

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

- Modern Core Perl
 - What's new in Perl 5.10, 5.12 & 5.14
- Advanced Testing
- Database access with DBIx::Class
- Handling Exceptions

What We Will Cover

- Profiling and Benchmarking
- Object oriented programming with Moose
- MVC Frameworks
 - Catalyst
- PSGI and Plack

Modern Core Perl



Perl Releases

- Perl 5 has moved to a regular release cycle
- Major release every year
 - In Spring
- Minor releases when required

Perl Version Numbers

- Even major numbers are production releases
 - 5.10, 5.12, 5.14, 5.16
- Odd major numbers are dev releases
 - 5.9, 5.11, 5.13, 5.15

Perl Support

- p5p provide support for current and previous major releases
 - Currently 5.12 and 5.14
- Further support may be available from distributors

Recent Perl Releases

- 5.10.0 – 2007 Dec 18
- 5.10.1 – 2009 Aug 22
- 5.12.0 – 2010 Apr 12
- 5.12.1 – 2010 May 16
- 5.12.2 – 2010 Sep 6
- 5.12.3 – 2011 Jan 21

Recent Perl Releases

- 5.14.0 – 2011 May 14
- 5.14.1 – 2011 Jun 16
- 5.12.4 – 2011 Jun 20
- 5.14.2 – 2011 Sep 26

Perl 5.10

- Released 18th Dec 2007
 - Perl's 20th birthday
- Many new features
- Well worth upgrading

New Features

- Defined-or operator
- Switch operator
- Smart matching
- say()
- Lexical \$_

New Features

- State variables
- Stacked file tests
- Regex improvements
- Many more

Defined Or

- Boolean expressions “short-circuit”
- `$val = $val || $default;`
- `$val ||= $default;`
- What if 0 is a valid value?

Defined Or

- Need to check “definedness”
- `$val = defined $val`
 `? $val : $default;`
- `$val = $default`
 unless defined `$val;`

Defined Or

- The defined or operator makes this easier
- `$val = $val // $default;`
- A different slant on truth
- Checks definedness
- Shortcircuit version too
- `$val // = $value;`

Switch Statement

- Switch.pm was added with Perl 5.8
- Source filter
- Parser limitations
 - Regular expressions
 - eval
- 5.10 introduces a build-in switch statement

Given ... When

- Switch is spelled “given”
- Case is spelled “when”
- Powerful matching syntax

Given Example

- `given ($foo) {
 when (/^abc/) { $abc = 1; }
 when (/^def/) { $def = 1; }
 when (/^xyz/) { $xyz = 1; }
 default { $nothing = 1; }
}`

New Keywords

- Four new keywords
 - given
 - when
 - default
 - continue

given

- `given(EXPR)`
- Assigns the result of `EXPR` to `$_` within the following block
- Similar to `do { my $_ = EXPR; ... }`

when

- `when (EXPR)`
- Uses smart matching to compare `$_` with `EXPR`
- Equivalent to `when ($_ ~~ EXPR)`
- `~~` is the new smart match operator
- Compares two values and “does the right thing”

default

- default defines a block that is executed if no other blocks match
- default block is optional

continue

- continue keyword falls through to the next when block
- Normal behaviour is to break out of given block once the first when condition is matched
- Inverse of most other programming languages

continue

- ```
given($foo) {
 when (/x/) { say '$foo contains an x';
 continue }
 when (/y/) { say '$foo contains a y' }
 default { say '$foo contains no x or y' }
}
```

# Smart Matching

- `~~` is the new Smart Match operator
- Different kinds of matches
- Dependent on the types of the operands
- See “perldoc perlsyn” for the full details
- Warning: Still under discussion

# Smart Match Examples

- `$foo == $bar; # == or cmp`
- `$foo =~ @bar; # array contains value`
- `$foo =~ %bar; # hash key exists`
- `$foo =~ qr{$bar}; # regex match`
- `@foo == @bar; # arrays are identical`
- `%foo =~ %bar; # hash keys match`
- Many more alternatives

# say()

- say() is a new alternative to print()
- Adds a new line at the end of each call
- `say($foo); # print $foo, "\n";`
- Two characters shorter than print
- Less typing

# Lexical \$\_

- \$\_ is a package variable
- Always exists in main package
- Can lead to subtle bugs when not localised correctly
- Can now use my \$\_ to create a lexically scoped variable called \$\_

