Hi.

I'm Michael Schwern.

Some of you may know me as Perl's bitch.
Let's just say CPAN is MY bitch.
I work as an independent consultant (hire me).

I want to talk to you today about a number.

240,000

```
my $self = shift;
```

Out of 20 million lines of module code in minicpan

~4000 authors on CPAN
60 per author
Half write more than one module
120 per author
Many don't write OO code, so it gets even higher.

Why do we have to do this?

```
sub foo {
    my $self = shift;
    my($this, $that) = @_;

    ...
}
```

Traditional thinking has some answers

1) Nothing can handle all the TMTOWDI of Perl.
2) Backwards compat
3) It's not so bad.

I thought this, too. Then I worked at a place with a simple source filter...

```
sub foo($self, $this, $that) {
    ...
}
```

And I coded using that for a year, and it was kinda nice.

Then I left that job.

And I found I missed it.
I wanted that back.
I'd become addicted to its conciseness.
It turns out it is that bad, but you only realize that once you've experienced something better.

But...

Source filters are the root of all ~~eval~~ evil.

Because only perl can reliably parse perl.
And a source filter has to run over the whole of your Perl code, not just the one part you care about.
So source filters are right out.
Which leaves us with...

...nothing.

What was needed was a way to hook into or trick the Perl compiler. To tell us when it saw a keyword and let us take over. But this is impossible.

When I want something that requires an impossible piece of technology, I talk about it a lot. And then usually someone makes it possible.

Simon Cozens took a swing at it with a prototype called Method.pm.

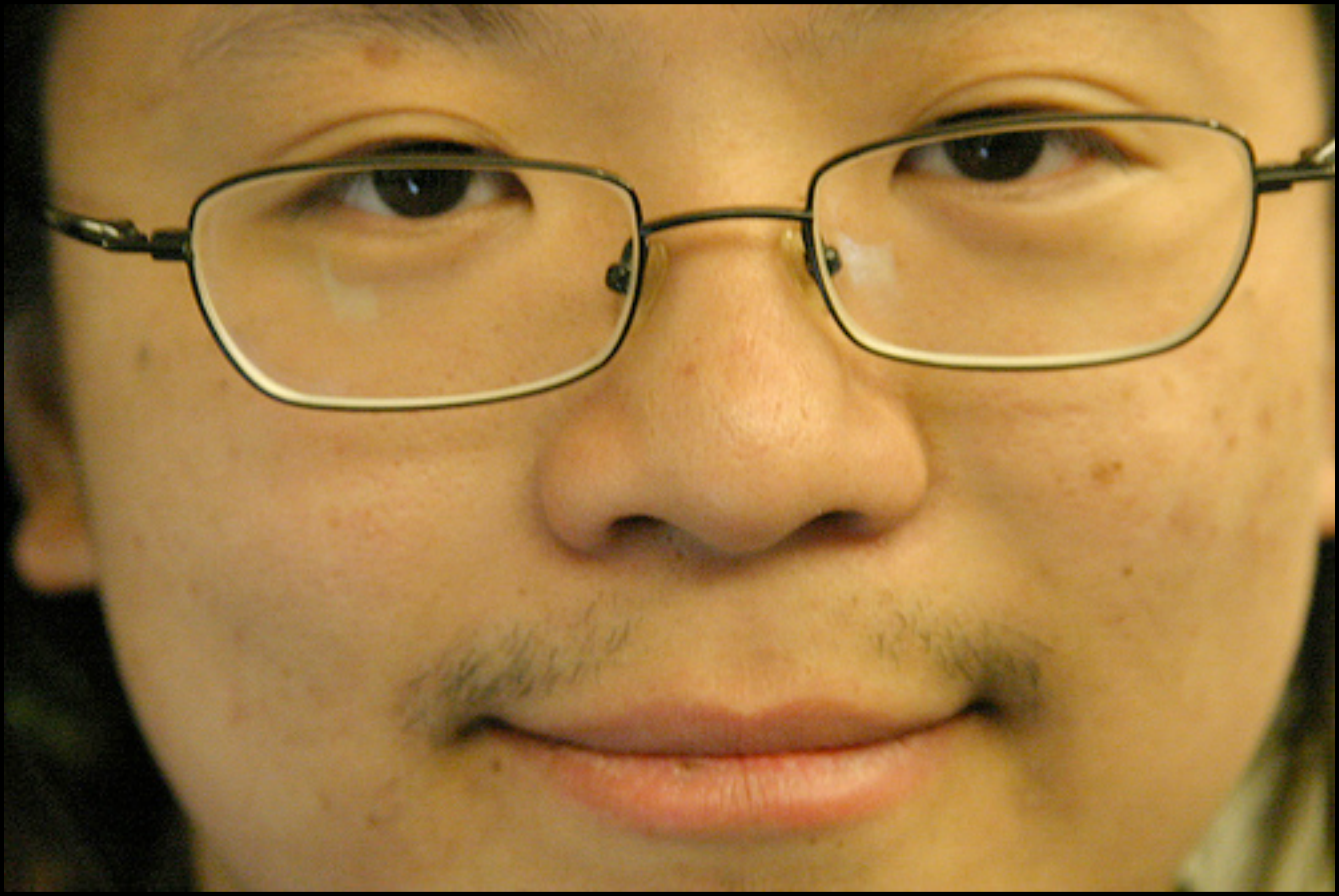
```
use Method;
sub foo :method {
    $self->bar();
}
```

Just after compile time, during the CHECK phase
Method.pm would walk the bytecode of the newly compiled module
Looking for subroutines with the method attribute.
Then it would inject the equivalent of "my $self = shift" into the bytecode.
A good idea...

...but it didn't work.

However, it did introduce the technique of injecting code directly into the opcode tree.

It was possible.



Then clkao picked up the torch and ran with it.

He got pretty far, and we got Data::Bind out of it but still no way to write subroutine prototypes.

Then Matt Trout took a lot of drugs and we got Devel::Declare.

This terrifying hack hooks into Perl's tokenizer and lets you rewrite code just before it's parsed.

