
  ```

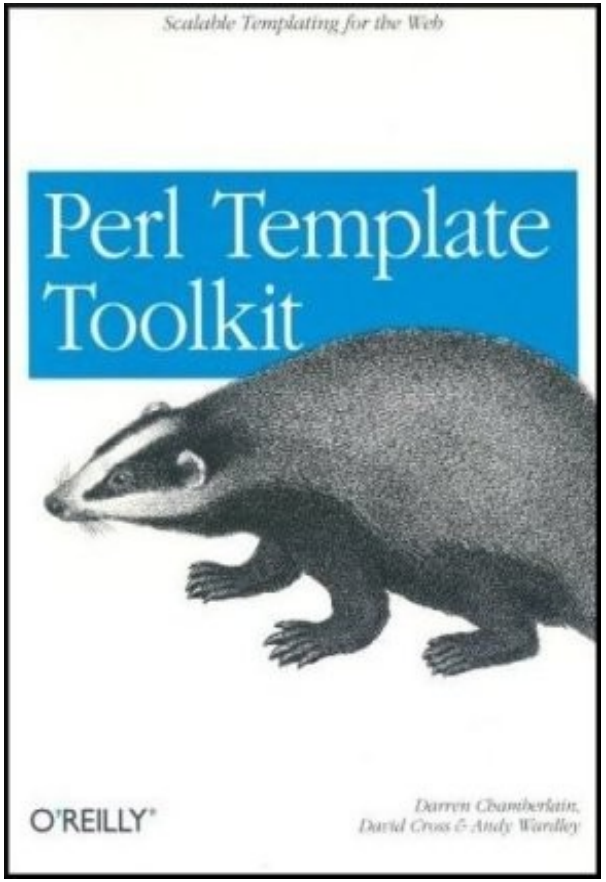
Templating Options

- 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!



{MAGNUM SOLUTIONS LIMITED}

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 -%]
    </ul>
  </body>
</html>
```

text.tt

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

  Image: [% obj.img %]
  [% obj.desc | wrap(50, '  ', '  ') %]

  [% 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

Template Examples

- Write a template which displays various attributes of the class you wrote earlier
- Write a program (like the one in this section) which uses your template
- Write another template that displays the data in a different format


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)

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

- 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
- Rose::DB
- Fey::ORM
- Class::DBI
- DBIx::Class
  - The current favourite
  - Highly recommended

Further Information

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

Database Examples

- Set up a local database with a few tables in it
- Write DBI code to insert data into the tables
- Write DBI code to produce a report of the data in the tables

That's All Folks

- Any Questions?