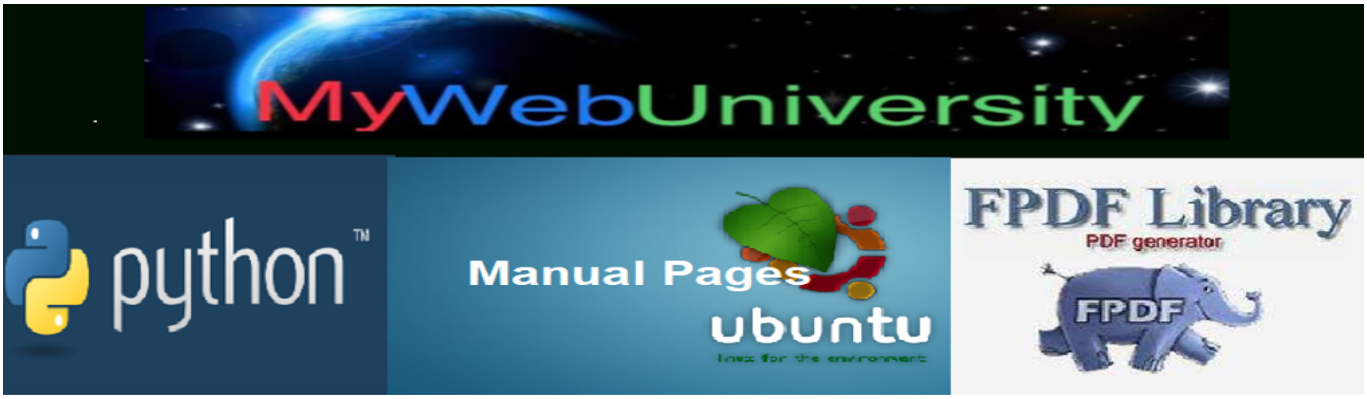

Rocky Enterprise Linux 9.2 Manual Pages on command 'Type::Tiny.3pm'

$ man Type::Tiny.3pm

Type::Tiny(3pm) User Contributed Perl Documentation Type::Tiny(3pm)

NAME

Type::Tiny - tiny, yet Moo(se)-compatible type constraint

SYNOPSIS

```
use v5.12;
use strict;
use warnings;
package Horse {
  use Moo;
  use Types::Standard qw( Str Int Enum ArrayRef Object );
  use Type::Params qw( compile );
  use namespace::autoclean;
  has name => (
    is       => 'ro',
    isa      => Str,
    required => 1,
  );
  has gender => (
    is       => 'ro',
    isa      => Enum[qw( f m )],
  );
  has age => (
    is       => 'rw',
```

```
    isa      => Int->where( '$_ >= 0' ),
  );
  has children => (
    is       => 'ro',
    isa      => ArrayRef[Object],
    default  => sub { return [] },
  );
  sub add_child {
    state $check = compile( Object, Object );  # method signature
    my ($self, $child) = $check->(@_);         # unpack @_
    push @{ $self->children }, $child;
    return $self;
  }
}
package main;
my $boldruler = Horse->new(
  name    => "Bold Ruler",
  gender  => 'm',
  age     => 16,
);
my $secretariat = Horse->new(
  name    => "Secretariat",
  gender  => 'm',
  age     => 0,
);
$boldruler->add_child( $secretariat );
```

STATUS

This module is covered by the Type-Tiny stability policy.

DESCRIPTION

This documents the internals of the Type::Tiny class. Type::Tiny::Manual is a better starting place if you're new.

Type::Tiny is a small class for creating Moose-like type constraint objects which are compatible with Moo, Moose and Mouse.

```
use Scalar::Util qw(looks_like_number);
use Type::Tiny;
my $NUM = "Type::Tiny"->new(
  name       => "Number",
  constraint => sub { looks_like_number($_) },
  message    => sub { "$_ ain't a number" },
);
package Ermintrude {
  use Moo;
  has favourite_number => (is => "ro", isa => $NUM);
}
package Bullwinkle {
  use Moose;
  has favourite_number => (is => "ro", isa => $NUM);
}
package Maisy {
  use Mouse;
  has favourite_number => (is => "ro", isa => $NUM);
}
```

Maybe now we won't need to have separate MooseX, MouseX and MooX versions of everything?