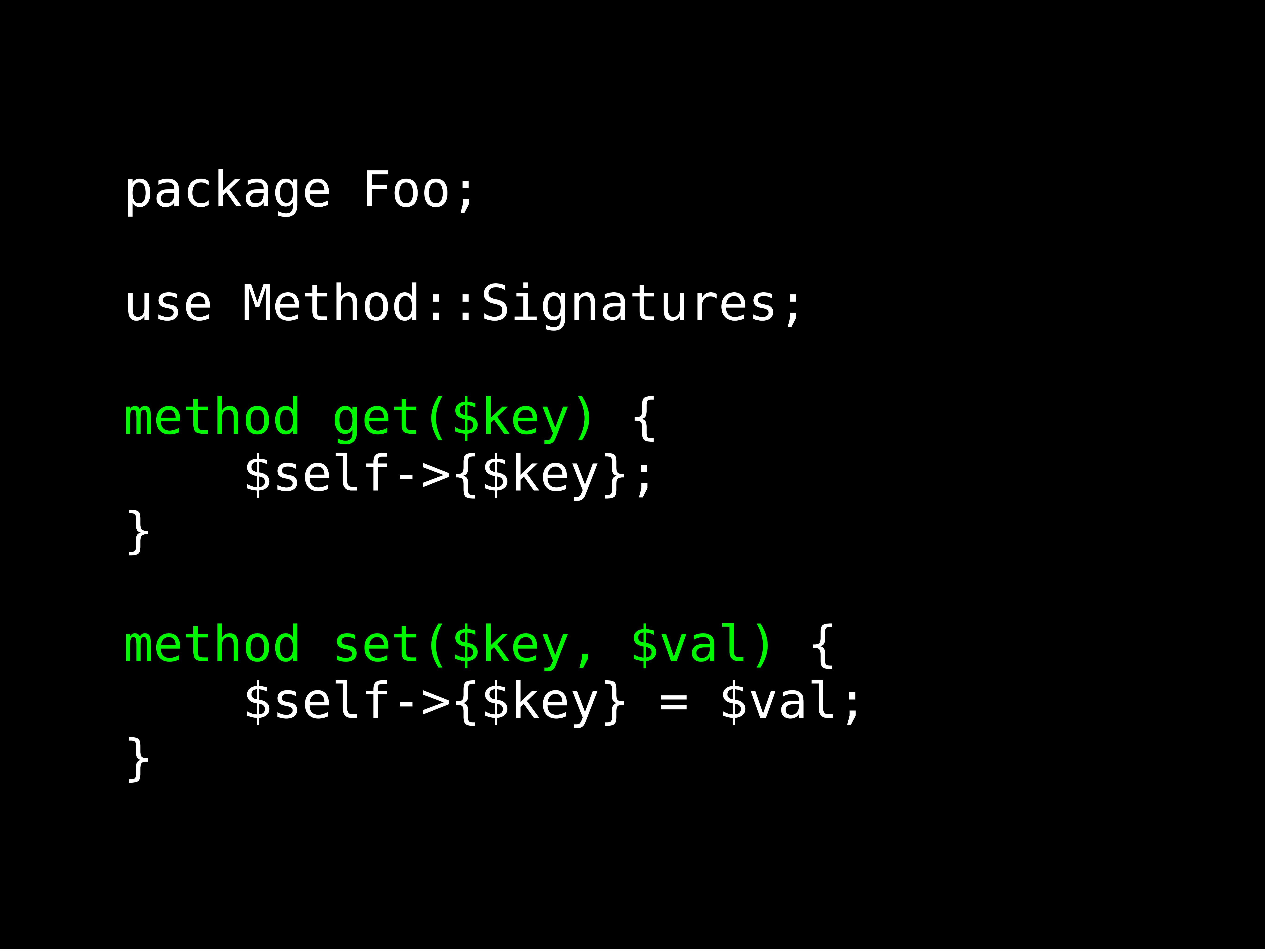
```
Devel::Declare->setup_for(
    $caller,
    { $keyword =>
        { const => \&parser }
    }
);
```

The end result is you can write new block keywords.

It's not trivial by a long shot, but it works and it's reliable.

I put a little sugar on top of that and released...

Method::Signatures



```
package Foo;

use Method::Signatures;

method get($key) {
    $self->{$key};
}

method set($key, $val) {
    $self->{$key} = $val;
}
```

Isn't that nice?

Method::Signatures rewrites this into normal Perl 5, without a source filter!

