Exceptions and Stack Traces

Exceptions

Friday, June 7, 13

Exceptions are an error handling mechanism. They are used to signal when it doesn't make sense for code to continue running - basically switching the default behavior from "continue running if nothing notices the error" to "stop running if nothing notices the error".

Exceptions

- signal exceptional conditions in your code

Friday, June 7, 13

Exceptions are an error handling mechanism. They are used to signal when it doesn't make sense for code to continue running - basically switching the default behavior from "continue running if nothing notices the error" to "stop running if nothing notices the error".

Exceptions

- signal exceptional conditions in your code
- used when it doesn't make sense for code to continue running

Friday, June 7, 13

Exceptions are an error handling mechanism. They are used to signal when it doesn't make sense for code to continue running - basically switching the default behavior from "continue running if nothing notices the error" to "stop running if nothing notices the error".

Exceptions

- used for errors and other exceptional conditions, not normal code flow

Exceptions are a general purpose mechanism, but when you get right down to it, they are really just a structured form of non-local goto. This is nice for errors, because it moves all of the error handling code out of the way of the actual code (making the actual code more readable), but if you start using exceptions for more than that, you lose that benefit.

Exceptions

- used for errors and other exceptional conditions, not normal code flow
- non-local code flow is confusing, so it should be minimized

Exceptions are a general purpose mechanism, but when you get right down to it, they are really just a structured form of non-local goto. This is nice for errors, because it moves all of the error handling code out of the way of the actual code (making the actual code more readable), but if you start using exceptions for more than that, you lose that benefit.

Exceptions

- used for two different things

Friday, June 7, 13

There are two cases that need consideration here - either you catch the exception, or you don't. If you catch the exception, you want an easily inspected data structure, but if you don't catch the exception, you want something human-readable.

Exceptions

- used for two different things
  - providing a human-readable error when things go wrong

There are two cases that need consideration here - either you catch the exception, or you don't. If you catch the exception, you want an easily inspected data structure, but if you don't catch the exception, you want something human-readable.

Exceptions

- used for two different things
  - providing a human-readable error when things go wrong
  - allowing code to handle errors that come up

Friday, June 7, 13

There are two cases that need consideration here - either you catch the exception, or you don't. If you catch the exception, you want an easily inspected data structure, but if you don't catch the exception, you want something human-readable.

Exceptions

- used for two different things
  - providing a human-readable error when things go wrong
    - allowing code to handle errors that come up
- you need to be able to handle both of these situations

Friday, June 7, 13

There are two cases that need consideration here - either you catch the exception, or you don't. If you catch the exception, you want an easily inspected data structure, but if you don't catch the exception, you want something human-readable.

Exceptions

Friday, June 7, 13

Most existing code (including the perl core!) throws strings for exceptions, mostly because this is easy. Unfortunately, this makes it quite hard to get information out of exceptions that you want to handle. You have to parse the information out of the human-readable string, every time you want to catch that type of exception. Then, you can never change your error messages again (including for things like translations) without potentially breaking downstream code. This is very unhelpful.

Exceptions

- string errors are common, but make handling exceptions difficult

Friday, June 7, 13

Most existing code (including the perl core!) throws strings for exceptions, mostly because this is easy. Unfortunately, this makes it quite hard to get information out of exceptions that you want to handle. You have to parse the information out of the human-readable string, every time you want to catch that type of exception. Then, you can never change your error messages again (including for things like translations) without potentially breaking downstream code. This is very unhelpful.

Exceptions

- string errors are common, but make handling exceptions difficult
  - the only way to deal with caught exceptions is through string matching

Friday, June 7, 13

Most existing code (including the perl core!) throws strings for exceptions, mostly because this is easy. Unfortunately, this makes it quite hard to get information out of exceptions that you want to handle. You have to parse the information out of the human-readable string, every time you want to catch that type of exception. Then, you can never change your error messages again (including for things like translations) without potentially breaking downstream code. This is very unhelpful.

Exceptions

- string errors are common, but make handling exceptions difficult
  - the only way to deal with caught exceptions is through string matching
  - but error messages need to change - clarifications, translations, etc

Friday, June 7, 13

Most existing code (including the perl core!) throws strings for exceptions, mostly because this is easy. Unfortunately, this makes it quite hard to get information out of exceptions that you want to handle. You have to parse the information out of the human-readable string, every time you want to catch that type of exception. Then, you can never change your error messages again (including for things like translations) without potentially breaking downstream code. This is very unhelpful.

Exceptions

Friday, June 7, 13

Most existing code (including the perl core!) throws strings for exceptions, mostly because this is easy. Unfortunately, this makes it quite hard to get information out of exceptions that you want to handle. You have to parse the information out of the human-readable string, every time you want to catch that type of exception. Then, you can never change your error messages again (including for things like translations) without potentially breaking downstream code. This is very unhelpful.

Exceptions

- Perl's builtin functions use special return values to indicate failure

Friday, June 7, 13

Most existing code (including the perl core!) throws strings for exceptions, mostly because this is easy. Unfortunately, this makes it quite hard to get information out of exceptions that you want to handle. You have to parse the information out of the human-readable string, every time you want to catch that type of exception. Then, you can never change your error messages again (including for things like translations) without potentially breaking downstream code. This is very unhelpful.

Exceptions

- Perl's builtin functions use special return values to indicate failure
  - This is **awful**

Friday, June 7, 13

Most existing code (including the perl core!) throws strings for exceptions, mostly because this is easy. Unfortunately, this makes it quite hard to get information out of exceptions that you want to handle. You have to parse the information out of the human-readable string, every time you want to catch that type of exception. Then, you can never change your error messages again (including for things like translations) without potentially breaking downstream code. This is very unhelpful.

