

## NAME

Type::Tiny::Manual::Libraries – defining your own type libraries

## MANUAL

### Defining a Type

A type is an object and you can create a new one using the constructor:

```
use Type::Tiny;

my $type = Type::Tiny->new(%args);
```

A full list of the available arguments can be found in the Type::Tiny documentation, but the most important ones to begin with are:

#### name

The name of your new type. Type::Tiny uses a convention of UpperCamelCase names for type constraints. The type name may also begin with one or two leading underscores to indicate a type intended for internal use only. Types using non-ASCII characters may cause problems on older versions of Perl (pre-5.8).

Although this is optional and types may be anonymous, a name is required for a type constraint to added to a type library.

#### constraint

A code reference checking `$_` and returning a boolean. Alternatively, a string of Perl code may be provided.

If you've been paying attention, you can probably guess that the string of Perl code may result in more efficient type checks.

#### parent

An existing type constraint to inherit from. A value will need to pass the parent constraint before its own constraint would be called.

```
my $Even = Type::Tiny->new(
    name      => 'EvenNumber',
    parent    => Types::Standard::Int,
    constraint => sub {
        # in this sub we don't need to check that $_ is an Int
        # because the parent will take care of that!

        $_ % 2 == 0
    },
);
```

Although the `parent` is optional, it makes sense whenever possible to inherit from an existing type constraint to benefit from any optimizations or XS implementations they may provide.

### Defining a Library

A library is a Perl module that exports type constraints as subs. `Types::Standard`, `Types::Common::Numeric`, and `Types::Common::String` are type libraries that are bundled with Type::Tiny.

To create a type library, create a package that inherits from `Type::Library`.

```
package MyTypes {
    use Type::Library -base;

    ...; # your type definitions go here
}
```

You can add types like this:

```

package MyTypes {
    use Type::Library -base;

    my $Even = Type::Tiny->new(
        name      => 'EvenNumber',
        parent    => Types::Standard::Int,
        constraint => sub {
            # in this sub we don't need to check that $_ is an Int
            # because the parent will take care of that!

            $_ % 2 == 0
        },
    );

    __PACKAGE__->add_type($Even);
}

```

There is a shortcut for adding types if they're going to be blessed Type::Tiny objects and not, for example, a subclass of Type::Tiny. You can just pass %args directly to add\_type.

```

package MyTypes {
    use Type::Library -base;

    __PACKAGE__->add_type(
        name      => 'EvenNumber',
        parent    => Types::Standard::Int,
        constraint => sub {
            # in this sub we don't need to check that $_ is an Int
            # because the parent will take care of that!

            $_ % 2 == 0
        },
    );
}

```

The add\_type method returns the type it just added, so it can be stored in a variable.

```
my $Even = __PACKAGE__->add_type(...);
```

This can be useful if you wish to use \$Even as the parent type to some other type you're going to define later.

Here's a bigger worked example:

```

package Example::Types {
    use Type::Library -base;
    use Types::Standard -types;
    use DateTime;

    # Type::Tiny::Class is a subclass of Type::Tiny for creating
    # InstanceOf-like types. It's kind of better though because
    # it does cool stuff like pass through $type->new(%args) to
    # the class's constructor.
    #
    my $dt = __PACKAGE__->add_type(
        Type::Tiny::Class->new(
            name      => 'Datetime',
            class     => 'DateTime',
        )
    )
}

```

```

);

my $dth = __PACKAGE__->add_type(
    name      => 'DatetimeHash',
    parent    => 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 ],
    ],
);

my $eph = __PACKAGE__->add_type(
    name      => 'EpochHash',
    parent    => Dict[ epoch => Int ],
);

# Can't just use "plus_coercions" method because that creates
# a new anonymous child type to add the coercions to. We want
# to add them to the type which exists in this library.
#
$dt->coercion->add_type_coercions(
    Int,      q{ DateTime->from_epoch(epoch => $_) },
    Undef,    q{ DateTime->now() },
    $dth,     q{ DateTime->new(%$_) },
    $eph,     q{ DateTime->from_epoch(%$_) },
);

__PACKAGE__->make_immutable;
}

```

`make_immutable` freezes to coercions of all the types in the package, so no outside code can tamper with the coercions, and allows `Type::Tiny` to make optimizations to the coercions, knowing they won't later be altered. You should always do this at the end.

