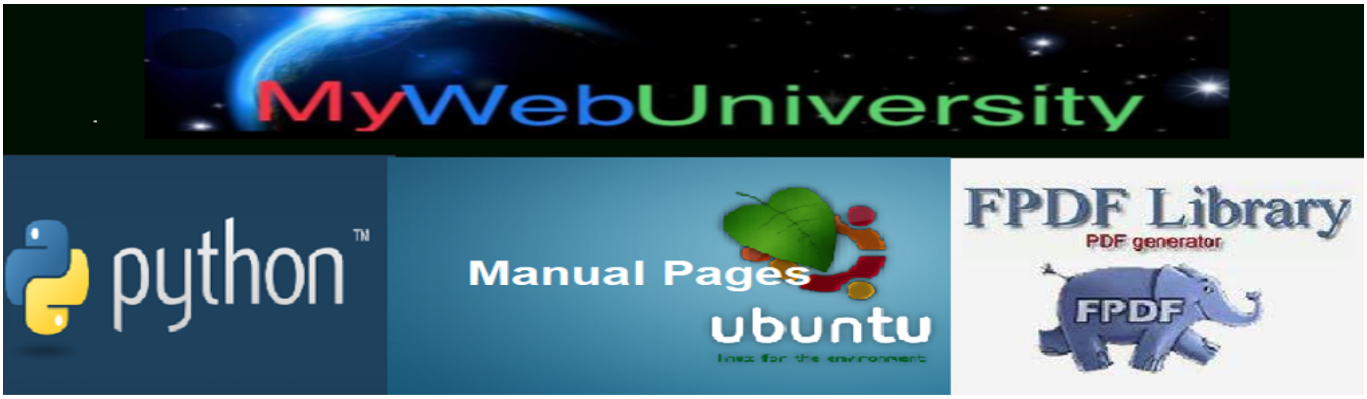

Rocky Enterprise Linux 9.2 Manual Pages on command 'Type::Tiny::Manual::UsingWithMoo3.3pm'

$ man Type::Tiny::Manual::UsingWithMoo3.3pm

Type::Tiny::Manual::UsingWithMoUserpContributed Perl DocumeType::Tiny::Manual::UsingWithMoo3(3pm)

NAME

Type::Tiny::Manual::UsingWithMoo3 - alternative use of Type::Tiny with Moo

MANUAL

Type Registries

In all the examples so far, we have imported a collection of type constraints into each class:

```
package Horse {
  use Moo;
  use Types::Standard qw( Str ArrayRef HashRef Int Any InstanceOf );
  use Types::Common::Numeric qw( PositiveInt );
  use Types::Common::String qw( NonEmptyStr );
  has name    => ( is => 'ro', isa => Str );
  has father  => ( is => 'ro', isa => InstanceOf["Horse"] );
  ...;
}
```

This creates a bunch of subs in the Horse namespace, one for each type. We've used namespace::autoclean to clean these up later.

But it is also possible to avoid pulling all these into the Horse namespace. Instead we'll use a type registry:

```
package Horse {
  use Moo;
  use Type::Registry qw( t );
```

```
    t->add_types('-Standard');
    t->add_types('-Common::String');
    t->add_types('-Common::Numeric');
    t->alias_type('InstanceOf["Horse"]' => 'Horsey');
    has name     => ( is => 'ro', isa => t('Str') );
    has father   => ( is => 'ro', isa => t('Horsey') );
    has mother   => ( is => 'ro', isa => t('Horsey') );
    has children => ( is => 'ro', isa => t('ArrayRef[Horsey]') );
    ...;
  }
```

You don't even need to import the "t()" function. Types::Registry can be used in an entirely object-oriented way.

```
  package Horse {
    use Moo;
    use Type::Registry;
    my $reg = Type::Registry->for_me;
    $reg->add_types('-Standard');
    $reg->add_types('-Common::String');
    $reg->add_types('-Common::Numeric');
    $reg->alias_type('InstanceOf["Horse"]' => 'Horsey');
    has name => ( is => 'ro', isa => $reg->lookup('Str') );
    ...;
  }
```

You could create two registries with entirely different definitions for the same named type.

```
  my $dracula = Aristocrat->new(name => 'Dracula');
  package AristocracyTracker {
    use Type::Registry;
    my $reg1 = Type::Registry->new;
    $reg1->add_types('-Common::Numeric');
    $reg1->alias_type('PositiveInt' => 'Count');
    my $reg2 = Type::Registry->new;
    $reg2->add_types('-Standard');
```

```
    $reg2->alias_type('InstanceOf["Aristocrat"]' => 'Count');
    $reg1->lookup("Count")->assert_valid("1");
    $reg2->lookup("Count")->assert_valid($dracula);
  }
```

Type::Registry uses "AUTOLOAD", so things like this work:

```
  $reg->ArrayRef->of( $reg->Int );
```

Although you can create as many registries as you like, Type::Registry will create a default registry for each package.

```
  # Create a new empty registry.
  #
  my $reg = Type::Registry->new;
  # Get the default registry for my package.
  # It will be pre-populated with any types we imported using `use`.
  #
  my $reg = Type::Registry->for_me;
  # Get the default registry for some other package.
  #
  my $reg = Type::Registry->for_class("Horse");
```

Type registries are a convenient place to store a bunch of types without polluting your namespace. They are not the same as type libraries though. Types::Standard, Types::Common::String, and Types::Common::Numeric are type libraries; packages that export types for others to use. We will look at how to make one of those later.