Exceptions

- Perl's builtin functions use special return values to indicate failure
  - This is **awful**
  - Use `autodie` to promote those errors into exceptions

Friday, June 7, 13

Most existing code (including the perl core!) throws strings for exceptions, mostly because this is easy. Unfortunately, this makes it quite hard to get information out of exceptions that you want to handle. You have to parse the information out of the human-readable string, every time you want to catch that type of exception. Then, you can never change your error messages again (including for things like translations) without potentially breaking downstream code. This is very unhelpful.

Exceptions

- Perl can throw any kind of scalar, but objects are easiest to deal with

Friday, June 7, 13

But die() can take any scalar, not just a string. Throwing some kind of data structure means that the exception can provide a consistent and stable API that is unrelated to any presentation details.

Exceptions

- Perl can throw any kind of scalar, but objects are easiest to deal with
  - can use `->isa` or `->does` to handle only some types of exceptions

Friday, June 7, 13

But die() can take any scalar, not just a string. Throwing some kind of data structure means that the exception can provide a consistent and stable API that is unrelated to any presentation details.

Exceptions

- Perl can throw any kind of scalar, but objects are easiest to deal with
  - can use `->isa` or `->does` to handle only some types of exceptions
  - code that catches an exception can call methods to deal with it

Friday, June 7, 13

But die() can take any scalar, not just a string. Throwing some kind of data structure means that the exception can provide a consistent and stable API that is unrelated to any presentation details.

Exceptions

Friday, June 7, 13

If you use an object, and add stringification overloading to that object, then unhandled exceptions will continue to work the same way (displaying a human-readable error).

Exceptions

- Died: MyApp::Error=HASH(0x7ff66b8315f8) at test.pl line 7

Friday, June 7, 13

If you use an object, and add stringification overloading to that object, then unhandled exceptions will continue to work the same way (displaying a human-readable error).

Exceptions

- `Died: MyApp::Error=HASH(0x7ff66b8315f8) at test.pl line 7`
- stringify overloading can provide a readable (localizable!) message for users

Friday, June 7, 13

If you use an object, and add stringification overloading to that object, then unhandled exceptions will continue to work the same way (displaying a human-readable error).

Throwable

Friday, June 7, 13

Throwable is a Moose role that provides a basic framework for building exception objects. The base Throwable role just provides a 'throw' method, but the distribution also comes with a Throwable::Error class, which provides stringification, introspectable stack trace handling and a nicer constructor which supports just passing an error message rather than key/value pairs.

Throwable

- role that provides mechanisms for building exception classes

Friday, June 7, 13

Throwable is a Moose role that provides a basic framework for building exception objects. The base Throwable role just provides a 'throw' method, but the distribution also comes with a Throwable::Error class, which provides stringification, introspectable stack trace handling and a nicer constructor which supports just passing an error message rather than key/value pairs.

Throwable

- role that provides mechanisms for building exception classes
- provides a `throw` method

Friday, June 7, 13

Throwable is a Moose role that provides a basic framework for building exception objects. The base Throwable role just provides a 'throw' method, but the distribution also comes with a Throwable::Error class, which provides stringification, introspectable stack trace handling and a nicer constructor which supports just passing an error message rather than key/value pairs.

Throwable

- role that provides mechanisms for building exception classes
- provides a `throw` method
- comes with `Throwable::Error`, which is a basic error class

Friday, June 7, 13

Throwable is a Moose role that provides a basic framework for building exception objects. The base Throwable role just provides a 'throw' method, but the distribution also comes with a Throwable::Error class, which provides stringification, introspectable stack trace handling and a nicer constructor which supports just passing an error message rather than key/value pairs.

Throwable

- role that provides mechanisms for building exception classes
- provides a `throw` method
- comes with `Throwable::Error`, which is a basic error class
  - adds stack trace handling, one-arg constructor, stringification

Friday, June 7, 13

Throwable is a Moose role that provides a basic framework for building exception objects. The base Throwable role just provides a 'throw' method, but the distribution also comes with a Throwable::Error class, which provides stringification, introspectable stack trace handling and a nicer constructor which supports just passing an error message rather than key/value pairs.

Throwable

```
package MyApp::Error;
use Moose;
with 'Throwable';
use overload '""' => sub { shift->message };

has message => (
    is       => 'ro',
    isa      => 'Str',
    required => 1,
);

# ...

MyApp::Error->throw(message => "Invalid argument");
```


Friday, June 7, 13

This is an example of how to define and use a basic Throwable class. This will die with an exception which exception handlers can get the message from via the ->message method, and will be stringified to that message if it ever reaches the end user.

Throwable

```
package MyApp::Error;
use Moose;
extends 'Throwable::Error';

# ...

MyApp::Error->throw("Invalid argument");
```


Friday, June 7, 13

And this is what that error class would look like using Throwable::Error. Note that in addition to the functionality in the previous example, this class will also have an introspectable stack trace, via Devel::StackTrace (this is configurable to use whatever stack trace introspection class you prefer).

HTTP::Throwable

Friday, June 7, 13

A good example of a set of exception classes using Throwable is HTTP::Throwable. These provide exception classes corresponding to HTTP error codes (403 Forbidden, 500 Internal Server Error, etc). As mentioned earlier, exceptions should be used for exceptional conditions, so there are no classes corresponding to things like 200 OK, because those should be part of your normal code path.

HTTP::Throwable

- uses `Throwable` to provide exceptions corresponding to HTTP error statuses

Friday, June 7, 13

A good example of a set of exception classes using Throwable is HTTP::Throwable. These provide exception classes corresponding to HTTP error codes (403 Forbidden, 500 Internal Server Error, etc). As mentioned earlier, exceptions should be used for exceptional conditions, so there are no classes corresponding to things like 200 OK, because those should be part of your normal code path.