# State Variables

- Lexical variables disappear when their scope is destroyed

- ```
sub variables {  
    my $x;  
  
    say ++$x;  
}
```

```
variables() for 1 .. 3;
```

State Variables

- State variables retain their value when their scope is destroyed

- ```
sub variables {
 state $x;

 say ++$x;
}
```

```
variables() for 1 .. 3;
```

# State Variables

- Like static variables in C
- Deprecating bugs
  - `my $x if 0;`



# Stacked File Tests

- People often think you can do this
- `-f -w -x $file`
- Previously you couldn't
- Now you can
- Equivalent to
- `-x $file && -w _ && -f _`

# Regex Improvements

- Plenty of regular expression improvements
- Named capture buffers
- Possessive quantifiers
- Relative backreferences
- New escape sequences
- Many more

# Named Capture Buffers

- Variables \$1, \$2, etc change if the regex is altered
- Named captures retain their names
- ( ?<name> . . . ) to define
- Use new %+ hash to access them

# Named Capture Example

- ```
while (<DATA>) {  
    if (/(?<header>[\w\s]+)  
        :\s+(?<value>.+)/x) {  
        print "${header} -> ";  
        print "${value}\n";  
    }  
}
```

Possessive Quantifiers

- `?+`, `*+`, `++`
- Grab as much as they can
- Never give it back
- Finer control over backtracking
- `'aaaa' =~ /a++a/`
 - Never matches
- Rick Astley quantifier

Relative Backreferences

- $\backslash g\{N\}$
- More powerful version of $\backslash 1$, $\backslash 2$, etc
- $\backslash g\{1\}$ is the same as $\backslash 1$
- $\backslash g\{-1\}$ is the last capture buffer
- $\backslash g\{-2\}$ is the one before that

New Escape Sequences

- `\h` – Horizontal white space
- `\v` – Vertical white space
- Also `\H` and `\V`

Accessing New Features

- Some new features would break backwards compatibility
- They are therefore turned off by default
- Various ways to turn them on

Feature Pragma

- Turn new features on with the feature pragma
- use feature 'say';
- use feature 'switch';
- use feature 'state';
- use feature ':5.10';

Implicit Loading

- Two ways to automatically turn on 5.10 features
- Require a high enough version of Perl
- `use 5.10.0; # Or higher`
- `-E` command line option
- `perl -e 'say "hello"'`
- `perl -E 'say "hello"'`

Perl 5.12

- Released 12 April 2010
 - 5.12.4 20 June 2011
- Many new enhancements

5.12 Enhancements

- package NAME VERSION syntax
- ... operator
- Implicit strictures
- Y2038 compliance

5.12 Enhancements

- Smart match changes
- New modules
 - autodie
 - parent

package NAME VER

- Declare the version of a package in the package declaration
- `package My::Package 1.23;`
- Equivalent to
- `package My::Package;
our $VERSION = 1.23;`

... Operator

- Called the “yada-yada” operator
- Used to stand in for unwritten code
- ```
sub unimplemented {
 ...
}
```
- Code compiles
- Throws an “unimplemented” exception when run

# Implicit Strictures

- Requiring a version of Perl greater than 5.11 implicitly turns on use strict
- `use 5.12.0;`
- Equivalent to
- `use strict;`  
`use feature ':5.12';`



# Y2038 Compliance

- Core time functions are now Y2038 compliant

# Smart Match Changes

- Some changes to Smart Match operator
- No longer commutative
- See new table in perlsyn
- Still in flux!

# New Modules

- Some new modules in the standard distribution
- autodie
- parent
  - Better version of base.

# Perl 5.14

- Released 14 May 2011
  - 5.14.2 26 Sept 2011
- Many new enhancements

# 5.14 Enhancements

- Non-destructive substitution
- Container functions accept references
- Package block
- New modules

# Non-destructive substitution

- New /r option on s/// and tr///
- Copies input
- Acts on copy
- Original unmodified
- `$_ = 'cat';`  
`$new = s/cat/dog/r'; # $_ remains 'cat'`

# Container functions accept references

- Array & hash functions used to require arrays or hashes
  - push @array, \$value
  - @keys = keys %hash
- Even if you have a reference
  - push @\$arrayref, \$value
  - @keys = keys %\$hashref