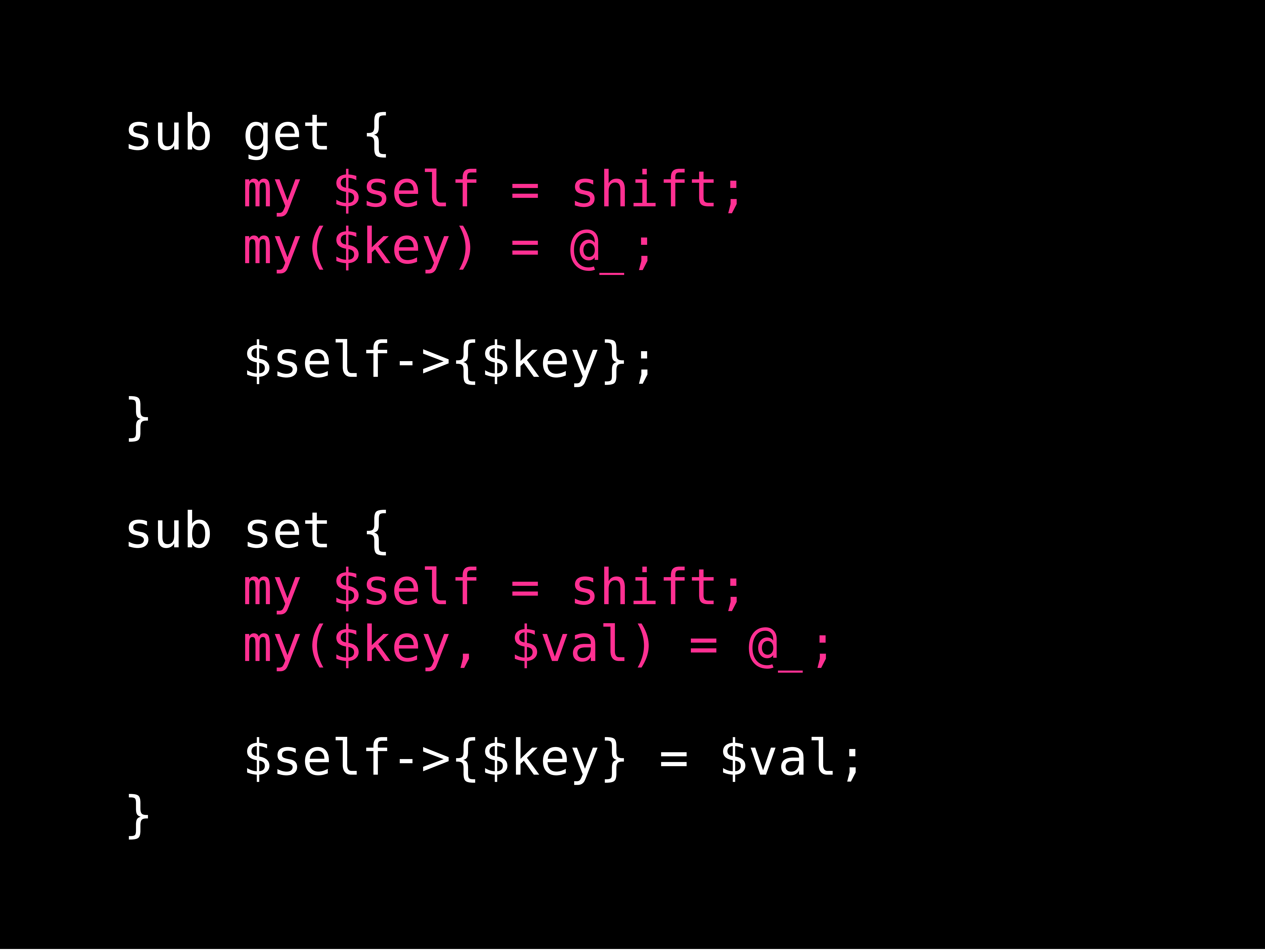
So this becomes...

```perl
package Foo;

sub get {
    my $self = shift;
    my($key) = @_;

    $self->{$key};
}

sub set {
    my $self = shift;
    my($key, $val) = @_;

    $self->{$key} = $val;
}
```

```
use Method::Signatures;

method get($key) {
    $self->{$key};
}

method set($key, $val) {
    $self->{$key} = $val;
}
```

There's only one line of scaffolding here to load the module.

Everything else is significant.



```
sub get {
    my $self = shift;
    my($key) = @_;

    $self->{$key};
}

sub set {
    my $self = shift;
    my($key, $val) = @_;

    $self->{$key} = $val;
}
```

Compare with this.

Less scaffolding means clearer code. The point is made clear.

That's pretty cool.

But if you remember earlier on we said there are 3 excuses why Perl 5 still has no signatures.

1. We don't need em

~~1. We don't need em~~

Once you use signatures you'll never go back.

2. Backwards Compatibility

~~2. Backwards Compatibility~~

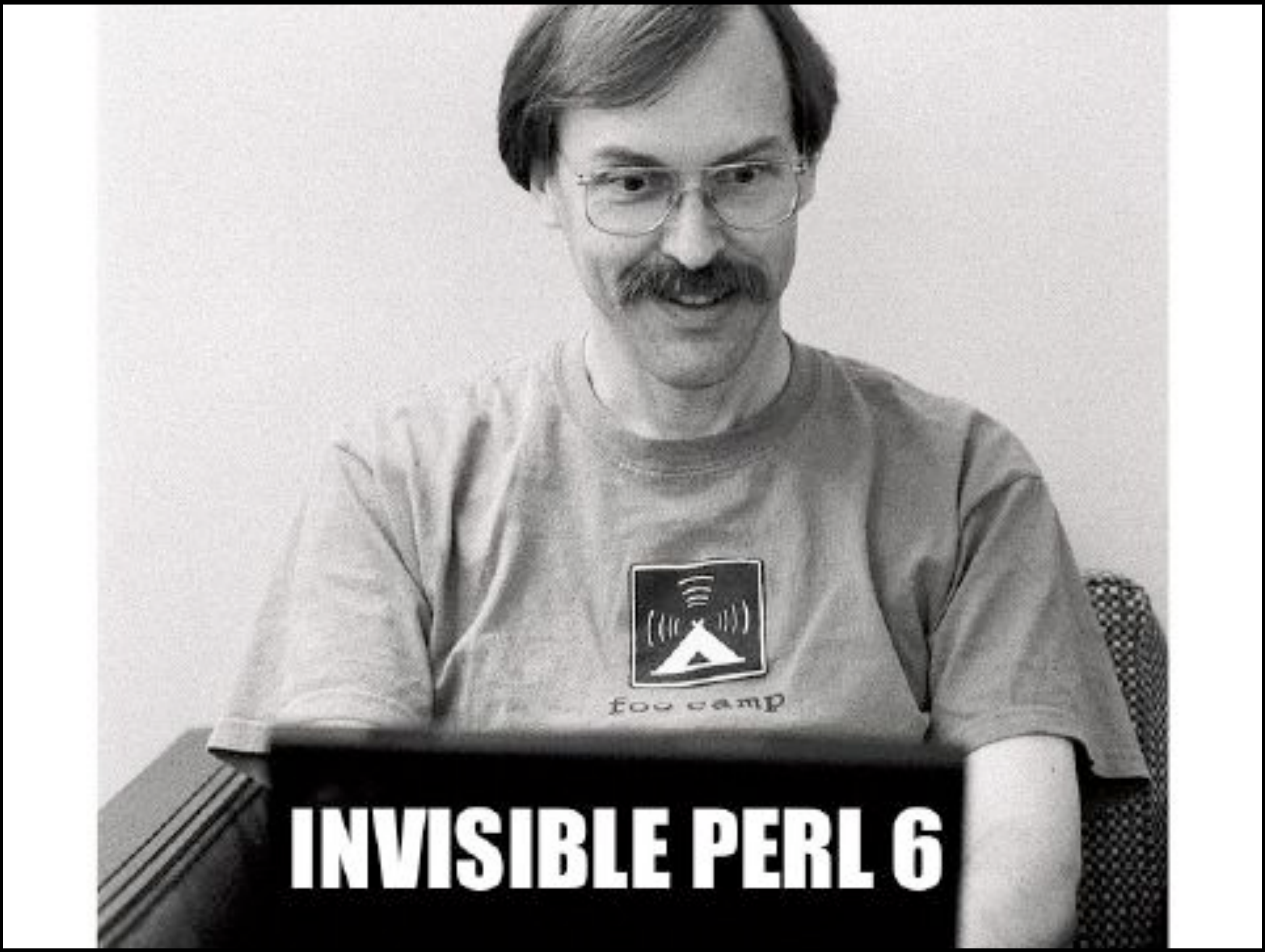
Method::Signatures works back to 5.8.1

3. TMTOWTDI

Well, that's tough.