HTTP::Throwable

- uses `Throwable` to provide exceptions corresponding to HTTP error statuses
  - note: not success statuses - exceptions are for exceptional conditions

Friday, June 7, 13

A good example of a set of exception classes using Throwable is HTTP::Throwable. These provide exception classes corresponding to HTTP error codes (403 Forbidden, 500 Internal Server Error, etc). As mentioned earlier, exceptions should be used for exceptional conditions, so there are no classes corresponding to things like 200 OK, because those should be part of your normal code path.

HTTP::Throwable

- uses `Throwable` to provide exceptions corresponding to HTTP error statuses
  - note: not success statuses - exceptions are for exceptional conditions
- `Plack::Middleware::HTTPExceptions` can catch this and generate a response

Friday, June 7, 13

A good example of a set of exception classes using Throwable is HTTP::Throwable. These provide exception classes corresponding to HTTP error codes (403 Forbidden, 500 Internal Server Error, etc). As mentioned earlier, exceptions should be used for exceptional conditions, so there are no classes corresponding to things like 200 OK, because those should be part of your normal code path.

HTTP::Throwable

- uses `Throwable` to provide exceptions corresponding to HTTP error statuses
  - note: not success statuses - exceptions are for exceptional conditions
- `Plack::Middleware::HTTPExceptions` can catch this and generate a response
- human-readable messages can be provided to show up in the HTTP body

A good example of a set of exception classes using Throwable is HTTP::Throwable. These provide exception classes corresponding to HTTP error codes (403 Forbidden, 500 Internal Server Error, etc). As mentioned earlier, exceptions should be used for exceptional conditions, so there are no classes corresponding to things like 200 OK, because those should be part of your normal code path.

HTTP::Throwable

```
sub call {
    my $self = shift;
    my ($req) = @_;

    http_throw(MethodNotAllowed => {
        message => "only GET requests are allowed",
    }) unless $req->method eq 'GET';

    http_throw(Found => {
        location => '/somewhere/else',
    }) if $req->path eq '/somewhere';

    # ...
}
```


Friday, June 7, 13

This is an example of using HTTP::Throwable. The http_throw function is basically sugar around calling the throw method on the specified HTTP::Throwable class. Constructor arguments for that class can also be provided - the 'message' attribute is always available to set the body of the HTTP response (useful for adding more diagnostics to 4xx and 5xx errors), and individual exception classes can have additional attributes corresponding to things like headers.

HTTP::Throwable

Friday, June 7, 13

Something to point out, for people who are used to using things like Exception::Class, is that these classes are also all autogenerated. Creating new classes with Moose is easy - there is no need to also provide custom syntax for creating exception classes separately. Also, the HTTP::Throwable exception classes use Throwable directly (not via Throwable::Exception), because stack traces are very rarely useful here - the only thing that something catching the exception is likely to want to do is return the corresponding HTTP response.

HTTP::Throwable

- autogenerates exception classes using metaprogramming

Friday, June 7, 13

Something to point out, for people who are used to using things like Exception::Class, is that these classes are also all autogenerated. Creating new classes with Moose is easy - there is no need to also provide custom syntax for creating exception classes separately. Also, the HTTP::Throwable exception classes use Throwable directly (not via Throwable::Exception), because stack traces are very rarely useful here - the only thing that something catching the exception is likely to want to do is return the corresponding HTTP response.

HTTP::Throwable

- autogenerates exception classes using metaprogramming
- shortcuts like `Exception::Class` aren't really necessary with Moose

Friday, June 7, 13

Something to point out, for people who are used to using things like Exception::Class, is that these classes are also all autogenerated. Creating new classes with Moose is easy - there is no need to also provide custom syntax for creating exception classes separately. Also, the HTTP::Throwable exception classes use Throwable directly (not via Throwable::Exception), because stack traces are very rarely useful here - the only thing that something catching the exception is likely to want to do is return the corresponding HTTP response.

Try::Tiny

Friday, June 7, 13

The flipside to throwing exceptions is being able to catch them. Unfortunately, catching exceptions in perl is much less straightforward, mostly due to the use of global variables, and the idioms for catching exceptions that people developed because of that.

Try::Tiny

- for historical reasons, catching exceptions in Perl is hard

Friday, June 7, 13

The flipside to throwing exceptions is being able to catch them. Unfortunately, catching exceptions in perl is much less straightforward, mostly due to the use of global variables, and the idioms for catching exceptions that people developed because of that.

Try::Tiny

- for historical reasons, catching exceptions in Perl is hard
- let's go shopping!

Friday, June 7, 13

The flipside to throwing exceptions is being able to catch them. Unfortunately, catching exceptions in perl is much less straightforward, mostly due to the use of global variables, and the idioms for catching exceptions that people developed because of that.

Try::Tiny

```
# WRONG
my $val = eval {

    # <try> code here

};
if ($@) {

    # <catch> code here

}
```


Friday, June 7, 13

For instance, this is one of the most common ways you'll see exception handling written. Unfortunately, it has several edge cases that can cause some serious headaches. For instance, what happens if the exception that is thrown evaluates to a false value? (A common case for this is failing to provide a message for an exception class which stringifies to that message - boolean overloading is generated from stringification overloading.)

Try::Tiny

```
# STILL WRONG
my ($val, $success);
$success = eval {
    $val = sub {

        # <try> code here

    }->();
    1;
}
if (!$success) {

    # <catch> code here

};
```


Friday, June 7, 13

So we can fix that issue by doing this. If an exception is thrown, the eval is guaranteed to return undef, and every other code path in the <try> block will return 1, so we can be sure that returning 1 means that no exceptions were thrown. If we're sure that the <try> block contains no 'return' statements, the sub { }->() construct can be replaced by do { }, and if it's just a single statement, it can just be included inline, but the sub version is necessary to be able to handle all possible cases.

