Exceptional Perl 6

Jonathan Worthington
@jnthnwrthngtn | 6guts.wordpress.com
In previous years...

Submit talk(s) to YAPC::EU

One (or maybe two) are accepted

Go to YAPC::EU

Give talk(s)
This year...

Submit talk(s) to YAPC::EU

Both rejected!
This year...

Submit talk(s) to YAPC::EU

Both rejected!

Factoring with Roles
This year...

Submit talk(s) to YAPC::EU

Both rejected!

Factoring with Roles

But...there already was a roles talk accepted
This year...

Submit talk(s) to YAPC::EU

Both rejected!

Factoring with Roles

Debugging Perl 6 Programs

But...there already was a roles talk accepted
This year...

Submit talk(s) to YAPC::EU

Both rejected!

Factoring with Roles
But...there already was a roles talk accepted

Debugging Perl 6 Programs
Well, debugging is, a rather boring topic, as we saw last year 😊
This year...

Submit talk(s) to YAPC::EU

Both rejected!

Then moritz++ - who did have accepted talks - couldn't come to YAPC 😞
This year...

Submit talk(s) to YAPC::EU

Both rejected!

Then moritz++ - who did have accepted talks - couldn't come to YAPC 😞

So, he passed this talk on to me 😊
But roles are niiiice...

And we all gotta debug...
So, today, I proudly present...
Exceptional Perl 6: A study of the design, throwing and catching of Perl 6 exceptions, which may be factored as roles, and their debugging
REPL Exploration

We attempt to call the `today` method on the class `Date`, but make a silly typo

```plaintext
> Date.todya
Method 'todya' not found for invocant of class 'Date'
```

This causes an exception to be thrown; a human-readable message describes the issue
To further explore exceptions, we use `try` in order to capture the exception into `!`

> `try Date.todya; say "Oops: $!"`
Oops: No such method 'todya' for invocant of type 'Date'

Interpolating it in a string once again yields the same human-readable message
From what we've seen so far, the contents of $! could be a string. But WHAT is it really?

```ruby
> try Date.todya; say $!.WHAT
X::Method::::NotFound()
```

From this we see that we don't have a string, but an object of type X::Method::::NotFound
REPL Exploration

To find out more about the exception object, we dump it using the `perl` method

```perl
> try Date.todya; say $!.perl
X::Method::NotFound.new(
    method => "todya",
    typename => "Date",
    private => Bool::False
)
```
REPL Exploration

The details held in the exception object are available through method calls

> try Date.todya; say $!.method()

todya

The methods enable programs to easily extract information about what went wrong
For Us and Them

Exception Objects

Stringify to a readable message for humans

Are typed objects with properties for programs
my regex insigline { ^ \s* [ <\?> | '('#' .* | '{' | '}')' ] \s* $ }

sub MAIN(*@files) { my $total = 0; for @files -> $filename { $total += lines($filename.IO).grep( { $_ !~~ /<&insigline>/ } ).elems; } say $total; }
When all the files passed to the script exist and are readable, things work out fine...

$ perl6 siglines.p6 A.pm B.pm
156

When one of them doesn't exist, less fine...

$ perl6 siglines.p6 A.pm B.pm C.pm
Unable to open filehandle from path 'C.pm'
We already know we could use try...

```perl
my $total = 0;
for @files -> $filename {
    try {
        $total += lines($filename.IO).grep(
            { $_ !~~ /<insigline>/ }
        ).elems;
    } 
    note "Can't read $filename" if $!;
} 
say $total;
```
We already know we could use try...

😊 Fixes the problem at hand

😊 Swallows any exception, not just IO ones

😢 We have to examine $!$ after the try, which doesn't feel as clean as we may desire
for @files -> $filename {
    try {
        $total += lines($filename.IO).grep(
            { $_ !~~ /<&insigline>/ }
        ).elems;
        CATCH {
            when X::IO {
                note "Couldn't read $filename";
            }
        }
    }
}

CATCH phasers trigger when an exception is thrown, and place it in $_ to allow smartmatching against it.
my $total = 0;
for @files -> $filename {
    $total += lines($filename.IO).grep(
        { $_ !~~ /<&insigline>/ }
    ).elems;
    CATCH {
        when X::IO {
            note "Couldn't read $filename";
        }
    }
}
say $total;

Any block can have a CATCH phaser, so we can place it directly in the loop body - much cleaner!
As CATCH goes inside of the scope, we can see the scope's lexicals!
CATCH and Rethrows

If a CATCH block does not successfully smart-match an exception, it is re-thrown for the next handler in the dynamic scope to consider.

```
CATCH {
    when X::IO {
        note "Couldn't read $filename";
    }
}
```

Anything not an X::IO is rethrown.
To catch any type of exception, use the **default** block inside of a **CATCH** block:

```
CATCH {
    when X::IO {
        note "Couldn't read $filename";
    }
    default {
        note "Failed to process $filename";
    }
}
```
Take a look, pass it on

A **CATCH** block that doesn't smart-match the exception may still take action based on it

```plaintext
CATCH {
    $logger.log_file_error($filename, $);
}
```

However, since it didn't successfully smart-match, the exception will be re-thrown
We have typed exceptions for errors from built-ins, the compiler, etc.

