NAME

Type::Tiny::Manual::UsingWithOther – using Type::Tiny with Class::InsideOut, Params::Check, and Object::Accessor.

MANUAL

The antlers crew aren't the only object-oriented programming toolkits in Perl town. Although Type::Tiny might have been built with Moose, Mouse, and Moo in mind, it can be used with other toolkits.

These toolkits are... well... hmm... okay... they exist.

If you are starting a new project, there's very little reason not to use Class::Tiny, Moo, or Moose. So you're probably okay to skip this part of the fine manual and go straight to Type::Tiny::Manual::UsingWithTestMore.

Class::InsideOut

You want Class::InsideOut 1.13 or above, which has support for blessed and overloaded objects (including Type::Tiny type constraints) for the `get_hook` and `set_hook` options.

```
package Person {
  use Class::InsideOut qw( public );
  use Types::Standard qw( Str Int );
  use Types::Common::Numeric qw( PositiveInt );
  use Type::Params qw( compile );

  # Type checks are really easy.
  # Just supply the type as a set hook.
  public name => my %_name, {
    set_hook => Str,
  };

  # Define a type that silently coerces negative values
  # to positive. It's silly, but it works as an example!
  my $Years = PositiveInt->plus_coercions(Int, q{ abs($_) });

  # Coercions are more annoying, but possible.
  public age => my %_age, {
    set_hook => sub { $_ = $Years->assert_coerce($_) },
  };

  # Parameter checking for methods is as expected.
  sub get_older {
    state $check = compile( $Years );
    my $self = shift;
    my ($years) = $check->(@_);
    $self->_set_age($self->age + $years);
  }
}
```

Params::Check and Object::Accessor

The Params::Check `allow()` function, the `allow` option for the Params::Check `check()` function, and the input validation mechanism for Object::Accessor all work in the same way, which is basically a limited pure-Perl implementation of the smart match operator. While this doesn't directly support Type::Tiny constraints, it does support coderefs. You can use Type::Tiny's `compiled_check` method to obtain a suitable coderef.

Param::Check example:

```
my $tmpl = {
  name => { allow => Str->compiled_check },
  age  => { allow => Int->compiled_check },
};
check($tmpl, { name => "Bob", age => 32 })
  or die Params::Check::last_error();
```

Object::Accessor example:

```
my $obj = Object::Accessor->new;
$obj->mk_accessors(
  { name => Str->compiled_check },
  { age  => Int->compiled_check },
);
```

Caveat: Object::Accessor doesn't die when a value fails to meet its type constraint; instead it outputs a warning to STDERR. This behaviour can be changed by setting `$Object::Accessor::FATAL = 1`.

Class::Struct

This is proof-of-concept of how Type::Tiny can be used to constrain attributes for Class::Struct. It's probably not a good idea to use this in production as it slows down `UNIVERSAL::isa` globally.

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

{
  my %MAP;
  my $orig_isa = \&UNIVERSAL::isa;
  *UNIVERSAL::isa = sub {
    return $MAP{$1}->check($_[0])
      if $_[1] =~ /^CLASSSTRUCT::TYPETINY::(.+)$/ && exists $MAP{$1};
    goto $orig;
  };
  my $orig_dn = \&Type::Tiny::display_name;
  *Type::Tiny::display_name = sub {
    if (caller(1) eq 'Class::Struct') {
      $MAP{$_[0]{uniq}} = $_[0];
      return "CLASSSTRUCT::TYPETINY::".$_[0]{uniq};
    }
    goto $orig_dn;
  };
}

struct Person => [ name => Str, age => Int ];

my $bob = Person->new(
  name => "Bob",
  age  => 21,
);

$bob->name("Robert");   # okay
$bob->name([]);         # dies
```

NEXT STEPS

Here's your next step:

- Type::Tiny::Manual::UsingWithTestMore

  Type::Tiny for test suites.

AUTHOR

Toby Inkster <tobyink@cpan.org>.

COPYRIGHT AND LICENCE

DISCLAIMER OF WARRANTIES