# Container functions accept references

- Array & hash functions now accept references
  - `push $array_ref, $value`
  - `@keys = keys $hash_ref`
- Currently experimental



# Package block

- Attach a code block to a package declaration
- `package MyPackage { ... }`
- Equivalent to
- `{ package MyPackage; ... }`
- Can also declare a version
- `package MyPackage 1.23 { ... }`

# New Modules

- Many modules for parsing META files
- CPAN::Meta::YAML & JSON::PP
- CPAN::Meta
- CPAN::Meta::Spec & CPAN::Meta::History
- Module::Metadata

# New Modules

- Other new modules
- HTTP::Tiny
- Perl::OSType
- Version::Requirements

# Perl 5.16

- Due in spring 2012
- Currently in development at 5.15
  - 5.15.7 – 2012 Jan 20
  - Code freeze – 2011 Dec 20

# Perl 5.16

- Look for changes in perldelta
  - perldelta5150
  - perldelta5151
  - etc...

# Some Highlights

- CORE on all keywords
- Continue outside switch
- Breakpoints with filenames
- Remove Perl 4 \*.pl

# More Information

- perldoc perl5100delta
- perldoc perl5120delta
- perldoc perl5140delta

# Core Perl Examples

- Write a program that uses at least three or four of the new features



# Advanced Testing



# Writing Test Modules

- Standard test modules all work together
- Built using `Test::Builder`
- Ensures that test modules all use the same framework
- Use it as the basis of your own `Test::*` modules
- Test your `Test::Builder` test modules with `Test::Builder::Tester`

# Test::Between

- `package Test::Between;`

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

```
use base 'Exporter';
our @EXPORT = qw(is_between);
```

```
use Test::Builder;
```

```
my $test = Test::Builder->new;
```

# Test::Between

- ```
sub is_between {  
    my ($item, $lower, $upper, $desc)  
        = @_;  
  
    return (  
$test->ok($lower le $item &&  
        $item le $upper, $desc)  
    || $test->diag("$item is not between  
$lower and $upper")  
);  
}
```
- 1;

Using Test::Between

- ```
#!/usr/bin/perl
use strict;
use warnings;
use Test::More tests => 3;
use Test::Between;

is_between('b', 'a', 'c', 'alpha');
is_between(2, 1, 3, 'numeric');
is_between('two', 1, 3, 'wrong');
```

# Test::Between Output

- ```
$ prove -v test.pl
test.pl ..
1..3
ok 1 - alpha
ok 2 - numeric
not ok 3 - wrong

# Failed test 'wrong'
# at test.pl line 11.
# two is not between 1 and 3
# Looks like you failed 1 test of 3.
Dubious, test returned 1 (wstat 256,
0x100)
Failed 1/3 subtests
```

Test::Between Output

- Test Summary Report

```
-----  
test.pl (Wstat: 256 Tests: 3 Failed: 1)  
  Failed test: 3  
  Non-zero exit status: 1  
Files=1, Tests=3,  1 wallclock secs  
( 0.07 usr  0.01 sys + 0.05 cusr  0.01  
csys =  0.14 CPU)  
Result: FAIL
```

Mocking Objects

- Sometimes it's hard to test external interfaces
- Fake them
- Test::MockObject pretends to be other objects
- Gives you complete control over what they return

Testing Reactors

- You're writing code that monitors a nuclear reactor
- It's important that your code reacts correctly when the reactor overheats
- You don't have a reactor in the test environment

Testing Reactors

- Even if you did, you wouldn't want to make it overheat every time you run the tests
- Especially if you're not 100% sure of your code
- Of if you're running unattended smoke tests
- Fake it with a mock object

My::Monitor Spec

- If the temperature of a reactor is over 100 then try to cool it down
- If you have tried cooling a reactor down 5 times and the temperature is still over 100 then return an error

My::Monitor Code

- `package My::Monitor;`

```
sub new {  
    my $class = shift;  
    my $self = { tries => 0 };  
  
    return bless $self, $class;  
}
```

My::Monitor Code

- ```
sub check {
 my $self = shift;
 my $reactor = shift;

 my $temp = $reactor->temperature;

 if ($temp > 100) {
 $reactor->cooldown;
 ++$self->{tries};
 if ($self->{tries} > 5) {
 return;
 }
 }
 return 1;
}
```