For now, here's the best way to think of the difference:

? Type registry

  Curate a collection of types for me to use here in this class. This collection is an implementaion detail.

? Type library

  Export a collection of types to be used across multiple classes. This collection is part of your API.

Importing Functions

We've seen how, for instance, Types::Standard exports a sub called "Int" that returns the Int type object.

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

```
  my $type = Int;
  $type->check($value) or die $type->get_message($value);
```

Type libraries are also capable of exporting other convenience functions.

"is_*"

This is a shortcut for checking a value meets a type constraint:

```
  use Types::Standard qw( is_Int );
  if ( is_Int($value) ) {
    ...;
  }
```

Calling "is_Int($value)" will often be marginally faster than calling "Int->check($value)" because it avoids a method call. (Method calls in Perl end up slower than normal function calls.)

Using things like "is_ArrayRef" in your code might be preferable to "ref($value) eq "ARRAY"" because it's neater, leads to more consistent type checking, and might even be faster. (Type::Tiny can be pretty fast; it is sometimes able to export these functions as XS subs.)

If checking type constraints like "is_ArrayRef" or "is_InstanceOf", there's no way to give a parameter. "is_ArrayRef[Int]($value)" doesn't work, and neither does "is_ArrayRef(Int, $value)" nor "is_ArrayRef($value, Int)". For some types like "is_InstanceOf", this makes them fairly useless; without being able to give a class name, it just acts the same as "is_Object". See "Exporting Parameterized Types" for a solution. Also, check out isa.

There also exists a generic "is" function.

```
  use Types::Standard qw( ArrayRef Int );
  use Type::Utils qw( is );
  if ( is ArrayRef[Int], \@numbers ) {
    ...;
  }
```

"assert_*"

While "is_Int($value)" returns a boolean, "assert_Int($value)" will throw an error if the value does not meet the constraint, and return the value otherwise. So you can do:

```
  my $sum = assert_Int($x) + assert_Int($y);
```

And you will get the sum of integers $x and $y, and an explosion if either of them is not an integer!

Assert is useful for quick parameter checks if you are avoiding Type::Params for some strange reason:

```
  sub add_numbers {
    my $x = assert_Num(shift);
    my $y = assert_Num(shift);
    return $x + $y;
  }
```

You can also use a generic "assert" function.

```
  use Type::Utils qw( assert );
  sub add_numbers {
    my $x = assert Num, shift;
    my $y = assert Num, shift;
    return $x + $y;
  }
```

"to_*"

This is a shortcut for coercion:

```
  my $truthy = to_Bool($value);
```

It trusts that the coercion has worked okay. You can combine it with an assertion if you want to make sure.

```
  my $truthy = assert_Bool(to_Bool($value));
```

Shortcuts for exporting functions

This is a little verbose:

```
  use Types::Standard qw( Bool is_Bool assert_Bool to_Bool );
```

Isn't this a little bit nicer?

```
  use Types::Standard qw( +Bool );
```

The plus sign tells a type library to export not only the type itself, but all of the convenience functions too.

You can also use:

```
  use Types::Standard -types;   # export Int, Bool, etc
  use Types::Standard -is;      # export is_Int, is_Bool, etc
  use Types::Standard -assert;  # export assert_Int, assert_Bool, etc
  use Types::Standard -to;      # export to_Bool, etc
  use Types::Standard -all;     # just export everything!!!
```

So if you imagine the functions exported by Types::Standard are like this:

```
qw(
  Str             is_Str          assert_Str
  Num             is_Num          assert_Num
  Int             is_Int          assert_Int
  Bool            is_Bool         assert_Bool     to_Bool
  ArrayRef        is_ArrayRef     assert_ArrayRef
);
# ... and more
```

Then "+" exports a horizonal group of those, and "-" exports a vertical group.

Exporting Parameterized Types

It's possible to export parameterizable types like ArrayRef, but it is also possible to export parameterized types.

```
use Types::Standard qw( ArrayRef Int );
use Types::Standard (
  '+ArrayRef' => { of => Int, -as => 'IntList' },
);
has numbers => (is => 'ro', isa => IntList);
```

Using "is_IntList($value)" should be significantly faster than "ArrayRef->of(Int)->check($value)".

This trick only works for parameterized types that have a single parameter, like ArrayRef, HashRef, InstanceOf, etc. (Sorry, "Dict" and "Tuple"!)

Do What I Mean!

```
use Type::Utils qw( dwim_type );
dwim_type("ArrayRef[Int]")
```

"dwim_type" will look up a type constraint from a string and attempt to guess what you meant.

If it's a type constraint that you seem to have imported with "use", then it should find it. Otherwise, if you're using Moose or Mouse, it'll try asking those. Or if it's in Types::Standard, it'll look there. And if it still has no idea, then it will assume dwim_type("Foo") means dwim_type("InstanceOf['Foo']").

It just does a big old bunch of guessing.

The "is" function will use "dwim_type" if you pass it a string as a type.

```
use Type::Utils qw( is );
if ( is "ArrayRef[Int]", \@numbers ) {
  ...;
}
```

NEXT STEPS

You now know pretty much everything there is to know about how to use type libraries.

Here's your next step:

? Type::Tiny::Manual::Libraries

Defining your own type libraries, including extending existing libraries, defining new types, adding coercions, defining parameterizable types, and the declarative style.

AUTHOR

Toby Inkster <tobyink@cpan.org>.

COPYRIGHT AND LICENCE

DISCLAIMER OF WARRANTIES