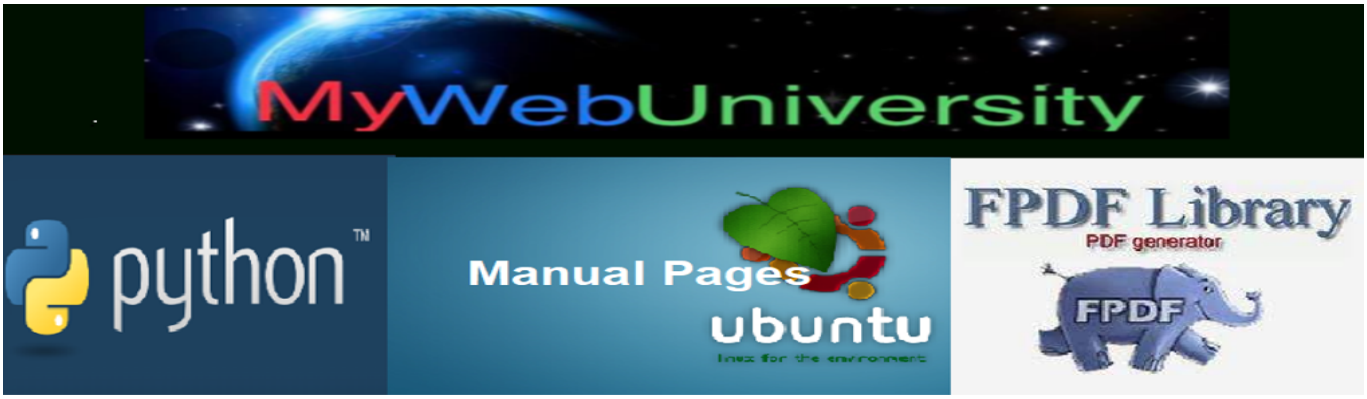

Rocky Enterprise Linux 9.2 Manual Pages on command 'Type::Tiny::Manual::UsingWithMoo2.3pm'

$ man Type::Tiny::Manual::UsingWithMoo2.3pm

Type::Tiny::Manual::UsingWithMoUserpContributed Perl DocumeType::Tiny::Manual::UsingWithMoo2(3pm)

NAME

Type::Tiny::Manual::UsingWithMoo2 - advanced use of Type::Tiny with Moo

MANUAL

What is a Type?

So far all the examples have shown you how to work with types, but we haven't looked at what a type actually is.

```
use Types::Standard qw( Int );
my $type = Int;
```

"Int" in the above code is just a function called with zero arguments which returns a blessed Perl object. It is this object that defines what the Int type is and is responsible for checking values meet its definition.

```
use Types::Standard qw( HashRef Int );
my $type = HashRef[Int];
```

The "HashRef" function, if called with no parameters returns the object defining the HashRef type, just like the "Int" function did before. But the difference here is that it's called with a parameter, an arrayref containing the Int type object. It uses this to make the HashRef[Int] type and returns that.

Like any object, you can call methods on it. The most important methods to know about are:

```
# check the value and return a boolean
#
$type->check($value);
# return an error message about $value failing the type check
```

```
  # but don't actually check the value
  #
  $type->get_message($value);
  # coerce the value
  #
  my $coerced = $type->coerce($value);
```

We've already seen some other methods earlier in the tutorial.

```
  # create a new type, same as the old type, but that has coercions
  #
  my $new_type = $type->plus_coercions( ... );
  # different syntax for parameterized types
  #
  my $href = HashRef;
  my $int = Int;
  my $href_of_int = $href->of($int);
```

So now you should understand this:

```
  use Types::Standard qw( ArrayRef Dict Optional );
  use Types::Common::Numeric qw( PositiveInt );
  use Types::Common::String qw( NonEmptyStr );
  my $RaceInfo = Dict[
    year    => PositiveInt,
    race    => NonEmptyStr,
    jockey  => Optional[NonEmptyStr],
  ];
  has latest_event  => ( is => 'rw', isa => $RaceInfo );
  has wins          => ( is => 'rw', isa => ArrayRef[$RaceInfo] );
  has losses        => ( is => 'rw', isa => ArrayRef[$RaceInfo] );
```

This can help you avoid repetition if you have a complex parameterized type that you need to reuse a few times.