We can but hope...

Constructor

"new(%attributes)"

Moose-style constructor function.

Attributes

Attributes are named values that may be passed to the constructor. For each attribute, there is a corresponding reader method. For example:

```
my $type = Type::Tiny->new( name => "Foo" );
print $type->name, "\n";   # says "Foo"
```

Important attributes

These are the attributes you are likely to be most interested in providing when creating your own type constraints, and most interested in reading when dealing with type constraint objects.

"constraint"

Coderef to validate a value ($_) against the type constraint. The coderef will not be called unless the value is known to pass any parent type constraint (see "parent" below).

Alternatively, a string of Perl code checking $_ can be passed as a parameter to the constructor, and will be converted to a coderef.

Defaults to "sub { 1 }" - i.e. a coderef that passes all values.

"parent"

Optional attribute; parent type constraint. For example, an "Integer" type constraint might have a parent "Number".

If provided, must be a Type::Tiny object.

"inlined"

A coderef which returns a string of Perl code suitable for inlining this type.

Optional.

(The coderef will be called in list context and can actually return a list of strings which will be joined with "&&". If the first item on the list is undef, it will be substituted with the type's parent's inline check.)

If "constraint" (above) is a coderef generated via Sub::Quote, then Type::Tiny may be able to automatically generate "inlined" for you. If "constraint" (above) is a string, it will be able to.

"name"

The name of the type constraint. These need to conform to certain naming rules (they must begin with an uppercase letter and continue using only letters, digits 0-9 and underscores).

Optional; if not supplied will be an anonymous type constraint.

"display_name"

A name to display for the type constraint when stringified. These don't have to conform to any naming rules. Optional; a default name will be calculated from the "name".

"library"

The package name of the type library this type is associated with. Optional.

Informational only: setting this attribute does not install the type into the package.

"deprecated"

Optional boolean indicating whether a type constraint is deprecated. Type::Library will issue a warning if you attempt to import a deprecated type constraint, but otherwise the type will continue to function as normal. There will not be deprecation warnings every time you validate a value, for instance. If omitted, defaults to the parent's deprecation status (or false if there's no parent).

"message"

Coderef that returns an error message when $_ does not validate against the type constraint. Optional (there's a vaguely sensible default.)

"coercion"

A Type::Coercion object associated with this type.

Generally speaking this attribute should not be passed to the constructor; you should rely on the default lazily-built coercion object.

You may pass "coercion => 1" to the constructor to inherit coercions from the constraint's parent. (This requires the parent constraint to have a coercion.)

"sorter"

A coderef which can be passed two values conforming to this type constraint and returns -1, 0, or 1 to put them in order. Alternatively an arrayref containing a pair of coderefs ? a sorter and a pre-processor for the Schwarzian transform. Optional.

The idea is to allow for:

```
@sorted = Int->sort( 2, 1, 11 );    # => 1, 2, 11
@sorted = Str->sort( 2, 1, 11 );    # => 1, 11, 2
```

"my_methods"

Experimental hashref of additional methods that can be called on the type constraint object.

Attributes related to parameterizable and parameterized types

The following additional attributes are used for parameterizable (e.g. "ArrayRef") and parameterized (e.g. "ArrayRef[Int]") type constraints. Unlike Moose, these aren't handled by separate subclasses.

"constraint_generator"

Coderef that is called when a type constraint is parameterized. When called, it is passed the list of parameters, though any parameter which looks like a foreign type constraint (Moose type constraints, Mouse type constraints, etc, and coderefs(!!!)) is first coerced to a native Type::Tiny object.

Note that for compatibility with the Moose API, the base type is not passed to the constraint generator, but can be found in the package variable $Type::Tiny::parameterize_type. The first parameter is also available as $_.

Types can be parameterized with an empty parameter list. For example, in Types::Standard, "Tuple" is just an alias for "ArrayRef" but "Tuple[]" will only allow zero-length arrayrefs to pass the constraint. If you wish "YourType" and "YourType[]" to mean the same thing, then do:

```
return $Type::Tiny::parameterize_type unless @_;
```

The constraint generator should generate and return a new constraint coderef based on the parameters. Alternatively, the constraint generator can return a fully-formed Type::Tiny object, in which case the "name_generator", "inline_generator", and "coercion_generator" attributes documented below are ignored.

