

MooseX::Method::Signatures(3pm) User Contributed Perl Documentation MooseX::Method::Signatures(3pm)

**NAME**

MooseX::Method::Signatures – (DEPRECATED) Method declarations with type constraints and no source filter

**VERSION**

version 0.49

**SYNOPSIS**

```
package Foo;

use Moose;
use MooseX::Method::Signatures;

method morning (Str $name) {
    $self->say("Good morning ${name}!");
}

method hello (Str :$who, Int :$age where { $_ > 0 }) {
    $self->say("Hello ${who}, I am ${age} years old!");
}

method greet (Str $name, Bool :$excited = 0) {
    if ($excited) {
        $self->say("GREETINGS ${name}!");
    }
    else {
        $self->say("Hi ${name}!");
    }
}

$foo->morning('Resi');                                # This works.

$foo->hello(who => 'world', age => 42);                  # This too.

$foo->greet('Resi', excited => 1);                      # And this as well.

$foo->hello(who => 'world', age => 'fortytwo');          # This doesn't.

$foo->hello(who => 'world', age => -23);                 # This neither.

$foo->morning;                                          # Won't work.

$foo->greet;                                           # Will fail.
```

**DESCRIPTION**

Provides a proper method keyword, like “sub” but specifically for making methods and validating their arguments against Moose type constraints.

**DEPRECATION NOTICE**

**Warning:** MooseX::Method::Signatures and MooseX::Declare are based on Devel::Declare, a giant bag of crack originally implemented by mst with the goal of upsetting the perl core developers so much by its very existence that they implemented proper keyword handling in the core.

As of perl5 version 14, this goal has been achieved, and modules such as Devel::CallParser, Function::Parameters, and Keyword::Simple provide mechanisms to mangle perl syntax that don't require hallucinogenic drugs to interpret the error messages they produce.

If you want to use declarative syntax in new code, please for the love of kittens get yourself a recent perl and look at Moops and core signatures instead.

**SIGNATURE SYNTAX**

The signature syntax is heavily based on Perl 6. However not the full Perl 6 signature syntax is supported yet and some of it never will be.



**Type Constraints**

```
method foo (                $affe) # no type checking
method bar (Animal          $affe) # $affe->isa('Animal')
method baz (Animal|Human $affe) # $affe->isa('Animal') || $affe->isa('Human')
```

**Positional vs. Named**

```
method foo ( $a, $b, $c) # positional
method bar (:$a, :$b, :$c) # named
method baz ( $a, $b, :$c) # combined
```

**Required vs. Optional**

```
method foo ($a , $b!, :$c!, :$d!) # required
method bar ($a?, $b?, :$c , :$d?) # optional
```

**Defaults**

```
method foo ($a = 42) # defaults to 42
```

**Constraints**

```
method foo ($foo where { $_ % 2 == 0 }) # only even
```

**Invocant**

```
method foo (          $moo) # invocant is called $self and is required
method bar ($self:   $moo) # same, but explicit
method baz ($class:  $moo) # invocant is called $class
```

**Labels**

```
method foo (:        $affe ) # called as $obj->foo(affe => $value)
method bar (:apan($affe)) # called as $obj->foo(apan => $value)
```

**Traits**

```
method foo (Affe $bar does trait)
method foo (Affe $bar is trait)
```

The only currently supported trait is `coerce`, which will attempt to coerce the value provided if it doesn't satisfy the requirements of the type constraint.

**Placeholders**

```
method foo ($bar, $, $baz)
```

Sometimes you don't care about some parameters you're being called with. Just put the bare sigil instead of a full variable name into the signature to avoid an extra lexical variable to be created.

**Complex Example**

```
method foo ( SomeClass $thing where { $_->can('stuff') } ):
    Str $bar = "apan",
    Int :$baz! = 42 where { $_ % 2 == 0 } where { $_ > 10 } )

# the invocant is called $thing, must be an instance of SomeClass and
# has to implement a 'stuff' method
# $bar is positional, required, must be a string and defaults to "apan"
# $baz is named, required, must be an integer, defaults to 42 and needs
# to be even and greater than 10
```

**CAVEATS AND NOTES**

This module is as stable now, but this is not to say that it is entirely bug free. If you notice any odd behaviour (messages not being as good as they could for example) then please raise a bug.

**Fancy signatures**

`Parse::Method::Signatures` is used to parse the signatures. However, some signatures that can be parsed by it aren't supported by this module (yet).

**No source filter**

While this module does rely on the hairy black magic of `Devel::Declare` it does not depend on a source filter. As such, it doesn't try to parse and rewrite your source code and there should be no weird side effects.

`Devel::Declare` only effects compilation. After that, it's a normal subroutine. As such, for all that hairy magic, this module is surprisingly stable.