How do you design a signature system that can do all the things that Perl programmers want to do?

You get Larry Wall to do it.



Perl 6 might not be ready,

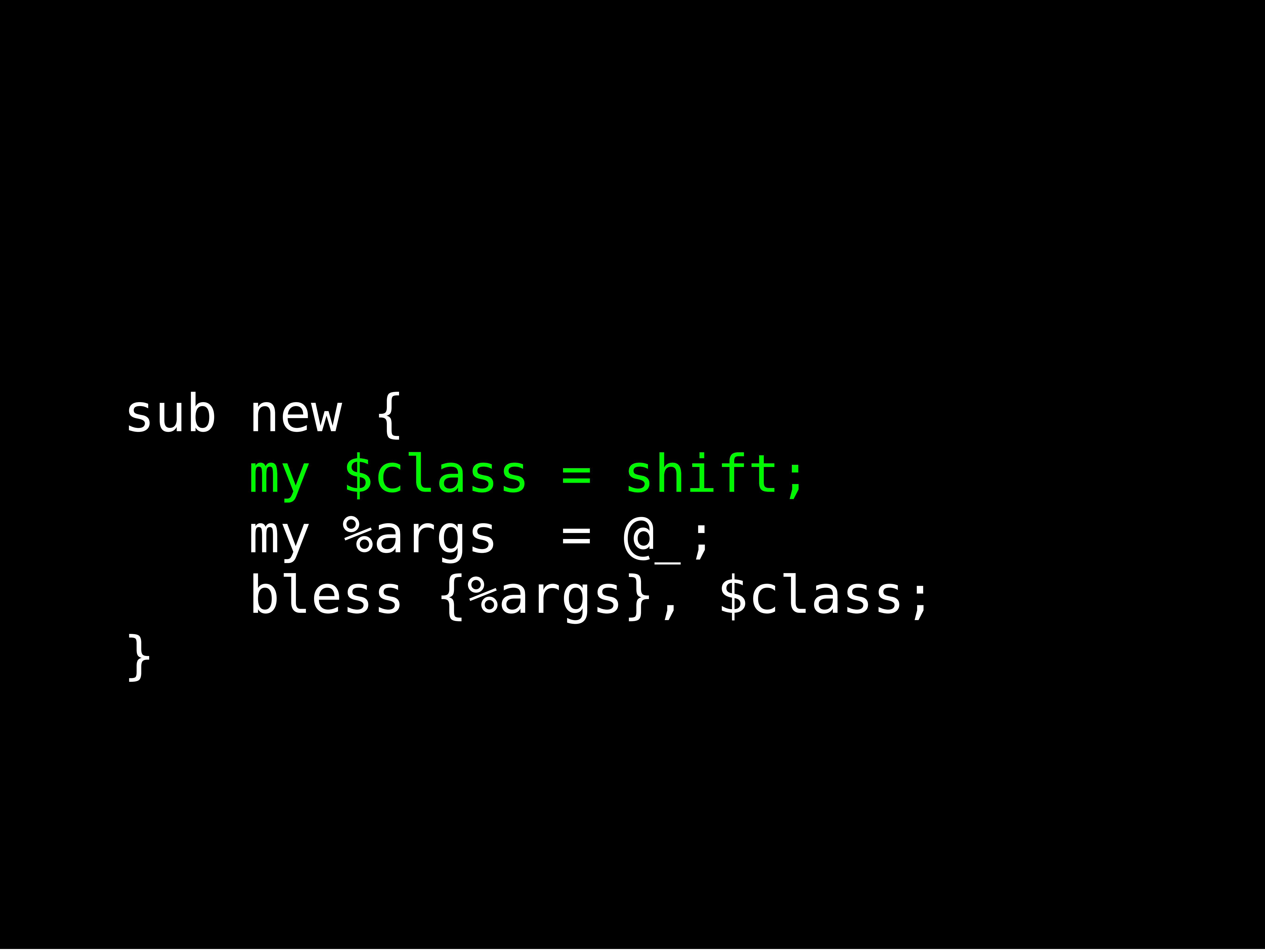
But big chunks of the spec are

So I just lifted the best parts from Synopsis 6.

```
method new($class: %args) {
    bless {%args}, $class;
}
```

Let's say you don't want the invocant to be $self. Ok.

That translates into this...



```
sub new {
    my $class = shift;
    my %args  = @_;
    bless {%args}, $class;
}
```

But isn't it wasteful to copy @_? I mean, I could just write this:

```
sub new {
    my $class = shift;
    bless {@_}, $class;
}
```

Ok, fine.

```
method new($class: @_) {
    bless {@_}, $class;
}
```

But what if you want to spell out what all the arguments are?

```
sub iso_date {
    my $self = shift;
    my %date = @_;

    return "$date{year}-$date{month}-$date{day} ".
           "$date{hour}:$date{min}:$date{sec}";
}

my $date = $obj->iso_date(
    year => 2008, month => 2,  day => 23,
    hour => 12,   min   => 23, sec => 0
);
```

Well that's just poo.

```
method iso_date(
    :$year, :$month, :$day,
    :$hour, :$min,   :$sec
)
{
    return "$year-$month-$day $hour:$min:$sec";
}

my $date = $obj->iso_date(
    year => 2008, month => 2,  day => 23,
    hour => 12,   min   => 23, sec => 0
);
```

Ahh.

But what if the user forgets an argument? Defaults would be nice.