Try::Tiny

```
my ($val, $success, $err);
$err = do {
    local $@;
    $success = eval {
        $val = sub {
            # <try> code here
        }->();
        1;
    };
    $@;
}
if (!$success) {
    # <catch, with $err instead of $@>
}
```


Friday, June 7, 13

But that still doesn't take care of all of the issues, if you are using a version of perl before 5.14. Before 5.14, $@ could be modified globally from within DESTROY methods. In that case, it's possible that you could create an object during the <try> block, throw an exception, that object runs its destructor which calls an eval which succeeds, and by the time the <try> block exits, $@ is undef. The code in the previous example will work fine in this case (since it doesn't check $@ for truth), but if you're the one writing the eval in the destructor, you could be causing issues for other code which isn't as careful. This version takes care of that issue by making sure to localize $@ around the eval, so the global $@ isn't modified. Unfortunately, the other direction (where we handle exceptions properly but a destructor uses eval without being careful enough) will still cause the exception itself to be lost (although we will still know that an exception was thrown).

Try::Tiny

Friday, June 7, 13

Try::Tiny takes all of that complicated logic and wraps it up in a much simpler syntax. The 'try' block is run, and if any exceptions are thrown, the 'catch' block is run with the exception in $_ and @_. Afterwards, regardless of whether an exception was thrown or not, or even if it was rethrown from the catch block, the 'finally' block will be run.

Try::Tiny

- wraps up the complexity and edge cases in a simple API

Friday, June 7, 13

Try::Tiny takes all of that complicated logic and wraps it up in a much simpler syntax. The 'try' block is run, and if any exceptions are thrown, the 'catch' block is run with the exception in $_ and @_. Afterwards, regardless of whether an exception was thrown or not, or even if it was rethrown from the catch block, the 'finally' block will be run.

Try::Tiny

- wraps up the complexity and edge cases in a simple API
- provides `try`, `catch`, and `finally` blocks that function as you'd expect

Friday, June 7, 13

Try::Tiny takes all of that complicated logic and wraps it up in a much simpler syntax. The 'try' block is run, and if any exceptions are thrown, the 'catch' block is run with the exception in $_ and @_. Afterwards, regardless of whether an exception was thrown or not, or even if it was rethrown from the catch block, the 'finally' block will be run.

Try::Tiny

```
my $val = try {
    get_val();
}
catch {
    warn $_;
    undef;
};
```


Friday, June 7, 13

This is a basic example of the syntax. If get_val throws an exception, then that exception will be turned into a warning and undef will be returned instead.

Try::Tiny

```
set_global_logger('special');
try {
    something_that_might_die();
}
catch {
    if (blessed($_) && $_->isa('Error::Unimportant')) {
        warn $_;
    }
    else {
        die $_;
    }
}
finally {
    set_global_logger('main');
};
```


Friday, June 7, 13

This is an example of when a finally block may be useful. If you may want to propagate the exception up further, a finally block allows you to avoid having to duplicate the cleanup logic in multiple places.

Try::Tiny

```
if ( try { require File::HomeDir } ) {
    $self->set_home(File::HomeDir->my_home);
}
```


Friday, June 7, 13

You can also just use a try block on its own, if you're testing for some optional features, where you don't care what errors might be thrown.

Exceptions & Moose

Friday, June 7, 13

In general, Moose's policy is to use exceptions whenever possible. Moose doesn't provide a way to make the constructor return undef if creating the object failed, for instance - it will always throw an exception. Since Moose is mostly just Perl, a lot of exceptions don't need to be handled by Moose itself (calling invalid methods, creating cycles in the inheritance graph, etc), so most exceptions that you will see that are generated by Moose come from either attributes or roles, since those are the two main things that Moose provides over perl itself.

Exceptions & Moose

- Moose uses exceptions for many different error conditions

Friday, June 7, 13

In general, Moose's policy is to use exceptions whenever possible. Moose doesn't provide a way to make the constructor return undef if creating the object failed, for instance - it will always throw an exception. Since Moose is mostly just Perl, a lot of exceptions don't need to be handled by Moose itself (calling invalid methods, creating cycles in the inheritance graph, etc), so most exceptions that you will see that are generated by Moose come from either attributes or roles, since those are the two main things that Moose provides over perl itself.

Exceptions & Moose

- Moose uses exceptions for many different error conditions
- most Moose exceptions you'll see come from either attributes or roles

Friday, June 7, 13

In general, Moose's policy is to use exceptions whenever possible. Moose doesn't provide a way to make the constructor return undef if creating the object failed, for instance - it will always throw an exception. Since Moose is mostly just Perl, a lot of exceptions don't need to be handled by Moose itself (calling invalid methods, creating cycles in the inheritance graph, etc), so most exceptions that you will see that are generated by Moose come from either attributes or roles, since those are the two main things that Moose provides over perl itself.

Exceptions & Moose

Friday, June 7, 13

Attributes can throw exceptions either from the constructor or from accessors. Both of these can throw errors due to failing type constraints, and this is the most common source of exceptions from Moose, overall. The constructor can also throw exceptions about required attributes being missing in the constructor parameters, and accessors (and delegations) can throw errors if you pass the wrong number of arguments to them.

Exceptions & Moose

- Attributes

Friday, June 7, 13

Attributes can throw exceptions either from the constructor or from accessors. Both of these can throw errors due to failing type constraints, and this is the most common source of exceptions from Moose, overall. The constructor can also throw exceptions about required attributes being missing in the constructor parameters, and accessors (and delegations) can throw errors if you pass the wrong number of arguments to them.