Optional; providing a generator makes this type into a parameterizable type constraint. If there is no generator, attempting to parameterize the type constraint will throw an exception.

"name_generator"

A coderef which generates a new display_name based on parameters. Called with the same parameters and package variables as the "constraint_generator". Expected to return a string.

Optional; the default is reasonable.

"inline_generator"

A coderef which generates a new inlining coderef based on parameters. Called with the same parameters and package variables as the "constraint_generator". Expected to return a coderef.

Optional.

"coercion_generator"

A coderef which generates a new Type::Coercion object based on parameters. Called with the same parameters and package variables as the "constraint_generator". Expected to return a blessed object.

Optional.

"deep_explanation"

This API is not finalized. Coderef used by Error::TypeTiny::Assertion to peek inside parameterized types and figure out why a value doesn't pass the constraint.

"parameters"

In parameterized types, returns an arrayref of the parameters.

Lazy generated attributes

The following attributes should not be usually passed to the constructor; unless you're doing something especially unusual, you should rely on the default lazily-built return values.

"compiled_check"

Coderef to validate a value ($_[0]) against the type constraint. This coderef is expected to also handle all validation for the parent type constraints.

"complementary_type"

A complementary type for this type. For example, the complementary type for an integer type would be all things that are not integers, including floating point numbers, but also alphabetic strings, arrayrefs, filehandles, etc.

"moose_type", "mouse_type"

Objects equivalent to this type constraint, but as a Moose::Meta::TypeConstraint or Mouse::Meta::TypeConstraint.

It should rarely be necessary to obtain a Moose::Meta::TypeConstraint object from Type::Tiny because the Type::Tiny object itself should be usable pretty much anywhere a Moose::Meta::TypeConstraint is expected.

Methods

Predicate methods

These methods return booleans indicating information about the type constraint. They are each tightly associated with a particular attribute. (See "Attributes".)

"has_parent", "has_library", "has_inlined", "has_constraint_generator", "has_inline_generator", "has_coercion_generator", "has_parameters", "has_message", "has_deep_explanation", "has_sorter"

Simple Moose-style predicate methods indicating the presence or absence of an attribute.

"has_coercion"

Predicate method with a little extra DWIM. Returns false if the coercion is a no-op.

"is_anon"

Returns true iff the type constraint does not have a "name".

"is_parameterized", "is_parameterizable"

Indicates whether a type has been parameterized (e.g. "ArrayRef[Int]") or could potentially be (e.g. "ArrayRef").

"has_parameterized_from"

Useless alias for "is_parameterized".

Validation and coercion

The following methods are used for coercing and validating values against a type constraint:

"check($value)"

Returns true iff the value passes the type constraint.

"validate($value)"

Returns the error message for the value; returns an explicit undef if the value passes the type constraint.

"assert_valid($value)"

Like "check($value)" but dies if the value does not pass the type constraint.

Yes, that's three very similar methods. Blame Moose::Meta::TypeConstraint whose API I'm attempting to emulate. :-)

"assert_return($value)"

Like "assert_valid($value)" but returns the value if it passes the type constraint.

This seems a more useful behaviour than "assert_valid($value)". I would have just changed "assert_valid($value)" to do this, except that there are edge cases where it could break Moose compatibility.

"get_message($value)"

Returns the error message for the value; even if the value passes the type constraint.

"validate_explain($value, $varname)"

Like "validate" but instead of a string error message, returns an arrayref of strings explaining the reasoning why the value does not meet the type constraint, examining parent types, etc.

The $varname is an optional string like '$foo' indicating the name of the variable being checked.

"coerce($value)"

Attempt to coerce $value to this type.

"assert_coerce($value)"

Attempt to coerce $value to this type. Throws an exception if this is not possible.

Child type constraint creation and parameterization

These methods generate new type constraint objects that inherit from the constraint they are called upon:

"create_child_type(%attributes)"

Construct a new Type::Tiny object with this object as its parent.

"where($coderef)"

Shortcut for creating an anonymous child type constraint. Use it like "HashRef->where(sub { exists($_->{name}) })". That said, you can get a similar result using overloaded "&":

```
HashRef & sub { exists($_->{name}) }
```

Like the "constraint" attribute, this will accept a string of Perl code:

```
HashRef->where('exists($_->{name})')
```

"child_type_class"

The class that create_child_type will construct by default.

