Method::Signatures

a how-to

Today I want to talk about Method::Signatures from the perspective of "how." Mostly I won't cover the "why," although some of that will come out too. If you need more on the "why," see `Method-Signatures - PPW 2008.pdf` in the `talks/` directory in the GitHub repo.

Before we cover the "how," let's briefly cover the "what."

```
void f (int foo, string bar)
{
    // do stuff
}
```

Here's a function signature in C++. Basically it's just the part in parentheses. You can see that it's just a parenthesized list of "formal parameters," assigning each one to a variable that's local to the function definition.

```
sub f
{
    my ($foo, $bar) = @_;

    # do stuff
}
```

In Perl, however, our parameters come in via `@_` and we have to get them out ourselves. Sometimes this is flexible: there are things we can do by having direct access to `@_` that are difficult or impossible in C++. But mostly it's a PITA.

One of the precepts of Perl is that the easy things should be easy and the hard things should be possible. So passing parameters should be easy. But it isn't. Or at least it isn't as easy as it should be.

Let's look at how signatures can make our lives easy.

* Unpacking

There are two things we commonly want to do with parameters.

We want to unpack them. Get them out of `@_` and into variables so we can deal with them more easily.

* Unpacking
* Validating

And we want to validate them. Make sure that the values we're being passed are what we expect.

* UNPACKING

* Validating

Let's start with unpacking. Here's the simplest case:

```
f($x, $y);
```

Let's say we want to call a function like so. Here's a simple signature that will do that.

```
f($x, $y);


func f ($foo, $bar)
{
    # do stuff
}
```

So what's different from a normal Perl function definition? First, we use "func" instead of "sub." This is partially an implementation detail (it's much easier to create a new keyword than to hijack an existing one), and partially a signal to the reader that something unusual is going on here.

Secondly, the signature itself. These are called "positional paramters," because their position in the list determines which value goes to which parameter: the first parameter goes to `$foo`, and the second one to `$bar`.

Of course, we're not *replacing* `@_`; just adding a layer on top of it. So Perl is still going to do all the things it would normally do. For instance if you do this:

```
f(@array);


func f ($foo, $bar)
{
    # do stuff
}
```

Perl is going to send `$array[0]` to `$foo`, `$array[1]` to `$bar`, and `$array[2]` (and up) go nowhere. But, with that caveat understood, positional parameters are pretty simple.

Of course, some people prefer named parameters

```
f( foo => $x, bar => $y );
```

like so. With one or two parameters, it's mostly an aesthetic choice. But what if you had ...

```
f( foo => $x, bar => $y );
f( foo => $x, bar => $y, baz => $z, qux => $q, wibble => $w );
```

... five? Named parameters are clearly superior here so that you don't have to worry about which parameter goes where. Additionally, you can ...

```
f( foo => $x, bar => $y );
f( foo => $x, bar => $y, baz => $z, qux => $q, wibble => $w );
f( bar => $y, qux => $q, foo => $x, wibble => $w, baz => $z );
```

... put them in any order. Or even ...

```
f( foo => $x, bar => $y );
f( foo => $x, bar => $y, baz => $z, qux => $q, wibble => $w );
f( bar => $y, qux => $q, foo => $x, wibble => $w, baz => $z );
f( bar => $y, wibble => $w, baz => $z );
```

... leave some of them out.

```
f( foo => $x, bar => $y );
```

So let's say we want named parameters. Can we do that with Method::Signatures? Of course we can!

```
f( foo => $x, bar => $y );


func f (:$foo, :$bar)
{
    # do stuff
}
```

To indicate a named parameter, just put a colon in front of the parameter. Why is it a colon? Because Larry says so. (That is, Perl 6 uses colons to indicate named parameters, and that's where we stole our syntax from. If you're going to invent syntax for Perl, Larry's brain is a pretty decent place to get it from.)

So we have positional parameters and named parameters. Can we put them together?

```
f($x, $y, baz => $z);


func f ($foo, $bar, :$baz)
{
    # do stuff
}
```

Of course we can! Just remember that the positional parameters have to come first.

Of course, so far we've been talking about regular functions. But the name of the module is *Method*::Signatures. What about methods?

```
$obj->f($x, $y);
```

So what's the difference between a function and a method anyway? Well, we call it using the arrow syntax. The thing we put on the left-hand side of the arrow—in this case it's an object—is called an "invocant." (Why is it called an invocant? Because Larry says so. Do you sense a pattern here?)

So, when there's an invocant, that actually shows up as our first parameter. Obviously Method::Signatures needs to know about that, since it will be handling the unpacking. How do we tell it that?

```
$obj->f($x, $y);


method f ($foo, $bar)
{
    # do stuff with $self
}
```

So what's different here than our very first example? That's right: `method` instead of `func`; nothing else. Using `method` tells Method::Signatures to automatically create `$self` for you.

Of course, it doesn't *have* to be an object on the left-hand side of the arrow. What else can we put there?

```
My::Class->f($x, $y);
```

That's right: a class. (Technically, a class *name*. Just as with files vs filenames, sometimes that distinction is important and sometimes not.) Now, when we call a method like this, we don't want to refer to our invocant as `$self`; we want to refer to it as `$class`. So, can we do that with Method::Signatures?

```
My::Class->f($x, $y);


method f ($class: $foo, $bar)
{
    # do stuff with $class
}
```

Of course we can! When I said using `method` automatically creates `$self` for you, I oversimplified. More correctly: using `method` automatically creates an invocant, and the default invocant is `$self`. But you can name it whatever you like.

