NAME

Error – Error/exception handling in an OO−ish way

VERSION

version 0.17029

SYNOPSIS

```
use Error qw(:try);

throw Error::Simple( "A simple error");

sub xyz {
    ...
    record Error::Simple("A simple error")
        and return;
}

unlink($file) or throw Error::Simple("$file: $!",$!);

try {
    do_some_stuff();
    die "error!" if $condition;
    throw Error::Simple "Oops!" if $other_condition;
}
catch Error::IO with {
    my $E = shift;
    print STDERR "File ", $E->{'-file'}, " had a problem\n";
}
except {
    my $E = shift;
    my $general_handler=sub {send_message $E->{-description}};
    return {
        UserException1 => $general_handler,
        UserException2 => $general_handler
    };
}
otherwise {
    print STDERR "Well I don't know what to say\n";
}
finally {
    close_the_garage_door_already(); # Should be reliable
}; # Don't forget the trailing ; or you might be surprised
```

DESCRIPTION

The `Error` package provides two interfaces. Firstly `Error` provides a procedural interface to exception handling. Secondly `Error` is a base class for errors/exceptions that can either be thrown, for subsequent catch, or can simply be recorded.

Errors in the class `Error` should not be thrown directly, but the user should throw errors from a sub-class of `Error`.

WARNING

Using the ‘‘Error’’ module is **no longer recommended** due to the black-magical nature of its syntactic sugar, which often tends to break. Its maintainers have stopped actively writing code that uses it, and discourage people from doing so. See the ‘‘SEE ALSO’’ section below for better recommendations.

PROCEDURAL INTERFACE

`Error` exports subroutines to perform exception handling. These will be exported if the `:try` tag is used in the `use` line.

try BLOCK CLAUSES

`try` is the main subroutine called by the user. All other subroutines exported are clauses to the try subroutine.

The BLOCK will be evaluated and, if no error is throw, try will return the result of the block.

`CLAUSES` are the subroutines below, which describe what to do in the event of an error being thrown within BLOCK.

catch CLASS with BLOCK

This clauses will cause all errors that satisfy `$err->isa(CLASS)` to be caught and handled by evaluating `BLOCK`.

`BLOCK` will be passed two arguments. The first will be the error being thrown. The second is a reference to a scalar variable. If this variable is set by the catch block then, on return from the catch block, try will continue processing as if the catch block was never found. The error will also be available in `$@`.

To propagate the error the catch block may call `$err->throw`

If the scalar reference by the second argument is not set, and the error is not thrown. Then the current try block will return with the result from the catch block.

except BLOCK

When `try` is looking for a handler, if an except clause is found `BLOCK` is evaluated. The return value from this block should be a HASHREF or a list of key-value pairs, where the keys are class names and the values are CODE references for the handler of errors of that type.

otherwise BLOCK

Catch any error by executing the code in `BLOCK`

When evaluated `BLOCK` will be passed one argument, which will be the error being processed. The error will also be available in `$@`.

Only one otherwise block may be specified per try block

finally BLOCK

Execute the code in `BLOCK` either after the code in the try block has successfully completed, or if the try block throws an error then `BLOCK` will be executed after the handler has completed.

If the handler throws an error then the error will be caught, the finally block will be executed and the error will be re-thrown.

Only one finally block may be specified per try block

COMPATIBILITY

Moose exports a keyword called `with` which clashes with Error's. This example returns a prototype mismatch error:

```
package MyTest;

use warnings;
use Moose;
use Error qw(:try);
```

(Thanks to `maik.hentsche@amd.com` for the report.).

CLASS INTERFACE

CONSTRUCTORS

The `Error` object is implemented as a HASH. This HASH is initialized with the arguments that are passed to it's constructor. The elements that are used by, or are retrievable by the `Error` class are listed below, other classes may add to these.

```
-file
-line
-text
-value
-object
```

If `-file` or `-line` are not specified in the constructor arguments then these will be initialized with the file name and line number where the constructor was called from.

If the error is associated with an object then the object should be passed as the `-object` argument. This will allow the `Error` package to associate the error with the object.