Exceptions & Moose

- Attributes
  - type constraint failures

Friday, June 7, 13

Attributes can throw exceptions either from the constructor or from accessors. Both of these can throw errors due to failing type constraints, and this is the most common source of exceptions from Moose, overall. The constructor can also throw exceptions about required attributes being missing in the constructor parameters, and accessors (and delegations) can throw errors if you pass the wrong number of arguments to them.

Exceptions & Moose

- Attributes
  - type constraint failures
  - missing required attributes

Attributes can throw exceptions either from the constructor or from accessors. Both of these can throw errors due to failing type constraints, and this is the most common source of exceptions from Moose, overall. The constructor can also throw exceptions about required attributes being missing in the constructor parameters, and accessors (and delegations) can throw errors if you pass the wrong number of arguments to them.

Exceptions & Moose

- Attributes
  - type constraint failures
  - missing required attributes
  - invalid arguments passed to accessors or delegations

Friday, June 7, 13

Attributes can throw exceptions either from the constructor or from accessors. Both of these can throw errors due to failing type constraints, and this is the most common source of exceptions from Moose, overall. The constructor can also throw exceptions about required attributes being missing in the constructor parameters, and accessors (and delegations) can throw errors if you pass the wrong number of arguments to them.

Exceptions & Moose

Friday, June 7, 13

In roles, the most common errors will come from method conflicts (composing more than one role that defines the same method) and unfulfilled required methods (composing a role with a 'requires' statement without a corresponding real method).

Exceptions & Moose

- Roles

Friday, June 7, 13

In roles, the most common errors will come from method conflicts (composing more than one role that defines the same method) and unfulfilled required methods (composing a role with a 'requires' statement without a corresponding real method).

Exceptions & Moose

- Roles
  - method conflicts

Friday, June 7, 13

In roles, the most common errors will come from method conflicts (composing more than one role that defines the same method) and unfulfilled required methods (composing a role with a 'requires' statement without a corresponding real method).

Exceptions & Moose

- Roles
  - method conflicts
  - unfulfilled method requirements

Friday, June 7, 13

In roles, the most common errors will come from method conflicts (composing more than one role that defines the same method) and unfulfilled required methods (composing a role with a 'requires' statement without a corresponding real method).

Exceptions & Moose

When validating type constraints, Moose generates a default error message for your type to use. Since it has to be general enough to handle any type constraint and any value, though, it's often not quite as helpful as it could be. If you know more about your type, you can provide a more helpful error message.

Exceptions & Moose

- type constraint errors can be made more useful with the 'message' option

Friday, June 7, 13

When validating type constraints, Moose generates a default error message for your type to use. Since it has to be general enough to handle any type constraint and any value, though, it's often not quite as helpful as it could be. If you know more about your type, you can provide a more helpful error message.

Exceptions & Moose

- type constraint errors can be made more useful with the 'message' option

```
subtype 'PositiveInt',
    as 'Int',
    where { $_ > 0 },
    message {
       "The value must be a positive integer, not $_"
    };
```


Friday, June 7, 13

When validating type constraints, Moose generates a default error message for your type to use. Since it has to be general enough to handle any type constraint and any value, though, it's often not quite as helpful as it could be. If you know more about your type, you can provide a more helpful error message.

Exceptions & Moose

Friday, June 7, 13

One problem with the default error message is that it just stringifies any value that it receives, which sometimes makes it hard to figure out what went wrong. If you install Devel::PartialDump, Moose will use that to stringify references, providing a much more readable result.

Exceptions & Moose

- installing `Devel::PartialDump` will make type constraint errors more readable

Friday, June 7, 13

One problem with the default error message is that it just stringifies any value that it receives, which sometimes makes it hard to figure out what went wrong. If you install Devel::PartialDump, Moose will use that to stringify references, providing a much more readable result.

Exceptions & Moose

- installing `Devel::PartialDump` will make type constraint errors more readable
- before:

Friday, June 7, 13

One problem with the default error message is that it just stringifies any value that it receives, which sometimes makes it hard to figure out what went wrong. If you install Devel::PartialDump, Moose will use that to stringify references, providing a much more readable result.

Exceptions & Moose

- installing `Devel::PartialDump` will make type constraint errors more readable
- before:

> Attribute (bar) does not pass the type constraint because: Validation failed for 'Str' with value ARRAY(0x17c9808)

Friday, June 7, 13

One problem with the default error message is that it just stringifies any value that it receives, which sometimes makes it hard to figure out what went wrong. If you install Devel::PartialDump, Moose will use that to stringify references, providing a much more readable result.

Exceptions & Moose

- installing `Devel::PartialDump` will make type constraint errors more readable
- before:

  Attribute (bar) does not pass the type constraint because: Validation failed for 'Str' with value ARRAY(0x17c9808)

- after:

Friday, June 7, 13

One problem with the default error message is that it just stringifies any value that it receives, which sometimes makes it hard to figure out what went wrong. If you install Devel::PartialDump, Moose will use that to stringify references, providing a much more readable result.

Exceptions & Moose

- installing `Devel::PartialDump` will make type constraint errors more readable
- before:

  Attribute (bar) does not pass the type constraint because: Validation failed for 'Str' with value ARRAY(0x17c9808)

- after:

  Attribute (bar) does not pass the type constraint because: Validation failed for 'Str' with value [1, 2, 3]

One problem with the default error message is that it just stringifies any value that it receives, which sometimes makes it hard to figure out what went wrong. If you install Devel::PartialDump, Moose will use that to stringify references, providing a much more readable result.

Exceptions & Moose

Friday, June 7, 13

As with perl itself, Moose throws string exceptions currently. We are, however, currently working on changing this - hopefully Moose will be using Throwable for its error system sometime soon.

Exceptions & Moose

- currently, all Moose exceptions are strings

Friday, June 7, 13

As with perl itself, Moose throws string exceptions currently. We are, however, currently working on changing this - hopefully Moose will be using Throwable for its error system sometime soon.