But where and how are they defined?
A peek inside Rakudo

Looking inside Rakudo's CORE.setting, we find that exception types are simply class definitions

```raku
my class X::Method::NotFound is Exception {
    has $.method;
    has $.typename;
    has Bool $.private = False;
    method message() {
        # ...
    }
}
```
Exceptional Perl 6: A study of the design, throwing and catching of Perl 6 exceptions, which may be factored as roles, and their debugging
A factoring challenge

All syntax errors should match \texttt{X::Syntax}

All Pod-related errors should match \texttt{X::Pod}

Clearly not all syntax errors are Pod errors, but not all Pod errors are going to be syntax errors

Roles are a neat solution to this kind of issue
Using roles

Roles provide a way to categorize exceptions and factor out shared properties

```
my role X::Comp is Exception {
    has $.filename;
    has $.line;
    has $.column;
    has @.modules;
    #...
}
my role X::Syntax does X::Comp {} }
my role X::Pod {} }
```

All compilation errors have a file, line, column and module trace
Using roles

Roles provide a way to categorize exceptions and factor out shared properties.

```perl
my role X::Comp is Exception {
    has $.filename;
    has $.line;
    has $.column;
    has @.modules;
    # ...
}
my role X::Syntax does X::Comp { }
my role X::Pod { }
```

Factor out the default parent Exception also
Role composition

Something that is a Pod error and a syntax error may compose both of the roles

```perl
my class X::Syntax::Pod::BeginWithoutIdentifier
  does X::Syntax
  does X::Pod
{
  method message() {
    '=begin must be followed by an identifier;'~
      ' (did you mean "=begin pod"?)'
  }
}
```
Why role composition?

When a role is composed into a class, its attributes and methods are copied to the class. If two roles supply the same method, it is detected as a conflict at compile time. The class must explicitly resolve the conflict, by providing a method of that name that does so.
Exceptional Perl 6: A study of the design, throwing and catching of Perl 6 exceptions, which may be factored as roles, and their debugging.
Poll::Simple

A very simple module for delivering polls

A list of options are passed to `new`

The `vote` method is used to vote on an option

There `result_graph` method renders the current results as a ASCII-art bar graph
class Poll::Simple {
    has @.options;
    has %!scores;

    submethod BUILD(:@!options) {
        %!scores{$_} = 0 for @!options;
    }

    method vote($option) {
        if $option eq any(@!options) {
            %!scores{$option}++;
        } else {
            die "Invalid poll option '\$option'";
        }
    }
}
Poll::Simple

The rendering of the bar graph will be handled by another module, Text::BarGraph

use Text::BarGraph;

class Poll::Simple {
  # ...

  method result_graph() {
    render_graph(%!scores);
  }
}
module Text::BarGraph;

sub render_graph(%data, :$label_char_limit = 25, :
$overall_width = 75) is export {
    my $label_chars = \[min\] %data.keys.max(*.chars),
        $label_char_limit;
    my $bar_width = $overall_width - ($label_chars + 2);
    my $max_value = %data.values.max;

    join "\n", %data.kv.map: -> $label, $value {
        my $entry = $label.chars > $label_chars
            ?? $label.substr(0, $label_chars)
            !! $label;
        $entry =~ ' ' x 1 + $label_chars - $label.chars;
        $entry =~ '=' x $bar_width * $value / $max_value;
    }
}
use Poll::Simple;

# Create a poll.
my $best_beer = Poll::Simple->new(
  options => '< Stout Lager Porter Ale Pilsner >
);

# Show the graph (all zero votes so far).
say $best_beer->result_graph();
use Poll::Simple;

# Create a poll.
my $best_beer = Poll::Simple.new(
  options => < Stout Lager Porter Ale Pilsner >
);

# Show the graph (all zero votes so far).
say $best_beer.result_graph();

$ perl6 -I. z.p6
Divide by zero
...
OH NOES!!!
Use the debugger!
There is no Perl 6 debugger!
OH YES THERE IS!
OH NO THERE ISN'T!
do live-demo() or fail;
Typed exceptions

At the moment, voting for an invalid option dies with a simple string

Let's make it a typed exception!
Adding Typed Exceptions

Our typed exception carries information on what is wrong and what to try, and can use it to produce a human-readable message also

class X::Poll::Simple::InvalidOption is Exception {
    has $.invalid;
    has @.valid;

    method message() {
        "'$.invalid()' is not a valid answer; vote any of:
        @.valid.join(",
        ")
    }
}
Using Typed Exceptions

The typed exception can be used with `die` in place of a string message

```perl
method vote($option) {
    if $option eq any(@!options) {
        %!scores{$option}++;
    }
    else {
        die X::Poll::Simple::InvalidOption.new(
            invalid => $option,
            valid   => @!options
        );
    }
}
```
Using Typed Exceptions

Alternatively, create the exception object and then call the \texttt{throw} method on it.
What's next?

Exceptions from the compiler and CORE setting are now typed; still some work in those issued by the meta-model and a couple of other cases.

Getting all of the exceptions documented in p6doc (for more on p6doc, see pmichaud's talk).

More work on the Rakudo debugger!
Thank you!
Questions?

E: jnthon@jnthon.net
T: @jnthonwrthngtn
W: 6guts.wordpress.com