The `Error` package remembers the last error created, and also the last error associated with a package. This could either be the last error created by a sub in that package, or the last error which passed an object blessed into that package as the `-object` argument.

Error**->new()**
See the Error::Simple documentation.

throw ([ARGS])
Create a new `Error` object and throw an error, which will be caught by a surrounding `try` block, if there is one. Otherwise it will cause the program to exit.

`throw` may also be called on an existing error to re-throw it.

with ([ARGS])
Create a new `Error` object and returns it. This is defined for syntactic sugar, eg

```
die with Some::Error ( ... );
```

record ([ARGS])
Create a new `Error` object and returns it. This is defined for syntactic sugar, eg

```
record Some::Error ( ... )
    and return;
```

STATIC METHODS

prior ([PACKAGE])
Return the last error created, or the last error associated with `PACKAGE`

flush ([PACKAGE])
Flush the last error created, or the last error associated with `PACKAGE`.It is necessary to clear the error stack before exiting the package or uncaught errors generated using `record` will be reported.

```
$Error->flush;
```

OBJECT METHODS

stacktrace
If the variable `$Error::Debug` was non-zero when the error was created, then `stacktrace` returns a string created by calling `Carp::longmess`. If the variable was zero the `stacktrace` returns the text of the error appended with the filename and line number of where the error was created, providing the text does not end with a newline.

object
The object this error was associated with

file The file where the constructor of this error was called from

line The line where the constructor of this error was called from

text The text of the error

`$err->associate($obj)`

Associates an error with an object to allow error propagation. I.e:

```
$ber->encode(...) or
    return Error->prior($ber)->associate($ldap);
```

OVERLOAD METHODS

stringify

A method that converts the object into a string. This method may simply return the same as the `text` method, or it may append more information. For example the file name and line number.

By default this method returns the `-text` argument that was passed to the constructor, or the string `"Died"` if none was given.

value

A method that will return a value that can be associated with the error. For example if an error was created due to the failure of a system call, then this may return the numeric value of `$!` at the time.

By default this method returns the `-value` argument that was passed to the constructor.

PRE-DEFINED ERROR CLASSES

Error::Simple

This class can be used to hold simple error strings and values. It's constructor takes two arguments. The first is a text value, the second is a numeric value. These values are what will be returned by the overload methods.

If the text value ends with `at file line 1` as $@ strings do, then this information will be used to set the `-file` and `-line` arguments of the error object.

This class is used internally if an eval'd block die's with an error that is a plain string. (Unless `$Error::ObjectifyCallback` is modified)

`$Error::ObjectifyCallback`

This variable holds a reference to a subroutine that converts errors that are plain strings to objects. It is used by Error.pm to convert textual errors to objects, and can be overridden by the user.

It accepts a single argument which is a hash reference to named parameters. Currently the only named parameter passed is `'text'` which is the text of the error, but others may be available in the future.

For example the following code will cause Error.pm to throw objects of the class MyError::Bar by default:

```
sub throw_MyError_Bar
{
    my $args = shift;
    my $err = MyError::Bar->new();
    $err->{'MyBarText'} = $args->{'text'};
    return $err;
}

{
    local $Error::ObjectifyCallback = \&throw_MyError_Bar;

    # Error handling here.
}
```

MESSAGE HANDLERS

`Error` also provides handlers to extend the output of the `warn()` perl function, and to handle the printing of a thrown `Error` that is not caught or otherwise handled. These are not installed by default, but are requested using the `:warndie` tag in the `use` line.

```
use Error qw( :warndie );
```

These new error handlers are installed in `$SIG{__WARN__}` and `$SIG{__DIE__}`. If these handlers are already defined when the tag is imported, the old values are stored, and used during the new code. Thus, to arrange for custom handling of warnings and errors, you will need to perform something like the following:

```
BEGIN {
  $SIG{__WARN__} = sub {
    print STDERR "My special warning handler: $_[0]"
  };
}

use Error qw( :warndie );
```