Exceptions & Moose

- currently, all Moose exceptions are strings
- this is being worked on

Friday, June 7, 13

As with perl itself, Moose throws string exceptions currently. We are, however, currently working on changing this - hopefully Moose will be using Throwable for its error system sometime soon.

Exceptions & Moose

- currently, all Moose exceptions are strings
- this is being worked on
- Gnome Outreach Program for Women

Friday, June 7, 13

As with perl itself, Moose throws string exceptions currently. We are, however, currently working on changing this - hopefully Moose will be using Throwable for its error system sometime soon.

Exceptions & Moose

- currently, all Moose exceptions are strings
- this is being worked on
- Gnome Outreach Program for Women
- (Upasana Shukla)++ @sweet_kid__

Friday, June 7, 13

As with perl itself, Moose throws string exceptions currently. We are, however, currently working on changing this - hopefully Moose will be using Throwable for its error system sometime soon.

Exceptions & Moose

Friday, June 7, 13

Don't take Moose not using structured exceptions as a reason to not use them yourself.

Exceptions & Moose

- in the meantime, use objects for your own exceptions

Friday, June 7, 13

Don't take Moose not using structured exceptions as a reason to not use them yourself.

Stack Traces

Friday, June 7, 13

Stack traces are one of the most useful tools when debugging, and so since Moose currently uses string exceptions, it generates those exceptions using Carp::confess. This is often overwhelming to people new to Moose, but the Moose community has found through years of development that once you get used to them and learn how to read them, they can be invaluable. Learning how to skim through stack traces quickly and effectively can make debugging significantly easier.

Stack Traces

- Moose uses `Carp::confess` by default

Friday, June 7, 13

Stack traces are one of the most useful tools when debugging, and so since Moose currently uses string exceptions, it generates those exceptions using Carp::confess. This is often overwhelming to people new to Moose, but the Moose community has found through years of development that once you get used to them and learn how to read them, they can be invaluable. Learning how to skim through stack traces quickly and effectively can make debugging significantly easier.

Stack Traces

- Moose uses `Carp::confess` by default
- this can be overwhelming at first, but it really is helpful

Stack traces are one of the most useful tools when debugging, and so since Moose currently uses string exceptions, it generates those exceptions using Carp::confess. This is often overwhelming to people new to Moose, but the Moose community has found through years of development that once you get used to them and learn how to read them, they can be invaluable. Learning how to skim through stack traces quickly and effectively can make debugging significantly easier.

Stack Traces

- Moose uses `Carp::confess` by default
- this can be overwhelming at first, but it really is helpful
- learning how to skim through stack traces is an important debugging skill

Friday, June 7, 13

Stack traces are one of the most useful tools when debugging, and so since Moose currently uses string exceptions, it generates those exceptions using Carp::confess. This is often overwhelming to people new to Moose, but the Moose community has found through years of development that once you get used to them and learn how to read them, they can be invaluable. Learning how to skim through stack traces quickly and effectively can make debugging significantly easier.

Stack Traces

Friday, June 7, 13

The first place you should look is at the first line in the output. This tells you what the error is, and in a lot of cases will be sufficient to work out the underlying problem by itself.

Stack Traces

- knowing what failed is the most important part

Friday, June 7, 13

The first place you should look is at the first line in the output. This tells you what the error is, and in a lot of cases will be sufficient to work out the underlying problem by itself.

Stack Traces

- knowing what failed is the most important part
- always start at the top

Friday, June 7, 13

The first place you should look is at the first line in the output. This tells you what the error is, and in a lot of cases will be sufficient to work out the underlying problem by itself.

Stack Traces

Attribute (bar) does not pass the type constraint because: Validation failed for 'Str' with value [ ] at /path/to/Moose/Meta/Attribute.pm line 1275.

```
Moose::Meta::Attribute::verify_against_type_constraint('Moose::Meta::Attribute=HASH
line 1262
Moose::Meta::Attribute::_coerce_and_verify('Moose::Meta::Attribute=HASH(0x321e090)'
Moose::Meta::Attribute::initialize_instance_slot('Moose::Meta::Attribute=HASH(0x321
Class/MOP/Class.pm line 525
Class::MOP::Class::_construct_instance('Moose::Meta::Class=HASH(0x322f430)', 'HASH(
Class::MOP::Class::new_object('Moose::Meta::Class=HASH(0x322f430)', 'HASH(0x34fea40
Moose::Meta::Class::new_object('Moose::Meta::Class=HASH(0x322f430)', 'HASH(0x34fea4
Moose::Object::new('Foo', 'bar', 'ARRAY(0x3348cf8)') called at test.psgi
Plack::Sandbox::test_2epsgi::create_foo('bar', 'ARRAY(0x3348cf8)') called at test.p
Plack::Sandbox::test_2epsgi::__ANON__('HASH(0x312e3e8)') called at /path/to/Plack/M
Plack::Middleware::Lint::call('Plack::Middleware::Lint=HASH(0x331cab8)', 'HASH(0x31
Plack::Component::__ANON__('HASH(0x312e3e8)') called at /path/to/Plack/Middleware/S
[...]
```


Friday, June 7, 13

For instance, here we see we have a type constraint error.

Stack Traces

Friday, June 7, 13

In some cases, looking at what threw the error can also provide more information. You can see the function that threw the error, which will hopefully have a reasonably informative name, and you can also see the arguments that it was passed.

Stack Traces

- If the message isn't clear, looking at the next line can be helpful

Friday, June 7, 13

In some cases, looking at what threw the error can also provide more information. You can see the function that threw the error, which will hopefully have a reasonably informative name, and you can also see the arguments that it was passed.