# My::Monitor Code

- ```
    } else {  
      $self->{tries} = 0;  
      return 1;  
    }  
  }  
  
  1;
```

Mock Reactor

- Create a mock reactor object that acts exactly how we want it to
- Reactor object has two interesting methods
- temperature - returns the current temperature
- cooldown - cools reactor and returns success or failure

monitor.t

- `use Test::More tests => 10;`
`use Test::MockObject;`

`# Standard tests`

`BEGIN { use_ok('My::Monitor'); }`

`ok(my $mon = My::Monitor->new);`
`isa_ok($mon, 'My::Monitor');`

monitor.t

- # Create Mock Reactor Object

```
my $t = 10;  
my $reactor = Test::MockObject;  
  
$reactor->set_bound('temperature',  
                   \ $t);  
  
$reactor->set_true('cooldown');
```

monitor.t

- # Test reactor

```
ok($mon->check($reactor));
```

```
$t = 120;
```

```
ok($mon->check($reactor)) for 1 .. 5;
```

```
ok(!$mon->check($reactor));
```

How Good Are Your Tests?

- How much of your code is exercised by your tests?
- Devel::Cover can help you to find out
- Deep internal magic
- Draws pretty charts
 - HARNESS_PERL_SWITCHES=
-MDevel::Cover make test
 - cover

Devel::Cover Output

Coverage Summary - Mozilla Firefox

File Edit View Go Bookmarks Tools Help deljcio.us

http://dave.org.uk/code/Calendar-Simple/cover/ Go

Coverage Summary

Database: /home/dave/src/Calendar-Simple/cover_db

file	stmt	bran	cond	sub	pod	time	total
blib/lib/Calendar/Simple.pm	100.0	96.7	75.0	100.0	100.0	100.0	95.2
Total	100.0	96.7	75.0	100.0	100.0	100.0	95.2

Done \$0.00

Devel::Cover Output

Condition Coverage: blib/lib/Calendar/Simple.pm - Mozilla Firefox

File: blib/lib/Calendar/Simple.pm

Coverage: 75.0%

line	% coverage	A	B	dec	condition
78	100	A	B	dec	\$year < 1970 and not \$dt
		0	X	0	
		1	0	0	
		1	1	1	
79	100	A	B	dec	\$mon < 1 or \$mon > 12
		0	0	0	
		0	1	1	
		1	X	1	
80	100	A	B	dec	\$start_day < 0 or \$start_day > 6
		0	0	0	
		0	1	1	

Devel::Cover Output

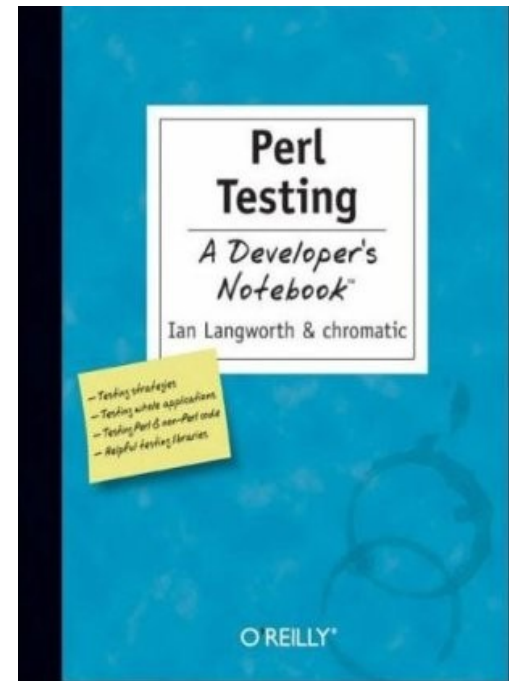
Line	Coverage	A	B	dec	Code Snippet
80	100	0	0	0	
		0	1	1	
		1	X	1	
162	33	0	0	0	<code>\$params{'mon'} \$now[0] + 1</code>
		0	1	1	
		1	X	1	
163	33	0	0	0	<code>\$params{'year'} \$now[1] + 1900</code>
		0	1	1	
		1	X	1	
164	100	0	0		<code>\$params{'begin'} 1</code>
		1	1		
165	67	0	0	0	<code>\$params{'end'} _days(\$mon, \$year)</code>
		0	1	1	
		1	X	1	

Alternative Test Paradigms

- Not everyone likes the Perl testing framework
- Other frameworks are available
- Test::Class
 - xUnit style framework
- Test::FIT
 - Framework for Interactive Testing
 - <http://fit.c2.com>

More Information

- Perl Testing: A Developer's Notebook (Ian Langworth & chromatic)
- perldoc Test::MockObject
- perldoc Test::Builder
- Devel::Cover
- etc...