"parameterize(@parameters)"

Creates a new parameterized type; throws an exception if called on a non-parameterizable type.

"of(@parameters)"

A cute alias for "parameterize". Use it like "ArrayRef->of(Int)".

"plus_coercions($type1, $code1, ...)"

Shorthand for creating a new child type constraint with the same coercions as this one, but then adding some extra coercions (at a higher priority than the existing ones).

"plus_fallback_coercions($type1, $code1, ...)"

Like "plus_coercions", but added at a lower priority.

"minus_coercions($type1, ...)"

Shorthand for creating a new child type constraint with fewer type coercions.

"no_coercions"

Shorthand for creating a new child type constraint with no coercions at all.

Type relationship introspection methods

These methods allow you to determine a type constraint's relationship to other type constraints in an organised hierarchy:

"equals($other)", "is_subtype_of($other)", "is_supertype_of($other)",

"is_a_type_of($other)"

Compare two types. See Moose::Meta::TypeConstraint for what these all mean. (OK, Moose doesn't define "is_supertype_of", but you get the idea, right?)

Note that these have a slightly DWIM side to them. If you create two Type::Tiny::Class objects which test the same class, they're considered equal. And:

```
my $subtype_of_Num = Types::Standard::Num->create_child_type;
my $subtype_of_Int = Types::Standard::Int->create_child_type;
$subtype_of_Int->is_subtype_of( $subtype_of_Num );  # true
```

"strictly_equals($other)", "is_strictly_subtype_of($other)", "is_strictly_supertype_of($other)", "is_strictly_a_type_of($other)"

Stricter versions of the type comparison functions. These only care about explicit inheritance via "parent".

```
my $subtype_of_Num = Types::Standard::Num->create_child_type;
my $subtype_of_Int = Types::Standard::Int->create_child_type;
$subtype_of_Int->is_strictly_subtype_of( $subtype_of_Num );  # false
```

"parents"

Returns a list of all this type constraint's ancestor constraints. For example, if called on the "Str" type constraint would return the list "(Value, Defined, Item, Any)".

Due to a historical misunderstanding, this differs from the Moose implementation of the "parents" method. In Moose, "parents" only returns the immediate parent type constraints, and because type constraints only have one immediate parent, this is effectively an alias for "parent". The extension module MooseX::Meta::TypeConstraint::Intersection is the only place where multiple type constraints are returned; and they are returned as an arrayref in violation of the base class' documentation. I'm keeping my behaviour as it seems more useful.

"find_parent($coderef)"

Loops through the parent type constraints including the invocant itself and returns the nearest ancestor type constraint where the coderef evaluates to true. Within the coderef the ancestor currently being checked is $_. Returns undef if there is no match.

In list context also returns the number of type constraints which had been looped through before the matching constraint was found.

"find_constraining_type"

Finds the nearest ancestor type constraint (including the type itself) which has a "constraint" coderef.

Equivalent to:

```
$type->find_parent(sub { not $_->_is_null_constraint })
```

"coercibles"

Return a type constraint which is the union of type constraints that can be coerced to this one (including this one). If this type constraint has no coercions, returns itself.

"type_parameter"

In parameterized type constraints, returns the first item on the list of parameters; otherwise returns undef. For example:

```
( ArrayRef[Int] )->type_parameter;    # returns Int
( ArrayRef[Int] )->parent;            # returns ArrayRef
```

Note that parameterizable type constraints can perfectly legitimately take multiple parameters (several of the parameterizable type constraints in Types::Standard do). This method only returns the first such parameter. "Attributes related to parameterizable and parameterized types" documents the "parameters" attribute, which returns an arrayref of all the parameters.

"parameterized_from"

Harder to spell alias for "parent" that only works for parameterized types.