Stack Traces

- If the message isn't clear, looking at the next line can be helpful
- It will show what threw the error and what arguments it had

Friday, June 7, 13

In some cases, looking at what threw the error can also provide more information. You can see the function that threw the error, which will hopefully have a reasonably informative name, and you can also see the arguments that it was passed.

Stack Traces

```
Attribute (bar) does not pass the type constraint because: Validation failed
for 'Str' with value [  ] at /path/to/Moose/Meta/Attribute.pm line 1275.
```

**Moose::Meta::Attribute::verify_against_type_constraint('Moose::Meta::Attribute
=HASH(0x321e090)', 'ARRAY(0x3348cf8)', 'instance', 'Foo=HASH(0x34fec80)')
called at /path/to/Moose/Meta/Attribute.pm line 1262**

```
Moose::Meta::Attribute::_coerce_and_verify('Moose::Meta::Attribute=HASH(0x321e090)'
Moose::Meta::Attribute::initialize_instance_slot('Moose::Meta::Attribute=HASH(0x321
Class/MOP/Class.pm line 525
Class::MOP::Class::_construct_instance('Moose::Meta::Class=HASH(0x322f430)', 'HASH(
Class::MOP::Class::new_object('Moose::Meta::Class=HASH(0x322f430)', 'HASH(0x34fea40
Moose::Meta::Class::new_object('Moose::Meta::Class=HASH(0x322f430)', 'HASH(0x34fea4
Moose::Object::new('Foo', 'bar', 'ARRAY(0x3348cf8)') called at test.psgi
Plack::Sandbox::test_2epsgi::create_foo('bar', 'ARRAY(0x3348cf8)') called at test.p
Plack::Sandbox::test_2epsgi::__ANON__('HASH(0x312e3e8)') called at /path/to/Plack/M
Plack::Middleware::Lint::call('Plack::Middleware::Lint=HASH(0x331cab8)', 'HASH(0x31
Plack::Component::__ANON__('HASH(0x312e3e8)') called at /path/to/Plack/Middleware/S
[...]
```


In this case, we don't get much extra information - the error already said it was a type constraint failure.

Stack Traces

Friday, June 7, 13

So the next place we want to go to is where your code called into Moose. You can find this by skipping through any stack frames that refer to calls to functions in Moose::* and Class::MOP::*. The first line that doesn't include that is the part of your code that called into Moose, and the line above it is the Moose code that it called. Trying to find this by reading up from the bottom of the stacktrace is typically not very helpful, since that is frequently just framework code (in persistent environments anyway).

Stack Traces

- Next, skip past any stack frames within Moose::* or Class::MOP::*

Friday, June 7, 13

So the next place we want to go to is where your code called into Moose. You can find this by skipping through any stack frames that refer to calls to functions in Moose::* and Class::MOP::*. The first line that doesn't include that is the part of your code that called into Moose, and the line above it is the Moose code that it called. Trying to find this by reading up from the bottom of the stacktrace is typically not very helpful, since that is frequently just framework code (in persistent environments anyway).

Stack Traces

- Next, skip past any stack frames within Moose::* or Class::MOP::*
- This will be the place in your code that the error is coming from

So the next place we want to go to is where your code called into Moose. You can find this by skipping through any stack frames that refer to calls to functions in Moose::* and Class::MOP::*. The first line that doesn't include that is the part of your code that called into Moose, and the line above it is the Moose code that it called. Trying to find this by reading up from the bottom of the stacktrace is typically not very helpful, since that is frequently just framework code (in persistent environments anyway).

Stack Traces

- Next, skip past any stack frames within Moose::* or Class::MOP::*
- This will be the place in your code that the error is coming from
- The line above that will be where you called into Moose

Friday, June 7, 13

So the next place we want to go to is where your code called into Moose. You can find this by skipping through any stack frames that refer to calls to functions in Moose::* and Class::MOP::*. The first line that doesn't include that is the part of your code that called into Moose, and the line above it is the Moose code that it called. Trying to find this by reading up from the bottom of the stacktrace is typically not very helpful, since that is frequently just framework code (in persistent environments anyway).

Stack Traces

- Next, skip past any stack frames within Moose::* or Class::MOP::*
- This will be the place in your code that the error is coming from
- The line above that will be where you called into Moose
- Reading from the bottom isn't usually helpful in persistent environments

Friday, June 7, 13

So the next place we want to go to is where your code called into Moose. You can find this by skipping through any stack frames that refer to calls to functions in Moose::* and Class::MOP::*. The first line that doesn't include that is the part of your code that called into Moose, and the line above it is the Moose code that it called. Trying to find this by reading up from the bottom of the stacktrace is typically not very helpful, since that is frequently just framework code (in persistent environments anyway).

Stack Traces

```
Attribute (bar) does not pass the type constraint because: Validation failed
for 'Str' with value [  ] at /path/to/Moose/Meta/Attribute.pm line 1275.

    Moose::Meta::Attribute::verify_against_type_constraint('Moose::Meta::Attribute=HASH
    line 1262
    Moose::Meta::Attribute::_coerce_and_verify('Moose::Meta::Attribute=HASH(0x321e090)'
    Moose::Meta::Attribute::initialize_instance_slot('Moose::Meta::Attribute=HASH(0x321
    Class/MOP/Class.pm line 525
    Class::MOP::Class::_construct_instance('Moose::Meta::Class=HASH(0x322f430)', 'HASH(
    Class::MOP::Class::new_object('Moose::Meta::Class=HASH(0x322f430)', 'HASH(0x34fea40
    Moose::Meta::Class::new_object('Moose::Meta::Class=HASH(0x322f430)', 'HASH(0x34fea4

    Moose::Object::new('Foo', 'bar', 'ARRAY(0x3348cf8)') called at test.psgi
    Plack::Sandbox::test_2epsgi::create_foo('bar', 'ARRAY(0x3348cf8)') called at
    test.psgi line 13

    Plack::Sandbox::test_2epsgi::__ANON__('HASH(0x312e3e8)') called at /path/to/Plack/M
    Plack::Middleware::Lint::call('Plack::Middleware::Lint=HASH(0x331cab8)', 'HASH(0x31
    Plack::Component::__ANON__('HASH(0x312e3e8)') called at /path/to/Plack/Middleware/S
    [...]
```


Friday, June 7, 13

So here we can see that we called Foo->new with arguments 'bar' and some array, on line 13 of our code. This is the place we should start tracking down how that function got its incorrect data.