"where"

One of the most useful methods you can call on a type object is "where".

```
  use Types::Standard qw( Int );
  has lucky_number => (
```

```
  is   => 'ro',
  isa  => Int->where(sub { $_ != 13 }),
);
```

I think you already understand what it does. It creates a new type constraint on the fly, restricting the original type.

Like with coercions, these restrictions can be expressed as a coderef or as a string of Perl code, operating on the $_ variable. And like with coercions, using a string of code will result in better performance.

```
use Types::Standard qw( Int );
has lucky_number => (
  is   => 'ro',
  isa  => Int->where(q{ $_ != 13 }),
);
```

Let's coerce a hashref of strings from an even-sized arrayref of strings:

```
use Types::Standard qw( HashRef ArrayRef Str );
has stringhash => (
  is   => 'ro',
  isa  => HashRef->of(Str)->plus_coercions(
    ArrayRef->of(Str)->where(q{ @$_ % 2 == 0 }), q{
      my %h = @$_;
      \%h;
    },
  ),
  coerce => 1,  # never forget!
);
```

If you understand that, you really are in the advanced class. Congratulations!

Unions

Sometimes you want to accept one thing or another thing. This is pretty easy with Type::Tiny.

```
use Types::Standard qw( HashRef ArrayRef Str );
has strings => (
  is    => 'ro',
  isa   => ArrayRef[Str] | HashRef[Str],
```

```
  );
```

Type::Tiny overloads the bitwise or operator so stuff like this should "just work".

That said, now any code that calls "$self->strings" will probably need to check if the value is an arrayref or a hashref before doing anything with it. So it may be simpler overall if you just choose one of the options and coerce the other one into it.

Intersections

Similar to a union is an intersection.

```
  package MyAPI::Client {
    use Moo;
    use Types::Standard qw( HasMethods InstanceOf );
    has ua => (
      is    => 'ro',
      isa   => (InstanceOf["MyUA"]) & (HasMethods["store_cookie"]),
    );
  }
```

Here we are checking that the UA is an instance of the MyUA class and also offers the "store_cookie" method. Perhaps "store_cookie" isn't provided by the MyUA class itself, but several subclasses of MyUA provide it.

Intersections are not useful as often as unions are. This is because they often make no sense. "(ArrayRef) & (HashRef)" would be a reference which was simultaneously pointing to an array and a hash, which is impossible.

Note that when using intersections, it is good practice to put parentheses around each type. This is to disambiguate the meaning of "&" for Perl, because Perl uses it as the bitwise and operator but also as the sigil for subs.

Complements

For any type Foo there is a complementary type ~Foo (pronounced "not Foo").

```
  package My::Class {
    use Moo;
    use Types::Standard qw( ArrayRef CodeRef );
    has things => ( is => 'ro', isa => ArrayRef[~CodeRef] );
  }
```

"things" is now an arrayref of anything except coderefs.

If you need a number that is not an integer:

```
  Num & ~Int
```

Types::Standard includes two types which are complements of each other: Undef and Defined.

NegativeInt might seem to be the complement of PositiveOrZeroInt but when you think about it, it is not. There are values that fall into neither category, such as non-integers, non-numeric strings, references, undef, etc.

"stringifies_to" and "numifies_to"

The Object type constraint provides "stringifies_to" and "numifies_to" methods which are probably best explained by examples.

"Object->numifies_to(Int)" means any object where "0 + $object" is an integer.

"Object->stringifies_to(StrMatch[$re])" means any object where "$object" matches the regular expression.

"Object->stringifies_to($re)" also works as a shortcut.

"Object->numifies_to($coderef)" and "Object->stringifies_to($coderef)" also work, where the coderef checks $_ and returns a boolean.

Other types which are also logically objects, such as parameterized HasMethods, InstanceOf, and ConsumerOf should also provide "stringifies_to" and "numifies_to" methods.