Testing Examples

- Write a test plan for the supplied Perl module
- Do you need to mock any of the interfaces?
- How good is the coverage of your tests?

Object Relational Mapping



ORM

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

SQL Is Tedious

- 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

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)
```

- Or something similar

Writing An ORM Layer

- Not actually that hard to do yourself
- Each class needs an associated table
- Each class needs a list of columns
- Create simple SQL for basic CRUD operations
- Don't do that

Perl ORM Options

- Plenty of choices on CPAN
- Fey::ORM
- Rose::DB
- Class::DBI
- DBIx::Class
 - The current favourite

DBIx::Class

- Standing on the shoulders of giants
- Learning from problems in Class::DBI
- More flexible
- More powerful

DBIx::Class Example

- Modeling a CD collection
- Three tables
- artist (artistid, name)
- cd (cdid, artist, title)
- track (trackid, cd, title)

Main Schema

- Define main schema class
- DB/Main.pm
- ```
package DB::Main;
use base qw/DBIx::Class::Schema/;

__PACKAGE__->load_classes();

1;
```

# Object Classes

- DB/Main/Artist.pm
- ```
package DB::Main::Artist;
use base qw/DBIx::Class/;
__PACKAGE__->load_components(qw/PK::Auto
Core/);
__PACKAGE__->table('artist');
__PACKAGE__->add_columns(qw/ artistid name
/);
__PACKAGE__->set_primary_key('artistid');
__PACKAGE__->has_many(cds =>
                        'DB::Main::Cd');

1;
```

Object Classes

- DB/Main/CD.pm
- ```
package DB::Main::CD;
use base qw/DBIx::Class/;
__PACKAGE__->load_components(qw/PK::Auto
Core/);
__PACKAGE__->table('cd');
__PACKAGE__->add_columns(qw/ cdid artist
title year /);
__PACKAGE__->set_primary_key('cdid');
__PACKAGE__->belongs_to(artist =>
'DB::Main::Artist');
1;
```

# Inserting Artists

- ```
my $schema =  
    DB::Main->connect($dbi_str);  
  
my @artists = ('The Beta Band',  
              'Beth Orton');  
  
my $art_rs = $schema->resultset('Artist');  
  
foreach (@artists) {  
    $art_rs->create({ name => $_ });  
}
```

Inserting CDs

- Hash of Artists and CDs
- ```
my %cds = ('The Three EPs' =>
 'The Beta Band',
 'Trailer Park' =>
 'Beth Orton');
```

# Inserting CDs

- Find each artist and insert CD

- ```
foreach (keys $cds) {  
    my ($artist) = $art_rs->search(  
        { name => $cds{$_} }  
    );
```

```
    $artist->add_to_cds({  
        title => $_,  
    });  
}
```


Retrieving Data

- Get CDs by artist
- ```
my ($artist) = $art_rs->search({
 name => 'Beth Orton',
});
```

```
foreach ($artist->cds) {
 say $_->title;
}
```

# Searching for Data

- Search conditions can be more complex

- Alternatives

- `$rs->search({year => 2006},  
                  {year => 2007});`

- Like

- `$rs->search({name =>  
                  { 'like', 'Dav%' }});`

# Searching for Data

- Combinations

```
- $rs->search({forename =>
 { 'like', 'Dav%' },
 surname => 'Cross' });
```

# Don't Repeat Yourself

- There's a problem with this approach
- Information is repeated
- Columns and relationships defined in the database schema
- Columns and relationships defined in class definitions

# Repeated Information

- ```
CREATE TABLE artist (  
    artistid INTEGER PRIMARY KEY,  
    name      TEXT NOT NULL  
);
```

Repeated Information