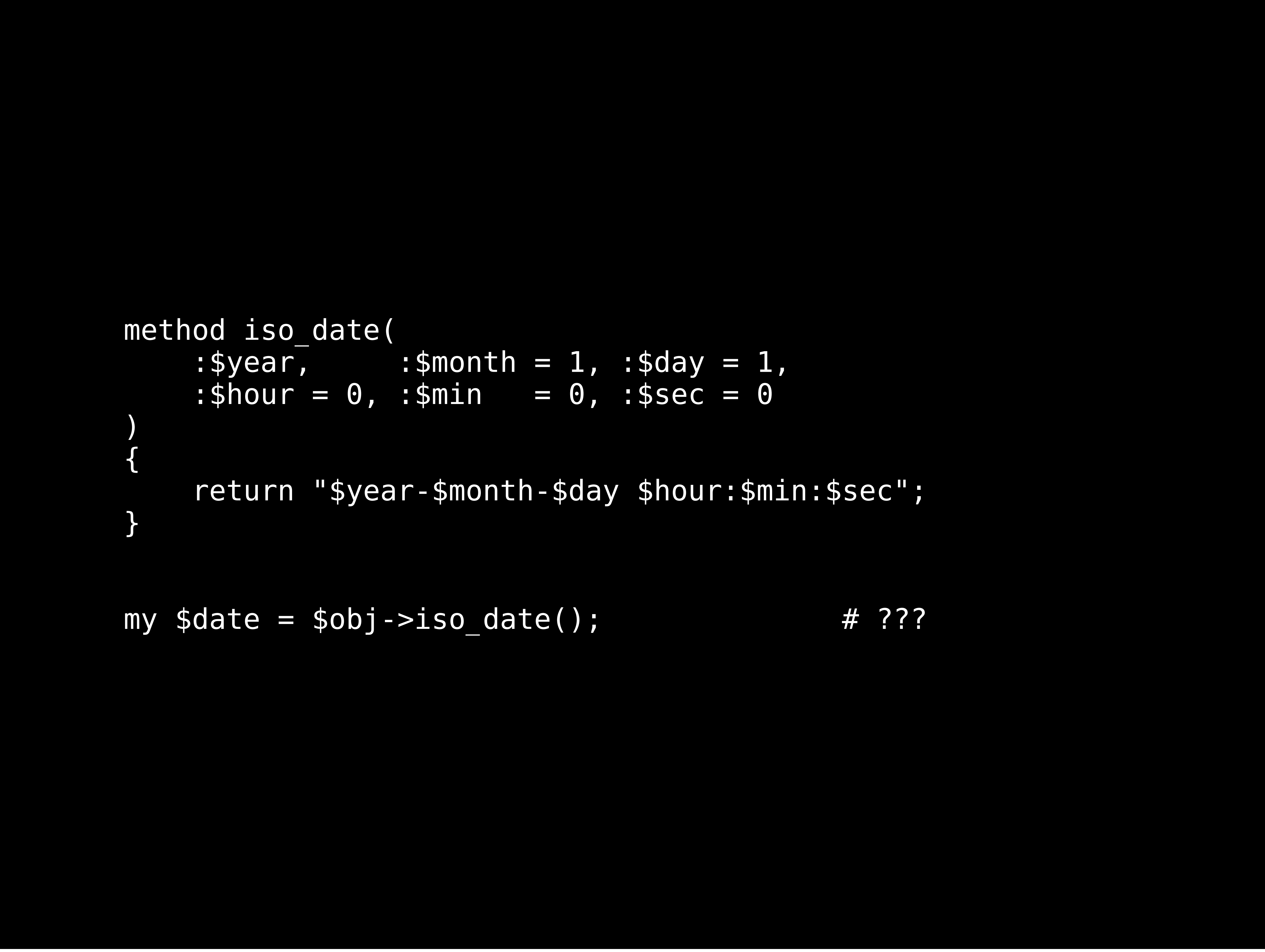
(Uhh, this currently doesn't parse)

```
method iso_date(:$year, :$month, :$day,
                :$hour, :$min,   :$sec)
{
    return "$year-$month-$day $hour:$min:$sec";
}

my $date = $obj->iso_date(
    year => 2008, month => 2,  day => 23,
    hour => 12,   min   => 23, sec => 0
);
```

This will parse.