**What about regular subroutines?**

Devel::Declare cannot yet change the way `sub` behaves. However, the `signatures` module can. Right now it only provides very basic signatures, but it's extendable enough that plugging `MooseX::Method::Signatures` signatures into that should be quite possible.

**What about the return value?**

Type constraints for return values can be declared using

```
method foo (Int $x, Str $y) returns (Bool) { ... }
```

however, this feature only works with scalar return values and is still considered to be experimental.

**Interaction with Moose::Role**

*Methods not seen by a role's requires*

Because the processing of the `MooseX::Method::Signatures` method and the `Moose` `with` keywords are both done at runtime, it can happen that a role will require a method before it is declared (which will cause `Moose` to complain very loudly and abort the program).

For example, the following will not work:

```
# in file Canine.pm

package Canine;

use Moose;
use MooseX::Method::Signatures;

with 'Watchdog';

method bark { print "Woof!\n"; }

1;

# in file Watchdog.pm

package Watchdog;

use Moose::Role;

requires 'bark'; # will assert! evaluated before 'method' is processed

sub warn_intruder {
    my $self = shift;
    my $intruder = shift;

    $self->bark until $intruder->gone;
}

1;
```

A workaround for this problem is to use `with` only after the methods have been defined. To take our previous example, **Canine** could be reworked thus:

```
package Canine;

use Moose;
use MooseX::Method::Signatures;

method bark { print "Woof!\n"; }

with 'Watchdog';
```



MooseX::Method::Signatures(3pm) User Contributed Perl Documentation MooseX::Method::Signatures(3pm)

```
1;
```

A better solution is to use `MooseX::Declare` instead of plain `MooseX::Method::Signatures`. It defers application of roles until the end of the class definition. With it, our example would become:

```
# in file Canine.pm

use MooseX::Declare;

class Canine with Watchdog {
    method bark { print "Woof!\n"; }
}

1;

# in file Watchdog.pm

use MooseX::Declare;

role Watchdog {
    requires 'bark';

    method warn_intruder ( $intruder ) {
        $self->bark until $intruder->gone;
    }
}

1;
```

#### *Subroutine redefined warnings*

When composing a `Moose::Role` into a class that uses `MooseX::Method::Signatures`, you may get a “Subroutine redefined” warning. This happens when both the role and the class define a method/subroutine of the same name. (The way roles work, the one defined in the class takes precedence.) To eliminate this warning, make sure that your `with` declaration happens after any method/subroutine declarations that may have the same name as a method/subroutine within a role.

### SEE ALSO

- `MooseX::Declare`
- `Method::Signatures::Simple`
- `Method::Signatures`
- `Devel::Declare`
- `Parse::Method::Signatures`
- `Moose`
- `signatures`

### SUPPORT

Bugs may be submitted through the RT bug tracker <https://rt.cpan.org/Public/Dist/Display.html?Name=MooseX-Method-Signatures> (or `bug-MooseX-Method-Signatures AT rt DOT cpan DOT org` [<mailto:bug-MooseX-Method-Signatures AT rt DOT cpan DOT org>](mailto:bug-MooseX-Method-Signatures AT rt DOT cpan DOT org)).

There is also a mailing list available for users of this distribution, at [<http://lists.perl.org/list/moose.html>](http://lists.perl.org/list/moose.html).

There is also an irc channel available for users of this distribution, at `irc://irc.perl.org/#moose`.

I am also usually active on irc, as ‘ether’ at `irc.perl.org`.

### AUTHOR

Florian Ragwitz [<rafl AT debian DOT org>](mailto:rafl AT debian DOT org)



## CONTRIBUTORS

- Karen Etheridge <ether AT cpan DOT org>
- Ash Berlin <ash AT cpan DOT org>
- Daniel Ruoso <daniel AT ruoso DOT com>
- Justin Hunter <justin DOT d DOT hunter AT gmail DOT com>
- Nicholas Perez <nperez AT cpan DOT org>
- Dagfinn Ilmari Mannsåker <ilmari AT ilmari DOT org>
- Rhesa Rozendaal <rhesa AT cpan DOT org>
- Yanick Champoux <yanick AT babyl DOT dyndns DOT org>
- Cory Watson <gphat AT cpan DOT org>
- Kent Fredric <kentfredric AT gmail DOT com>
- Lukas Mai <l DOT mai AT web DOT de>
- Matt Kraai <kraai AT ftbfs DOT org>
- Jonathan Scott Duff <duff AT pobox DOT com>
- Jesse Luehrs <doy AT tozt DOT net>
- Hakim Cassimally <osfameron AT cpan DOT org>
- Dave Rolsky <autarch AT urth DOT org>
- Ricardo SIGNES <rjbs AT cpan DOT org>
- Sebastian Willert <willert AT cpan DOT org>
- Steffen Schwigon <ss5 AT renormalist DOT net>

## COPYRIGHT AND LICENCE

This software is copyright (c) 2008 by Florian Ragwitz.

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