The library will export types **Datetime**, **DatetimeHash**, and **EpochHash**. The **Datetime** type will have coercions from **Int**, **Undef**, **DatetimeHash**, and **EpochHash**.

### Extending Libraries

`Type::Utils` provides a helpful function `extends`.

```

package My::Types {
    use Type::Library -base;
    use Type::Utils qw( extends );

    BEGIN { extends("Types::Standard") };

    # define your own types here
}

```

The `extends` function (which you should usually use in a `BEGIN { }` block not only loads another type library, but it also adds all the types from it to your library.

This means code using the above `My::Types` doesn't need to do:

```
use Types::Standard qw( Str );
use My::Types qw( Something );
```

It can just do:

```
use My::Types qw( Str Something );
```

Because all the types from `Types::Standard` have been copied across into `My::Types` and are also available there.

`extends` can be passed a list of libraries; you can inherit from multiple existing libraries. It can also recognize and import types from `MooseX::Types`, `MouseX::Types`, and `Specio::Exporter` libraries.

### Custom Error Messages

A type constraint can have custom error messages. It's pretty simple:

```
Type::Tiny->new(
  name      => 'EvenNumber',
  parent    => Types::Standard::Int,
  constraint => sub {
    # in this sub we don't need to check that $_ is an Int
    # because the parent will take care of that!

    $_ % 2 == 0
  },
  message   => sub {
    sprintf '%s is not an even number', Type::Tiny::_dd($_);
  },
);
```

The message coderef just takes a value in `$_` and returns a string. It may use `Type::Tiny::_dd()` as a way of pretty-printing a value. (Don't be put off by the underscore in the function name. `_dd()` is an officially supported part of `Type::Tiny`'s API now.)

You don't have to use `_dd()`. You can generate any error string you like. But `_dd()` will help you make `undef` and the empty string look different, and will pretty-print references, and so on.

There's no need to supply an error message coderef unless you really want custom error messages. The default sub should be reasonable.

### Inlining

In Perl, sub calls are relatively expensive in terms of memory and CPU use. The **PositiveInt** type inherits from **Int** which inherits from **Num** which inherits from **Str** which inherits from **Defined** which inherits from **Item** which inherits from **Any**.

So you might think that to check if `$value` is a **PositiveInt**, it needs to be checked all the way up the inheritance chain. But this is where one of `Type::Tiny`'s big optimizations happens. `Type::Tiny` can glue together a bunch of checks with a stringy eval, and get a single coderef that can do all the checks in one go.

This is why when `Type::Tiny` gives you a choice of using a coderef or a string of Perl code, you should usually choose the string of Perl code. A single coderef can "break the chain".

But these automatically generated strings of Perl code are not always as efficient as they could be. For example, imagine that **HashRef** is defined as:

```

my $Defined = Type::Tiny->new(
    name      => 'Defined',
    constraint => 'defined($_)',
);
my $Ref = Type::Tiny->new(
    name      => 'Ref',
    parent    => $Defined,
    constraint => 'ref($_)',
);
my $HashRef = Type::Tiny->new(
    name      => 'HashRef',
    parent    => $Ref,
    constraint => 'ref($_) eq "HASH"',
);

```

Then the combined check is:

```
defined($_) and ref($_) and ref($_) eq "HASH"
```

Actually in practice it's even more complicated, because `Type::Tiny` needs to localize and set `$_` first.