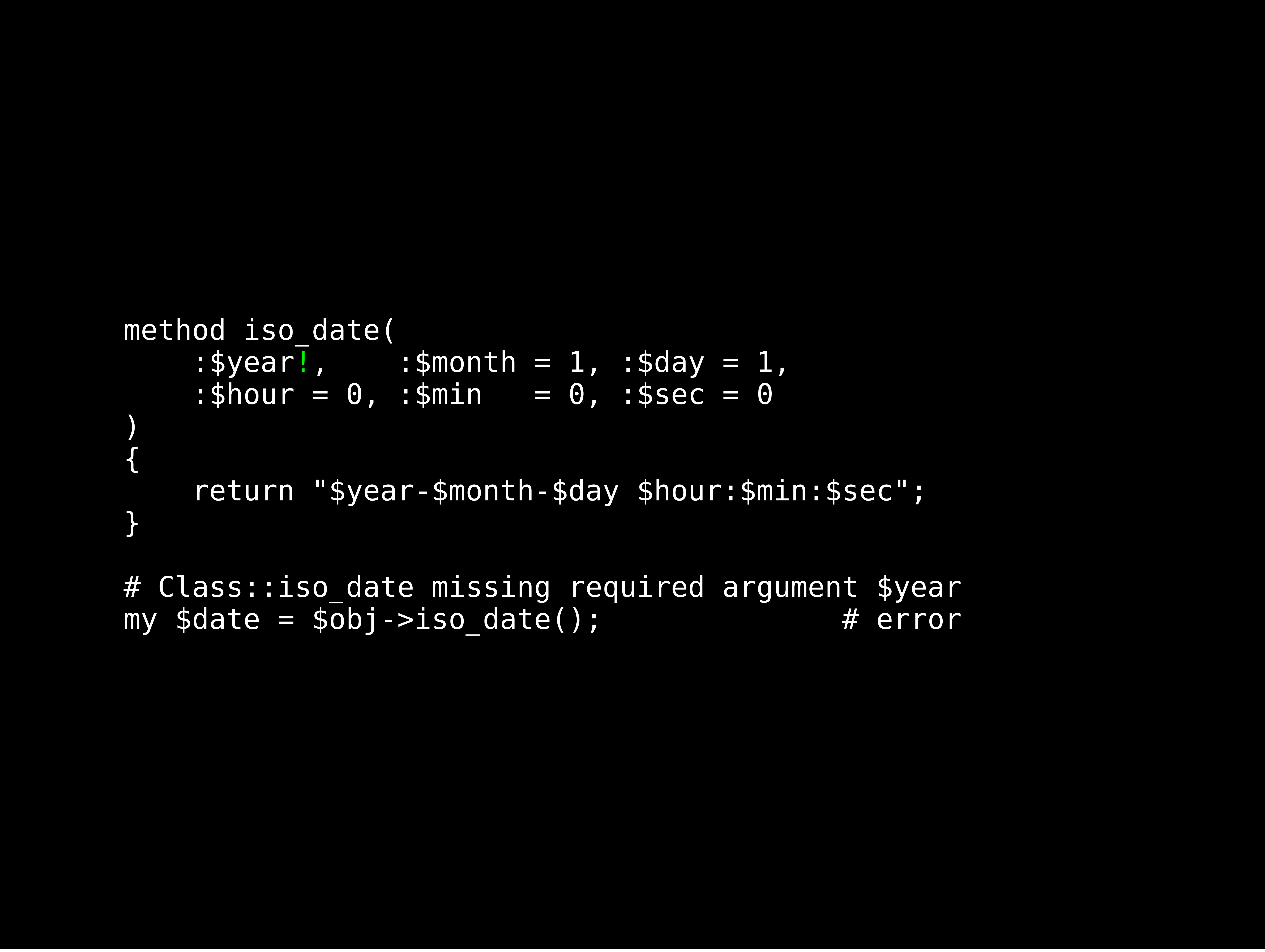
That'll be fixed.

```
method iso_date(
    :$year,     :$month = 1, :$day = 1,
    :$hour = 0, :$min   = 0, :$sec = 0
)
{
    return "$year-$month-$day $hour:$min:$sec";
}


my $date = $obj->iso_date(year => 2008);
```



```
method iso_date(
    :$year,     :$month = 1, :$day = 1,
    :$hour = 0, :$min   = 0, :$sec = 0
)
{
    return "$year-$month-$day $hour:$min:$sec";
}


my $date = $obj->iso_date();              # ???
```

But if they forget the year it goes all wrong
So how about required arguments?



```
method iso_date(
    :$year!,    :$month = 1, :$day = 1,
    :$hour = 0, :$min   = 0, :$sec = 0
)
{
    return "$year-$month-$day $hour:$min:$sec";
}

# Class::iso_date missing required argument $year
my $date = $obj->iso_date();              # error
```

Now, I want to give you an idea of just how powerful this is.
What if you wanted to do that by hand?

```
method iso_date(
    :$year!,    :$month = 1, :$day = 1,
    :$hour = 0, :$min   = 0, :$sec = 0
)
{
    return "$year-$month-$day $hour:$min:$sec";
}
```

How much code does this represent?

```
use Carp;

sub iso_date {
    my $self = shift;
    my(%args) = @_;

    croak("iso_date() missing required argument $year")
      unless exists $args{'year'};

    my $year  = delete $args{'year'};
    my $month = exists $args{'month'} ? delete $args{'month'} : 1;
    my $day   = exists $args{'day'}   ? delete $args{'day'}   : 1;
    my $hour  = exists $args{'hour'}  ? delete $args{'hour'}  : 0;
    my $min   = exists $args{'min'}   ? delete $args{'min'}   : 0;
    my $sec   = exists $args{'sec'}   ? delete $args{'sec'}   : 0;

    croak("iso_date() given extra arguments @{[ keys %args ]}")
      if keys %args;

    return "$year-$month-$day $hour:$min:$sec";
}
```

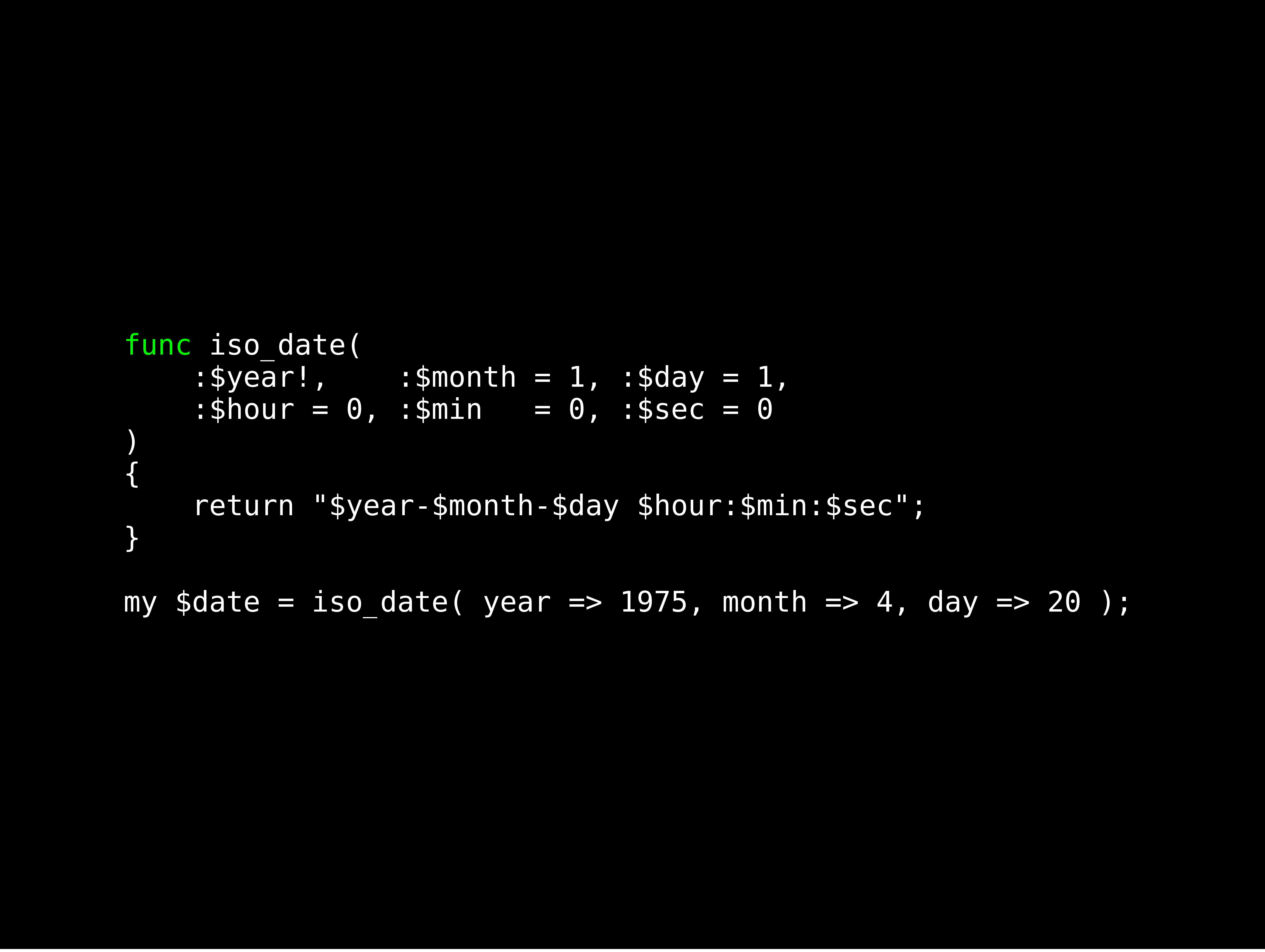
All that.

