NAME

Type::Tiny::Manual::Libraries – how to build a type library with Type::Tiny, Type::Library and Type::Utils

SYNOPSIS

A type library is a collection of type constraints, optionally with coercions.

The following is an example type library:

```
package Example::Types;

use Type::Library
   -base,
   -declare => qw( Datetime DatetimeHash EpochHash );
use Type::Utils -all;
use Types::Standard -types;

class_type Datetime, { class => "DateTime" };

declare DatetimeHash,
   as Dict[
      year       => Int,
      month      => Optional[ Int ],
      day        => Optional[ Int ],
      hour       => Optional[ Int ],
      minute     => Optional[ Int ],
      second     => Optional[ Int ],
      nanosecond => Optional[ Int ],
      time_zone  => Optional[ Str ],
   ];

declare EpochHash,
   as Dict[ epoch => Int ];

coerce Datetime,
   from Int,          via { "DateTime"->from_epoch(epoch => $_) },
   from Undef,        via { "DateTime"->now },
   from DatetimeHash, via { "DateTime"->new(%$_) },
   from EpochHash,    via { "DateTime"->from_epoch(%$_) };

1;
```

DESCRIPTION

Here's a line by line description of what's going on in the type library.

```
package Example::Types;
```

Type libraries are packages. It is recommended that re-usable type libraries be given a name in the `Types::*` namespace. For application-specific type libraries, assuming your application's namespace is `MyApp::*` then name the type library `MyApp::Types`, or if more than one is needed, use the `MyApp::Types::*` namespace.

```
use Type::Library
   -base,
   -declare => qw( Datetime DatetimeHash EpochHash );
```

The `-base` part is used to establish inheritance. It makes `Example::Types` a child class of `Type::Library`.

Declaring the types we're going to define ahead of their definition allows us to use them as barewords later on. (Note that in code which *uses* our type library, the types will always be available as barewords. The

declaration above just allows us to use them within the library itself.)

```
use Type::Utils -all;
```

Imports some utility functions from Type::Utils. These will be useful for defining our types and the relationships between them.

```
use Types::Standard -types;
```

Here we import a standard set of type constraints from Types::Standard. There is no need to do this, but it's often helpful to have a base set of types which we can define our own in terms of.

Note that although we've imported the types to be able to use in our library, we haven't *added* the types to our library. We've imported `Str`, but other people won't be able to re-import `Str` from our library. If you actually want your library to *extend* another library, do this instead:

```
BEGIN { extends "Types::AnotherLibrary" };
```

(Note: if your code breaks here when you upgrade from version 0.006 or below, saying that the 'extends' keyword has not been declared, just add '-all' after use Type::Utils.)

OK, now we're ready to declare a few types.

```
class_type Datetime, { class => "DateTime" };
```

This creates a type constraint named "Datetime" which is all objects blessed into the DateTime package. Because this type constraint is not anonymous (it has a name), it will be automagically installed into the type library.

The next two statements declare two further types constraints, using type constraints from the Types::Standard library. Let's look at `EpochHash` in more detail. This is a hashref with one key called "epoch" and a value which is an integer.

```
declare EpochHash,
   as Dict[ epoch => Int ];
```

`EpochHash` inherits from the `Dict` type defined in Types::Standard. It equally could have been defined as:

```
declare EpochHash,
   as HashRef[Int],
   where { scalar(keys(%$_))==1 and exists $_->{epoch} };
```

Or even:

```
declare EpochHash,
   where {
          ref($_) eq "HASH"
      and scalar(keys(%$_))==1
      and exists $_->{epoch}
   };
```

Lastly we set up coercions. It's best to define all your types before you define any coercions.

```
coerce Datetime,
   from Int,          via { "DateTime"->from_epoch(epoch => $_) },
   from Undef,        via { "DateTime"->now },
   from DatetimeHash, via { "DateTime"->new(%$_) },
   from EpochHash,    via { "DateTime"->from_epoch(%$_) };
```

These are simply coderefs that will be fired when you want a Datetime, but are given something else. For more information on coercions, see Type::Tiny::Manual::Coercions.