- ```
package DB::Main::Artist;
use base qw/DBIx::Class/;
__PACKAGE__->
 load_components(qw/PK::Auto Core/);
__PACKAGE__->table('artist');
__PACKAGE__->
 add_columns(qw/ artistid name /);
__PACKAGE__->
 set_primary_key('artistid');
__PACKAGE__->has_many('cds' =>
 'DB::Main::Cd');
```

# Database Metadata

- Some people don't put enough metadata in their databases
- Just tables and columns
- No relationships. No constraints
- You may as well make each column VARCHAR(255)

# Database Metadata

- Describe your data in your database
- It's what your database is for
- It's what your database does best



# No Metadata (Excuse 1)

- "This is the only application that will ever access this database"
- Nonsense
- All data will be shared eventually
- People will update your database using other applications
- Can you guarantee that someone won't use mysql to update your database?

# No Metadata (Excuse 2)

- "Database doesn't support those features"
- Nonsense
- MySQL 3.x is not a database
  - It's a set of data files with a vaguely SQL-like query syntax
- MySQL 4.x is a lot better
- MySQL 5.x is most of the way there
- Don't be constrained by using inferior tools

# DBIC::Schema::Loader

- Creates classes by querying your database metadata
- No more repeated data
- We are now DRY
- Schema definitions in one place
- But...
- Performance problems

# Performance Problems

- You don't really want to generate all your class definitions each time your program is run
- Need to generate the classes in advance
- `dump_to_dir` method
- Regenerate classes each time schema changes

# Alternative Approach

- Need one canonical definition of the data tables
- Doesn't need to be SQL DDL
- Could be in Perl code
- Write DBIx::Class definitions
- Generate DDL from those
- Harder approach
  - Might need to generate ALTER TABLE

# Conclusions

- ORM is a bridge between relational objects and program objects
- Avoid writing SQL in common cases
- DBIx::Class is the currently fashionable module
- Lots of plugins
- Caveat: ORM may be overkill for simple programs

# More Information

- Manual pages (on CPAN)
- DBIx::Class
- DBIx::Class::Manual::\*
- DBIx::Class::Schema::Loader
- Mailing list (Google for it)

# DBIx::Class Examples

- Create the CD database on your computer
- Use DBIx::Class::Schema::Loader to generate classes for your database
- Write code to insert data into the tables
- Write code to report on the data in the tables



# Exception Handling



# Error Handling

- How do you handle errors in your code?
- Return error values from subroutines
- ```
sub get_object {  
    my ($class, $id) = @_;  
    if (my $obj = find_obj_in_db($id)) {  
        return $obj;  
    } else {  
        return;  
    }  
}
```

Problems

- What if someone doesn't check return code?
- ```
my $obj = MyClass->get_object(100);
print $obj->name; # error
```
- Caller assumes that \$obj is a valid object
- Bad things follow

# Basic Exceptions

- Throw an exception instead
- ```
sub get_object {  
    my ($class, $id) = @_;  
    if (my $obj = find_obj_in_db($id)) {  
        return $obj;  
    } else {  
        die "No object found with id: $id";  
    }  
}
```
- Now caller has to deal with exceptions

Dealing with Exceptions

- Use eval to catch exceptions

- ```
my $obj = eval {
 MyClass->get_object(100)
};
```

```
if ($?) {
 # handle exception...
} else {
 print $obj->name; # error
}
```

# Exceptions as Objects

- `$_@` can be set to an object
- ```
sub get_object {  
    my ($class, $id) = @_  
    if (my $obj = find_obj_in_db($id)) {  
        return $obj;  
    } else {  
        die MyException->new(  
            type => 'obj_not_found',  
            id    => $id,  
        );  
    }  
}
```

Exception::Class

- Easy way to define your own exception objects
- Define exception hierarchies
- As recommended in *Perl Best Practices*

Define Exceptions

- ```
use Exception::Class (
 'MyException',
 'AnotherException' => { isa => 'MyException' },
 'YetAnotherException' => {
 isa => 'AnotherException',
 description =>
 'These exceptions are related to IPC'
 },
 'ExceptionWithFields' => {
 isa => 'YetAnotherException',
 fields => ['grandiosity', 'quixotic'],
 alias => 'throw_fields',
 },
);
```



# Using Exceptions

- `eval { MyException->throw( error => 'I feel funny.' ) };`