Note that setting `$SIG{__WARN__}` after the `:warndie` tag has been imported will overwrite the handler that `Error` provides. If this cannot be avoided, then the tag can be explicitly `import`ed later

```
use Error;

$SIG{__WARN__} = ...;

import Error qw( :warndie );
```

EXAMPLE

The `__DIE__` handler turns messages such as

```
Can't call method "foo" on an undefined value at examples/warndie.pl line 16.
```

into

```
Unhandled perl error caught at toplevel:

  Can't call method "foo" on an undefined value

Thrown from: examples/warndie.pl:16

Full stack trace:

        main::inner('undef') called at examples/warndie.pl line 20
        main::outer('undef') called at examples/warndie.pl line 23
```

SEE ALSO

See Exception::Class for a different module providing Object-Oriented exception handling, along with a convenient syntax for declaring hierarchies for them. It doesn't provide Error's syntactic sugar of `try { ... }`, `catch { ... }`, etc. which may be a good thing or a bad thing based on what you want. (Because Error's syntactic sugar tends to break.)

Error::Exception aims to combine Error and Exception::Class ''with correct stringification''.

TryCatch and Try::Tiny are similar in concept to Error.pm only providing a syntax that hopefully breaks less.

KNOWN BUGS

None, but that does not mean there are not any.

AUTHORS

Graham Barr <gbarr@pobox.com>

The code that inspired me to write this was originally written by Peter Seibel <peter@weblogic.com> and adapted by Jesse Glick <jglick@sig.bsh.com>.

`:warndie` handlers added by Paul Evans <leonerd@leonerd.org.uk>

MAINTAINER

Shlomi Fish, <http://www.shlomifish.org/> .

PAST MAINTAINERS

Arun Kumar U <u_arunkumar@yahoo.com>

COPYRIGHT

SUPPORT

Websites

The following websites have more information about this module, and may be of help to you. As always, in addition to those websites please use your favorite search engine to discover more resources.

- MetaCPAN

  A modern, open-source CPAN search engine, useful to view POD in HTML format.

  <https://metacpan.org/release/Error>

- Search CPAN

  The default CPAN search engine, useful to view POD in HTML format.

  <http://search.cpan.org/dist/Error>

- RT: CPAN's Bug Tracker

  The RT (Request Tracker) website is the default bug/issue tracking system for CPAN.

  <https://rt.cpan.org/Public/Dist/Display.html?Name=Error>

- CPAN Ratings

  The CPAN Ratings is a website that allows community ratings and reviews of Perl modules.

  <http://cpanratings.perl.org/d/Error>

- CPANTS

  The CPANTS is a website that analyzes the Kwalitee (code metrics) of a distribution.

  <http://cpants.cpanauthors.org/dist/Error>

- CPAN Testers

  The CPAN Testers is a network of smoke testers who run automated tests on uploaded CPAN distributions.

  <http://www.cpantesters.org/distro/E/Error>

- CPAN Testers Matrix

  The CPAN Testers Matrix is a website that provides a visual overview of the test results for a distribution on various Perls/platforms.

  <http://matrix.cpantesters.org/?dist=Error>

- CPAN Testers Dependencies

  The CPAN Testers Dependencies is a website that shows a chart of the test results of all dependencies for a distribution.

  <http://deps.cpantesters.org/?module=Error>

Bugs / Feature Requests

Please report any bugs or feature requests by email to `bug-error at rt.cpan.org`, or through the web interface at <https://rt.cpan.org/Public/Bug/Report.html?Queue=Error>. You will be automatically notified of any progress on the request by the system.

Source Code

The code is open to the world, and available for you to hack on. Please feel free to browse it and play with it, or whatever. If you want to contribute patches, please send me a diff or prod me to pull from your repository :)

<https://github.com/shlomif/perl-error.pm>

```
git clone git://github.com/shlomif/perl-error.pm.git
```

AUTHOR

Shlomi Fish (http://www.shlomifish.org/)

BUGS

Please report any bugs or feature requests on the bugtracker website <https://github.com/shlomif/perl-error.pm/issues>

When submitting a bug or request, please include a test-file or a patch to an existing test-file that illustrates the bug or desired feature.

COPYRIGHT AND LICENSE