Using Your Library

Use a custom types library just like you would Types::Standard:

```
package MyClass;
use Moose;
use DateTime;
use Example::Types qw( Datetime ); # import the custom type

has 'sometime' => (
  is      => 'rw',
  isa     => Datetime,
  coerce  => 1,
);
```

Type libraries defined with Type::Library are also able to export some convenience functions:

```
use Example::Types qw( is_Datetime to_Datetime assert_Datetime );

my $dt = Foo::get_datetime;

unless ( is_Datetime $dt )
{
   $dt = to_Datetime $dt;
}

assert_Datetime $dt;
```

These functions act as shortcuts for:

```
use Example::Types qw( Datetime );

my $dt = Foo::get_datetime;

unless ( Datetime->check($dt) )
{
   $dt = Datetime->coerce($dt);
}

Datetime->assert_return($dt);
```

Pick whichever style you think is clearer!

Type::Library–based libraries provide a shortcut for importing a type constraint along with all its associated convenience functions:

```
# Shortcut for qw( DateTime is_Datetime to_Datetime assert_Datetime )
#
use Example::Types qw( +Datetime );
```

See Type::Tiny::Manual for other ways to make use of type libraries.

ADVANCED TOPICS

Messages

It is sometimes nice to be able to emit a more useful error message than the standard:

```
Value "Foo" did not pass type constraint "Bar"
```

It is possible to define custom error messages for types.

```
declare MediumInteger, as Integer,
   where   { $_ >= 10 and $_ < 20 },
   message {
      return Integer->get_message($_) if !Integer->check($_);
      return "$_ is too small!"       if $_ < 10;
      return "$_ is so very, very big!";
   };
```

Parameterized Constraints

Parameterized type constraints are those that can generate simple child type constraints by passing parameters to their `parameterize` method. For example, ArrayRef in Types::Standard:

```
use Types::Standard;

my $ArrayRef         = Types::Standard::ArrayRef;
my $Int              = Types::Standard::Int;
my $ArrayRef_of_Ints = $ArrayRef->parameterize($Int);
```

Type libraries provide some convenient sugar for this:

```
use Types::Standard qw( ArrayRef Int );

my $ArrayRef_of_Ints = ArrayRef[Int];
```

Unlike Moose which has separate meta classes for parameterizable, parameterized and non-parameterizable type constraints, Type::Tiny handles all that in one.

To create a parameterizable type constraint, you'll need to pass an extra named parameter to `declare`. Let's imagine that we want to make our earlier `NonEmptyHash` constraint accept a parameter telling it the minimum size of the hash. For example `NonEmptyHash[4]` would need to contain at least four key-value pairs. Here's how you'd do it:

```
declare NonEmptyHash, as HashLike,
   where     { scalar values %$_ },
   inline_as {
      my ($constraint, $varname) = @_;
      return sprintf(
         '%s and scalar values %%{%s}',
         $constraint->parent->inline_check($varname),
         $varname,
      );
   },
   # Generate a new "where" coderef...
   constraint_generator => sub {
      my ($minimum) = @_;
      die "parameter must be positive" unless int($minimum) > 0;
      return sub {
          scalar(values(%$_)) >= int($minimum);
      };
   },
   # Generate a new "inline_as" coderef...
   inline_generator => sub {
      my ($minimum) = @_;
      return sub {
         my ($constraint, $varname) = @_;
         return sprintf(
            '%s and scalar(values(%%{%s})) >= %d',
            $constraint->parent->inline_check($varname),
            $varname,
```

```
                $minimum,
            );
        };
    };
```

SEE ALSO

Some type libraries on CPAN:

- Types::Standard
- Types::Path::Tiny
- Types::XSD / Types::XSD::Lite
- Types::Set
- more <https://github.com/tobyink/p5-type-tiny/wiki/Type-libraries>!

AUTHOR

Toby Inkster <tobyink@cpan.org>.

COPYRIGHT AND LICENCE

DISCLAIMER OF WARRANTIES