Hint for people subclassing Type::Tiny: Since version 1.006000, the methods for determining subtype, supertype, and type equality should not be overridden in subclasses of Type::Tiny. This is because of the problem of diamond inheritance. If X and Y are both subclasses of Type::Tiny, they both need to be consulted to figure out how type constraints are related; not just one of them should be overriding these methods. See the source code for Type::Tiny::Enum for an example of how subclasses can give hints about type relationships to Type::Tiny. Summary: push a coderef onto @Type::Tiny::CMP. This coderef will be passed two type constraints. It should then return one of the constants Type::Tiny::CMP_SUBTYPE (first type is a subtype of second type), Type::Tiny::CMP_SUPERTYPE (second type is a subtype of first type), Type::Tiny::CMP_EQUAL (the two types are exactly the same), Type::Tiny::CMP_EQUIVALENT (the two types are effectively the same), or Type::Tiny::CMP_UNKNOWN (your coderef couldn't establish any

relationship).

Type relationship introspection function

"Type::Tiny::cmp($type1, $type2)"

The subtype/supertype relationship between types results in a partial ordering of type constraints.

This function will return one of the constants: Type::Tiny::CMP_SUBTYPE (first type is a subtype of second type), Type::Tiny::CMP_SUPERTYPE (second type is a subtype of first type), Type::Tiny::CMP_EQUAL (the two types are exactly the same), Type::Tiny::CMP_EQUIVALENT (the two types are effectively the same), or Type::Tiny::CMP_UNKNOWN (couldn't establish any relationship). In numeric contexts, these evaluate to -1, 1, 0, 0, and 0, making it potentially usable with "sort" (though you may need to silence warnings about treating the empty string as a numeric value).

List processing methods

"grep(@list)"

Filters a list to return just the items that pass the type check.

```
@integers = Int->grep(@list);
```

"first(@list)"

Filters the list to return the first item on the list that passes the type check, or undef if none do.

```
$first_lady = Woman->first(@people);
```

"map(@list)"

Coerces a list of items. Only works on types which have a coercion.

```
@truths = Bool->map(@list);
```

"sort(@list)"

Sorts a list of items according to the type's preferred sorting mechanism, or if the type doesn't have a sorter coderef, uses the parent type. If no ancestor type constraint has a sorter, throws an exception. The "Str", "StrictNum", "LaxNum", and "Enum" type constraints include sorters.

```
@sorted_numbers = Num->sort( Num->grep(@list) );
```

"rsort(@list)"

Like "sort" but backwards.

"any(@list)"

Returns true if any of the list match the type.

```
  if ( Int->any(@numbers) ) {
    say "there was at least one integer";
  }
```

"all(@list)"

Returns true if all of the list match the type.

```
  if ( Int->all(@numbers) ) {
    say "they were all integers";
  }
```

"assert_any(@list)"

Like "any" but instead of returning a boolean, returns the entire original list if any item on it matches the type, and dies if none does.

"assert_all(@list)"

Like "all" but instead of returning a boolean, returns the original list if all items on it match the type, but dies as soon as it finds one that does not.

Inlining methods

The following methods are used to generate strings of Perl code which may be pasted into stringy "eval"uated subs to perform type checks:

"can_be_inlined"

Returns boolean indicating if this type can be inlined.

"inline_check($varname)"

Creates a type constraint check for a particular variable as a string of Perl code.

For example:

```
  print( Types::Standard::Num->inline_check('$foo') );
```

prints the following output:

```
  (!ref($foo) && Scalar::Util::looks_like_number($foo))
```

For Moose-compat, there is an alias "_inline_check" for this method.

"inline_assert($varname)"

Much like "inline_check" but outputs a statement of the form:

```
  ... or die ...;
```

Can also be called line "inline_assert($varname, $typevarname, %extras)". In this case, it will generate a string of code that may include $typevarname which is supposed to be the name of a variable holding the type itself. (This is kinda complicated, but it allows a useful string to still be produced if the type is not

inlineable.) The %extras are additional options to be passed to Error::TypeTiny::Assertion's constructor and must be key-value pairs of strings only, no references or undefs.

Other methods

"qualified_name"

For non-anonymous type constraints that have a library, returns a qualified "MyLib::MyType" sort of name. Otherwise, returns the same as "name".

"isa($class)", "can($method)", "AUTOLOAD(@args)"

If Moose is loaded, then the combination of these methods is used to mock a Moose::Meta::TypeConstraint.

If Mouse is loaded, then "isa" mocks Mouse::Meta::TypeConstraint.