Stack Traces

Friday, June 7, 13

Keep in mind that the information in stack traces is really just a "best effort" kind of deal. Perl doesn't keep copies of everything on the stack (since that would be quite inefficient), so if you're passing around a value that changes, everything in the stack that points to that value will be changed. This frequently comes up when you see 'undef' being used as the first argument to a method - this usually means that the object has been overwritten or gone out of scope since that method was called. Reblessing an object can have a similar effect. Also, deleting functions or packages while they are still on the call stack can end up confusing Carp, although this is less common.

Stack Traces

- the information in stack traces isn't guaranteed to be correct

Friday, June 7, 13

Keep in mind that the information in stack traces is really just a "best effort" kind of deal. Perl doesn't keep copies of everything on the stack (since that would be quite inefficient), so if you're passing around a value that changes, everything in the stack that points to that value will be changed. This frequently comes up when you see 'undef' being used as the first argument to a method - this usually means that the object has been overwritten or gone out of scope since that method was called. Reblessing an object can have a similar effect. Also, deleting functions or packages while they are still on the call stack can end up confusing Carp, although this is less common.

Stack Traces

- the information in stack traces isn't guaranteed to be correct
  - the stack doesn't keep copies of arguments, so they may be incorrect

Friday, June 7, 13

Keep in mind that the information in stack traces is really just a "best effort" kind of deal. Perl doesn't keep copies of everything on the stack (since that would be quite inefficient), so if you're passing around a value that changes, everything in the stack that points to that value will be changed. This frequently comes up when you see 'undef' being used as the first argument to a method - this usually means that the object has been overwritten or gone out of scope since that method was called. Reblessing an object can have a similar effect. Also, deleting functions or packages while they are still on the call stack can end up confusing Carp, although this is less common.

Stack Traces

- the information in stack traces isn't guaranteed to be correct
  - the stack doesn't keep copies of arguments, so they may be incorrect
  - deleting functions or packages can confuse things

Friday, June 7, 13

Keep in mind that the information in stack traces is really just a "best effort" kind of deal. Perl doesn't keep copies of everything on the stack (since that would be quite inefficient), so if you're passing around a value that changes, everything in the stack that points to that value will be changed. This frequently comes up when you see 'undef' being used as the first argument to a method - this usually means that the object has been overwritten or gone out of scope since that method was called. Reblessing an object can have a similar effect. Also, deleting functions or packages while they are still on the call stack can end up confusing Carp, although this is less common.

Moose Extensions

Friday, June 7, 13

There are several extensions on CPAN which provide additional ways to generate exceptions with Moose. MooseX::StrictConstructor is a common one. By default, Moose just ignores any arguments that are passed to the constructor which don't correspond to an attribute. With MooseX::StrictConstructor, however, those are turned into exceptions.

Moose Extensions

- MooseX::StrictConstructor

Friday, June 7, 13

There are several extensions on CPAN which provide additional ways to generate exceptions with Moose. MooseX::StrictConstructor is a common one. By default, Moose just ignores any arguments that are passed to the constructor which don't correspond to an attribute. With MooseX::StrictConstructor, however, those are turned into exceptions.

Moose Extensions

- MooseX::StrictConstructor
  - throws an error if unknown parameters are passed to the constructor

Friday, June 7, 13

There are several extensions on CPAN which provide additional ways to generate exceptions with Moose. MooseX::StrictConstructor is a common one. By default, Moose just ignores any arguments that are passed to the constructor which don't correspond to an attribute. With MooseX::StrictConstructor, however, those are turned into exceptions.

Moose Extensions

```
package Foo;
use Moose;
use MooseX::StrictConstructor;

has bar => (
    is  => 'ro',
    isa => 'Str',
);

Foo->new(bar => "BAR"); # fine
Foo->new(baz => "BAZ"); # dies
```

Moose Extensions

Friday, June 7, 13

MooseX::Params::Validate doesn't exactly cause Moose to throw new errors, but it is commonly used with Moose to provide parameter checking for methods, using Moose type constraints.

Moose Extensions

- MooseX::Params::Validate

Friday, June 7, 13

MooseX::Params::Validate doesn't exactly cause Moose to throw new errors, but it is commonly used with Moose to provide parameter checking for methods, using Moose type constraints.

Moose Extensions

- MooseX::Params::Validate
  - basic method parameter validation

Friday, June 7, 13

MooseX::Params::Validate doesn't exactly cause Moose to throw new errors, but it is commonly used with Moose to provide parameter checking for methods, using Moose type constraints.

Moose Extensions

```
package Foo;
use Moose;
use MooseX::Params::Validate;

sub bar {
    my $self = shift;
    my ($baz, $quux) = validated_list(
        \@_,
        baz  => { isa => 'Baz' },
        quux => { isa => 'Int', optional => 1 },
    );
    # ...
}

Foo->new->bar(baz => Baz->new, quux => 100);
```

Questions??

https://metacpan.org/module/Throwable
https://metacpan.org/module/HTTP::Throwable
https://metacpan.org/module/Try::Tiny
https://metacpan.org/module/MooseX::StrictConstructor
https://metacpan.org/module/MooseX::Params::Validate