```
my $e;
```

```
if ($e = Exception::Class->caught('MyException')) {
 warn $e->error, "\n", $e->trace->as_string, "\n";
 warn join ' ', $e->euid, $e->egid, $e->uid,
 $e->gid, $e->pid, $e->time;
 exit;
}
elsif ($e = Exception::Class->caught('ExceptionWithFields')) {
 $e->quixotic ? do_something_wacky() : do_something_sane();
}
else {
 $e = Exception::Class->caught();
 ref $e ? $e->rethrow : die $e;
}
```

# Try Catch

- TryCatch adds “syntactic sugar”
- ```
try {  
    ...  
}  
catch ($e) {  
    ...  
}
```
- Looks a lot like many other languages

TryCatch with Scalar

- ```
try {
 some_function_that_might_die();
}
catch ($e) {
 if ($e =~ /some error/) {
 # handle error
 } else {
 die $e;
 }
}
```

# TryCatch with Object

- ```
try {
    some_function_that_might_die();
}
catch ($e) {
    if ($e->type eq 'file') {
        # handle error
    } else {
        die $e;
    }
}
```

Better Checks

- ```
try {
 some_function_that_might_die();
}
catch (My::Error $e) {
 # handle error
}
```

# Even Better Checks

- ```
try {
  some_function_that_might_die();
}
catch (HTTP::Error $e
  where { $e->code == 404 }) {
  # handle 404 error
}
```

Multiple Checks

- ```
try {
 some_function_that_might_die();
}
catch (HTTP::Error $e where { $e->code == 404 }) {
 # handle 404 error
}
catch (HTTP::Error $e where { $e->code == 500 }) {
 # handle 500 error
}
catch (HTTP::Error $e) {
 # handle other HTTP error
}
catch ($e) {
 # handle other error
}
```

# More on Exceptions

- Exceptions force callers to deal with error conditions
- But you have to explicitly code to throw exceptions
- `open my $fh, '<', 'somefile.txt'`  
`or die $!;`
- What if you forget to check the return value?



# Not Checking Errors

- ```
open my $fh, '<', 'somefile.txt';  
while (<$fh>) {  
    # do something useful  
}
```
- If the 'open' fails, you can't read any data
- You might not even get any warnings

Automatic Exceptions

- `use Fatal qw(open);`
- Comes with Perl since 5.003
- Errors in built-in functions become fatal errors
- This solves our previous problem
- `open my $fh, '<', 'somefile.txt';`

However

- Fatal.pm has its own problems
- Unintelligent check for success or failure
- Checks return value for true/false
- Assumes false is failure
- Can't be used if function can legitimately return a false value
 - e.g. fork

Also

- Nasty error messages
- ```
$ perl -MFatal=open -E'open my $fh,
"notthere" '
Can't open(GLOB(0x86357a4), notthere):
No such file or directory at (eval 1)
line 4
main::__ANON__('GLOB(0x86357a4)',
'notthere') called at -e line 1
```

# Enter autodie

- autodie is a cleverer Fatal
- In Perl core since 5.10.1
- Fatal needed a list of built-ins
- autodie assumes all built-ins
  - `use autodie;`

# Turning autodie on/off

- Lexically scoped
  - use autodie;
  - no autodie;
- Turn off for specific built-ins
  - no autodie 'open';

# Failing Calls

- autodie has more intelligence about failing calls
- Not just a boolean check
- Understands fork, system, etc

# Errors are Objects

- Errors thrown by autodie are objects
- Can be inspected for details of the error
- ```
try {  
    open my $fh, '<', 'not-there';  
}  
  
catch ($e) {  
    warn 'Error opening ', $e->args->[-1], "\n";  
    warn 'File: ', $e->file, "\n";  
    warn 'Function: ', $e->function, "\n";  
    warn 'Package: ', $e->package, "\n";  
    warn 'Caller: ', $e->caller, "\n";  
    warn 'Line: ', $e->line, "\n";  
}
```


Nicer Errors Too

- `$ perl -Mautodie -E'open my $fh, "not-there"'`
`Can't open($fh, 'not-there'): No such file or directory at -e line 1`

More Information

- perldoc TryCatch
 - See also Try::Tiny
- perldoc Fatal
- perldoc autodie
- autodie - The art of Klingon Programming
 - <http://perltraining.com.au/tips/2008-08-20.html>

Exceptions Examples

- Write a program that relies on autodie to throw exceptions
- Write a class which uses Exception::Class to throw exceptions
- Write code which uses your class

That's all folks

- Any questions?