All that scaffolding goes away.

Now you no longer have to choose between being concise and being thorough.

Some more features...

Functions?



```
func iso_date(
    :$year!,    :$month = 1, :$day = 1,
    :$hour = 0, :$min   = 0, :$sec = 0
)
{
    return "$year-$month-$day $hour:$min:$sec";
}

my $date = iso_date( year => 1975, month => 4, day => 20 );
```

Doesn't do this yet
But it will

Devel::Declare currently can't override sub
Even if it could, I couldn't because...

```
sub first(&@) {
    my($code, @list) = @_;
    ...
}
```

...it already means something.
"prototypes"
If you don't know what they are, feel lucky.
They don't have much to do with signatures.
People sometimes try to pretend they do.

```
func first($code, @list) with proto(&@)
{
    ...
}
```

I'd rather keep them separate.

```
func first($code, @_) with proto(&@)
{
    ...
}
```

A bit more efficient, don't copy the list.

```
func first($code, @list is alias)
           with proto(&@)
{
    ...
}
```

Or just alias instead of copying.

```
method display($text,
               :$justify = "left", :$enchef = 0)
{
    ...
}

# $text    = "Stuff";
# $justify = "right";
# $enchef  = 0;
$obj->display("Stuff", justify => "right");
```

You can mix positional and named arguments.
There are some restrictions to avoid ambiguity.
(Just make sure to put all the positionals first)
They might be relaxed.

We have required arguments, what about optional ones?

```
method substr(
    $str, $offset, $length?, $replacement?
) {
    ...
}

$obj->substr("string", 5);  # good
$obj->substr("string");     # error
```

Some C programmers might know the "const"
Says you're not going to change that variable
Both as a promise to the caller
And a check on your code.

```
method get_page($url is ro) {
    # error
    $url = 'http://instantrimshot.com';
}
```

Keeps you honest.

```
my $method = method ($arg) {
    ...
};

$obj->$method($stuff);
```

Anonymous methods work.

When signatures came up on p5p the biggest argument was over aliasing.

Perl 6 passes by reference. Arguments are aliases, not copies.
Perl 5 passes by copy, but it is possible to alias.

```
sub strip_ws {
    $_[0] =~ s{^\s+}{};
    $_[0] =~ s{\s+$}{};
    return;
}

my $str = "  foo  ";
strip_ws($str);
```

Some folks thought it would be a good idea to alias by default.
There's some powerful things you can do with it.
It saves memory.
But it's very un-Perl5-ish for a function to change its arguments.
Unless the argument is a reference.
It's possible in Perl 5, but the syntax is yucky so few people do it
(or know about it)
It's like a gun you keep locked up and unloaded.
Making Perl 5 silently pass-by-reference is like walking around with a loaded gun.

So I feel its safer to pass by copy in Perl 5, and leave pass-by-reference to Perl 6 where it's better built into the language.

BUT!

```
method strip_ws($str is alias) {
    $str =~ s{^\s+}{};
    $str =~ s{\s+$}{};
    return;
}
```

This is done using the amazing Data::Alias module.

Unfortunately, Data::Alias is busted on Windows < 5.10 or 5.8.9.
Hopefully that will be fixed.

How much performance would you pay?

99%?

19.95%?

1%?

1%

In a fair test between Method::Signatures and equivalent hand-written methods.
That means the equivalent is doing all the same parameter validation.
M::S is 1% slower.
This is because M::S is writing out new Perl code at compile time.
So there's no difference.
If you can hand-write it, M::S can do the same.
Turns out that 1% is the difference between a hard coded subroutine and one assigned to an alias.

This whole adventure started out shortly after 5.10 came out.
Supposing what if we put function signatures into 5.12
Implement the obvious features
Keep it Perl 5 ish
Make sure it can be expanded later
I think we've got it

5.12 or bust!

Half the problem was figuring out what the syntax should be.
Now someone has to implement it inside Perl.

What's missing?

I did a code sprint this week to get all the wow features in before PPW.
But there's still a lot to be done.

sub?

Devel::Declare can't override subs yet, but they're working on it.

Meanwhile, I could write a func() keyword or something.

Once that works, I'll release a unified "signatures" pragma.

Debugger is hosed

Devel::Declare causes the debugger to have a fit.
This is the showstopper.
Hopefully there's enough drugs on Earth to help Matt fix it.

Error checking

There's almost none

The guts need work to support more.

Also throw objects.

Types

There's no type or class checks.
It might be added, but I don't want to get into making a Perl 5 type system.
Fortunately, someone else does.

MooseX::Method::Signatures

Does basically the same thing, but with antlers.
Provides a more complete Perl 6 signature implementation.
Florian and I have been swapping features.
Hopefully we'll unify the guts.