"stringifies_to" and "numifies_to" work on unions if all of the type constraints in the union offer the method.

"stringifies_to" and "numifies_to" work on intersections if at least one of of the type constraints in the intersection offers the method.

"with_attribute_values"

Another one that is probably best explained using an example:

```
  package Horse {
    use Types::Standard qw( Enum Object );
    has gender  => (
      is  => 'ro',
      isa => Enum['m', 'f'],
    );
    has father  => (
      is  => 'ro',
      isa => Object->with_attribute_values(gender => Enum['m']),
    );
    has mother  => (
```

```
    is  => 'ro',
    isa => Object->with_attribute_values(gender => Enum['f']),
  );
}
```

In this example when you set a horse's father, it will call "$father->gender" and check that it matches Enum['m'].

This method is in the same family as "stringifies_as" and "numifies_as", so like those, it only applies to Object and similar type constraints, can work on unions/intersections under the same circumstances, and will also accept coderefs and regexps.

```
has father  => (
  is  => 'ro',
  isa => Object->with_attribute_values(gender => sub { $_ eq 'm' }),
);
has mother  => (
  is  => 'ro',
  isa => Object->with_attribute_values(gender => qr/^f/i),
);
```

All of "stringifies_as", "numifies_as", and "with_attributes_as" are really just wrappers around "where". The following two are roughly equivalent:

```
my $type1 = Object->with_attribute_values(foo => Int, bar => Num);
my $type2 = Object->where(sub {
  Int->check( $_->foo ) and Num->check( $_->bar )
});
```

The first will result in better performing code though.

Tied Variables

It is possible to tie variables to a type constraint.

```
use Types::Standard qw(Int);
tie my $n, Int, 4;
print "$n\n";   # says "4"
$n = 5;         # ok
$n = "foo";     # dies
```

This feature requires Type::Tie which is a separate thing to install. Type::Tiny will automatically load Type::Tie in the background if it detects you're trying to tie a

variable to a type.

You can also tie arrays:

```
tie my @numbers, Int;
push @numbers, 1 .. 10;
```

And hashes:

```
tie my %numbers, Int;
$numbers{lucky}   = 7;
$numbers{unlucky} = 13;
```

Earlier in the manual, it was mentioned that there is a problem with code like this:

```
push @{ $horse->children }, $non_horse;
```

This can be solved using tied variables.

```
tie @{ $horse->children }, InstanceOf["Horse"];
```

Here is a longer example using builders and triggers.

```
package Horse {
  use Moo;
  use Types::Standard qw( Str Num ArrayRef InstanceOf );
  use Type::Params qw(compile);
  use namespace::autoclean;
  my $ThisClass = InstanceOf[ __PACKAGE__ ];
  has name       => ( is => 'ro',    isa => Str );
  has gender     => ( is => 'ro',    isa => Str );
  has age        => ( is => 'rw',    isa => Num );
  has children   => (
    is        => 'rw',
    isa       => ArrayRef[$ThisClass],
    builder   => "_build_children",
    trigger   => sub { shift->_trigger_children(@_) },
  );
  # tie a default arrayref
  sub _build_children {
    my $self = shift;
    tie my @kids, $ThisClass;
    \@kids;
```

```
    }
    # this method will tie an arrayref provided by the caller
    sub _trigger_children {
      my $self = shift;
      my ($new) = @_;
      tie @$new, $ThisClass;
    }
    sub add_child {
      state $check = compile($ThisClass, $ThisClass);
      my ($self, $kid) = &$check;
      push @{ $self->children }, $kid;
      return $self;
    }
  }
```

Now it's pretty much impossible for the caller to make a mess by adding a non-horse as a child.

NEXT STEPS

Here's your next step:

? Type::Tiny::Manual::UsingWithMoo3

There's more than one way to do it! Alternative ways of using Type::Tiny, including type registries, exported functions, and "dwim_type".

AUTHOR

Toby Inkster <tobyink@cpan.org>.

COPYRIGHT AND LICENCE

DISCLAIMER OF WARRANTIES