But in practice, the following should be a sufficient check:

```
ref($_) eq "HASH"
```

It is possible for the **HashRef** type to have more control over the string of code generated.

```

my $HashRef = Type::Tiny->new(
    name      => 'HashRef',
    parent    => $Ref,
    constraint => 'ref($_) eq "HASH"',
    inline    => sub {
        my $varname = pop;
        sprintf 'ref(%s) eq "HASH"', $varname;
    },
);

```

The inlined coderef gets passed the name of a variable to check. This could be `'$_'` or `'$var'` or `$some{deep}{thing}[0]`. Because it is passed the name of a variable to check, instead of always checking `$_`, this enables very efficient checking for parameterized types.

Although in this case, the inlining coderef is just returning a string, technically it returns a list of strings. If there's multiple strings, `Type::Tiny` will join them together in a big `"&&"` statement.

As a special case, if the first item in the returned list of strings is `undef`, then `Type::Tiny` will substitute the parent type constraint's inlined string in its place. So an inlining coderef for even numbers might be:

```

Type::Tiny->new(
    name      => 'EvenNumber',
    parent    => Types::Standard::Int,
    constraint => sub { $_ % 2 == 0 },
    inline    => sub {
        my $varname = pop;
        return (undef, "$varname % 2 == 0");
    },
);

```

Even if you provide a coderef as a string, an inlining coderef has the potential to generate more efficient code, so you should consider providing one.

## Parameterizable Types

This is probably the most “meta” concept that is going to be covered. Building your own type constraint that can be parameterized like **ArrayRef** or **HasMethods**.

The type constraint we’ll build will be **MultipleOf[\$i]** which checks that an integer is a multiple of \$i.

```
__PACKAGE__->add_type(
    name      => 'MultipleOf',
    parent    => Int,

    # This coderef gets passed the contents of the square brackets.
    constraint_generator => sub {
        my $i = assert_Int(shift);
        # needs to return a coderef to use as a constraint for the
        # parameterized type
        return sub { $_ % $i == 0 };
    },

    # optional but recommended
    inline_generator => sub {
        my $i = shift;
        return sub {
            my $varname = pop;
            return (undef, "$varname % $i == 0");
        };
    },

    # probably the most complex bit
    coercion_generator => sub {
        my $i = $_[2];
        require Type::Coercion;
        return Type::Coercion->new(
            type_coercion_map => [
                Num, qq{ int($i * int(\$_/$i)) }
            ],
        );
    },
);
```

Now we can define an even number like this:

```
__PACKAGE__->add_type(
    name      => 'EvenNumber',
    parent    => __PACKAGE__->get_type('MultipleOf')->of(2),
    coercion  => 1, # inherit from parent
);
```

Note that it is possible for a type constraint to have a constraint *and* a constraint\_generator.

```
BaseType          # uses the constraint
BaseType[]        # constraint_generator with no arguments
BaseType[$x]      # constraint_generator with an argument
```

In the **MultipleOf** example above, **MultipleOf[]** with no number would throw an error because of `assert_Int(shift)` not finding an integer.

But it is certainly possible for **BaseType[]** to be meaningful and distinct from **BaseType**.

For example, **Tuple** is just the same as **ArrayRef** and accepts any arrayref as being valid. But **Tuple[]** will only accept arrayrefs with zero elements in them. (Just like **Tuple[Any,Any]** will only accept arrayrefs with

two elements.)

## **NEXT STEPS**

After that last example, probably have a little lie down. Once you're recovered, here's your next step:

- Type::Tiny::Manual::UsingWithMoose

How to use Type::Tiny with Moose, including the advantages of Type::Tiny over built-in type constraints, and Moose-specific features.

## **AUTHOR**

Toby Inkster <tobyink@cpan.org>.

## **COPYRIGHT AND LICENCE**

This software is copyright (c) 2013–2014, 2017–2019 by Toby Inkster.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

## **DISCLAIMER OF WARRANTIES**

THIS PACKAGE IS PROVIDED “AS IS” AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.