Combine Method::Signatures with Moose and Perl 5 is starting to look like a real OO language!

```
package Foo;

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

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

Foo->hello( who => "you", age => 42 );
```

Constraints

```
method divide(
    $dividend,
    $divisor where { $_ > 0 }
) {
    return $dividend / $divisor;
}
```

Return signature

There's no way to specify the return signature.

Introspection

Some way to inspect the signature of a method.

The guts need reworking to support that, too.

~~java~~perldoc?

Once you can introspect the signature
You can think about generating documentation

Multi-methods?

```
method foo($arg?) {
    if( $arg ) {
        ...do this...
    }
    else {
        ...do that...
    }
}
```

```
method foo() {
    ...do this...
}

method foo($arg) {
    ...do that...
}
```

Makes for cleaner code.
This is a distinct possibility.

This is the signature system Perl 5 deserves...
Wish it existed 10 years ago
But we didn't have all this experience and Perl 6 design to draw on.

Thanks...

Simon Cozens

clkao

Matt Trout

Already mentioned for their foundation work

Larry and the Perl 6 Dancers

The Perl 6 Design team for doing all the work on function parameters worthy of Perl

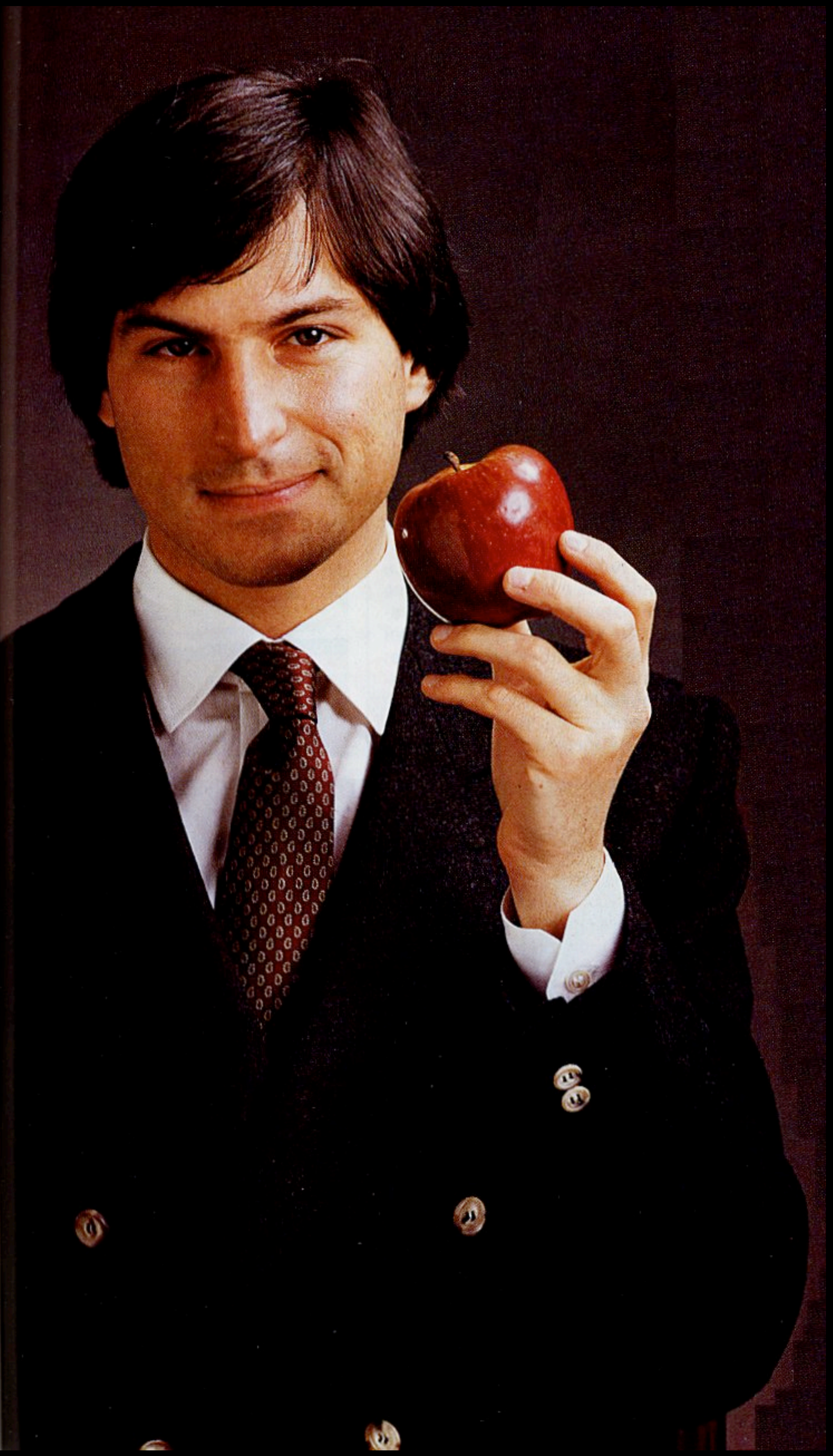
Matthijs van Duin (Data::Alias, Sub::Name)

The amazing Data::Alias module which lets "is alias" work at full speed
Sub::Name which allows the methods to have the right name in caller()

Florian Ragwitz (MooseX::Method::Signatures)

I've been bouncing ideas of Florian.
He fixed attribute handling
And hacked some on Devel::Declare.

Oh wait...



One More Thing

there is one more thing.

And I'm not thrilled about the magic aliasing feature.

```
# @foo should not change
$obj->this(@foo);

# @foo might change
$obj->that(\@foo);
```

That's really how it should work in Perl 5.

The magic aliases break that.
Now methods can reach out into their caller's data.
Without warning
That's an encapsulation violation.

But I also think Perl 5's referencing syntax can get clunky.
It's annoying to always be working with references inside methods.

Maybe we can have the best of both worlds...

```
method increment(\@args) {
    $_++ for @args;
}

my @nums = (2, 4, 8);

# @nums is now (3, 5, 9)
Foo->increment(\@nums);
```

Method::Signatures

It's on CPAN

http://slideshare.net/schwern

Slides will be on Slideshare.

Thanks!