"DOES($role)"

Overridden to advertise support for various roles.

See also Type::API::Constraint, etc.

"TIESCALAR", "TIEARRAY", "TIEHASH"

These are provided as hooks that wrap Type::Tie. (Type::Tie is distributed separately, and can be used with non-Type::Tiny type constraints too.) They allow the following to work:

```
use Types::Standard qw(Int);
tie my @list, Int;
push @list, 123, 456;   # ok
push @list, "Hello";    # dies
```

The following methods exist for Moose/Mouse compatibility, but do not do anything useful.

"compile_type_constraint"

"hand_optimized_type_constraint"

"has_hand_optimized_type_constraint"

"inline_environment"

"meta"

Overloading

- Stringification is overloaded to return the qualified name.
- Boolification is overloaded to always return true.
- Coderefification is overloaded to call "assert_return".
- On Perl 5.10.1 and above, smart match is overloaded to call "check".

- ? The "==" operator is overloaded to call "equals".
- ? The "<" and ">" operators are overloaded to call "is_subtype_of" and "is_supertype_of".
- ? The "~" operator is overloaded to call "complementary_type".
- ? The "|" operator is overloaded to build a union of two type constraints. See Type::Tiny::Union.
- ? The "&" operator is overloaded to build the intersection of two type constraints. See Type::Tiny::Intersection.

Previous versions of Type::Tiny would overload the "+" operator to call "plus_coercions" or "plus_fallback_coercions" as appropriate. Support for this was dropped after 0.040.

Constants

"Type::Tiny::SUPPORT_SMARTMATCH"

Indicates whether the smart match overload is supported on your version of Perl.

Package Variables

$Type::Tiny::DD

This undef by default but may be set to a coderef that Type::Tiny and related modules will use to dump data structures in things like error messages.

Otherwise Type::Tiny uses it's own routine to dump data structures. $DD may then be set to a number to limit the lengths of the dumps. (Default limit is 72.)

This is a package variable (rather than get/set class methods) to allow for easy localization.

$Type::Tiny::AvoidCallbacks

If this variable is set to true (you should usually do it in a "local" scope), it acts as a hint for type constraints, when generating inlined code, to avoid making any callbacks to variables and functions defined outside the inlined code itself.

This should have the effect that "$type->inline_check('$foo')" will return a string of code capable of checking the type on Perl installations that don't have Type::Tiny installed. This is intended to allow Type::Tiny to be used with things like Mite.

The variable works on the honour system. Types need to explicitly check it and decide to generate different code based on its truth value. The bundled types in Types::Standard, Types::Common::Numeric, and Types::Common::String all do. (StrMatch is sometimes unable to, and will issue a warning if it needs to rely on callbacks when asked not to.)

Most normal users can ignore this.

Environment

"PERL_TYPE_TINY_XS"

Currently this has more effect on Types::Standard than Type::Tiny. In future it may be used to trigger or suppress the loading XS implementations of parts of Type::Tiny.

BUGS

Please report any bugs to <https://github.com/tobyink/p5-type-tiny/issues>.

SEE ALSO

The Type::Tiny homepage <https://typetiny.toby.ink/>.

Type::Tiny::Manual, Type::API.

Type::Library, Type::Utils, Types::Standard, Type::Coercion.

Type::Tiny::Class, Type::Tiny::Role, Type::Tiny::Duck, Type::Tiny::Enum, Type::Tiny::Union, Type::Tiny::Intersection.

Moose::Meta::TypeConstraint, Mouse::Meta::TypeConstraint.

Type::Params.

Type::Tiny on GitHub <https://github.com/tobyink/p5-type-tiny>, Type::Tiny on Travis-CI <https://travis-ci.com/tobyink/p5-type-tiny>, Type::Tiny on AppVeyor <https://ci.appveyor.com/project/tobyink/p5-type-tiny>, Type::Tiny on Codecov <https://codecov.io/gh/tobyink/p5-type-tiny>, Type::Tiny on Coveralls <https://coveralls.io/github/tobyink/p5-type-tiny>.

AUTHOR

Toby Inkster <tobyink@cpan.org>.

THANKS

Thanks to Matt S Trout for advice on Moo integration.

COPYRIGHT AND LICENCE

DISCLAIMER OF WARRANTIES