`$class` is similar to our other parameters, but it has a few special rules:

- It has to come first.
- There can be only one of them.
- It has to be a "bare" parameter. That is, all the other cool things we're going to talk about doing to parameters later, you can't do to invocants. Just the name.
- And you have to put a colon after it, and then the comma isn't necessary. (Why is it a colon? Because Larry says so. But you knew that.)

Now, notice that I said "put a colon after it and drop the comma." Why didn't I just say "use a colon instead of a comma"?

```
My::Class->f();
```

What about this example? You wouldn't need a comma for only one parameter right? (Side note: if you *do* put a comma at the end of your parameter list, it still works.)

```
My::Class->f();


method f ($class: )
{
    # do stuff with $class
}
```

But you still need the colon. Notice I put a space there between the colon and the close parend. This is not necessary. That is, this

```
My::Class->f();


method f ($class:)
{
    # do stuff with $class
}
```

parses perfectly correctly. But I don't like it. Looks too much like a smiley. Also, I like how the space reminds us visually of the empty parameter list.

So that's methods and invocants. I'm going to go back now to using functions for my examples because it's simpler, but just remember that, anything you can do with a function, you can do with a method.

So, what else do we want to do when we unpack parameters?

```
sub f
{
    my ($foo, $bar) = @_;
    $bar ||= -1;

    # do stuff
}
```

How often have you written code that looks like this? You're saying, if my caller doesn't pass a value for `$bar`, we'll use a default value instead.

Now, how often have you realized that you have a bug in code like this? What happens if someone passes 0 for `$bar`?

```
sub f
{
    my ($foo, $bar) = @_;
    $bar //= -1;

    # do stuff
}
```

What you really meant was this. Assuming you're using Perl 5.10 or above. And, if you're not ... why not?

But the point is, this is a common thing we want to do when unpacking. Can Method::Signatures help us with this?

```
f($x, $y);



func f ($foo, $bar = -1)
{
    # do stuff
}
```

Of course it can!

So this is going to work just like the `//=` code, right? Well, not quite ... That is, this one will work the same.

```
f($x, $y);
f($x);



func f ($foo, $bar = -1)
{
    # do stuff
}
```

And so will this one.

```
f($x, $y);
f($x);
f($x, undef);


func f ($foo, $bar = -1)
{
    # do stuff
}
```

But this one ... not so much.

You see, there's a difference between not passing something at all, and passing a value that's undefined. This is just like checking hash values: if you see that the value for a certain hash key is undefined, it might be that the key doesn't exist in the hash, but it might be that it exists, but the value is undefined. Sometimes you don't care about that distinction, and sometimes you do. So Perl gives you a way to treat them the same, and a way to treat them differently.

So Method::Signatures needs to give you a way to treat passing `undef` the same as not passing anything at all, and a way to treat them differently. This code is the way to treat them differently.

```
f($x, $y);
f($x);
f($x, undef);


func f ($foo, $bar //= -1)
{
    # do stuff
}
```

And this code is the way to treat them the same. Now all 3 calls will work exactly as the longhand `//=` code would.

So now we know how to apply the default in the case where `undef` is passed. But what if you want the default to apply when 0 is passed? or the empty string? or a string of nothing but spaces? Or, what if the `||=` code wasn't actually a bug, and you really did mean that you want the default applied whenever the value is false in any way? Can Method::Signatures help with *that*?

Of course it can!

```
func f ($foo, $bar //= -1)
```

You see, this code is really a shorthand way of writing ...

```
func f ($foo, $bar //= -1)

func f ($foo, $bar = -1 when undef)
```

... this code. What it's saying is, if the value isn't passed, it will be assigned the default—that *always* happens. But *also*, if `undef` is passed, the default will be assigned. So now we can see how it would be simple to instead apply the default when the passed in value ...

```
func f ($foo, $bar //= -1)


func f ($foo, $bar = -1 when undef)
func f ($foo, $bar = -1 when 0)
```

... is zero ...

```
func f ($foo, $bar //= -1)


func f ($foo, $bar = -1 when undef)
func f ($foo, $bar = -1 when 0)
func f ($foo, $bar = -1 when '')
```

... or the empty string ...

```
func f ($foo, $bar //= -1)


func f ($foo, $bar = -1 when undef)
func f ($foo, $bar = -1 when 0)
func f ($foo, $bar = -1 when '')
func f ($foo, $bar = -1 when /^\s+$/)
```

... or a string of nothing but spaces ...

```
func f ($foo, $bar //= -1)


func f ($foo, $bar = -1 when undef)
func f ($foo, $bar = -1 when 0)
func f ($foo, $bar = -1 when '')
func f ($foo, $bar = -1 when /^\s+$/)
func f ($foo, $bar = -1 when { !$_ })
```

... or anything false.

Now, the way this works is via smart matching. Remember that, if there's no value at all, you get the default automatically and the `when` part isn't even consulted. But, assuming there *is* a value, it's smart-matched against whatever you provide in the `when` clause.

And of course you can do all sorts of crazy stuff with smart matching.

```
func f ($foo, $bar = 0 when [undef, qw< no false off >])
{
    # do stuff
}
```

That one's pretty obvious, right?

```
my %ERRORS =
(
    E100    =>  "The frobnobulator is all grizzed up.",
    E233    =>  "WARNING! Dagrilated toadthwacker down.",
    E666    =>  "Satan.",
    E951    =>  "Please reset your gymnozzle.",
);

func fatal ($msg = $ERRORS{$_[0]} when \%ERRORS)
{
    say STDERR $msg;
    exit 1;
}
```

How about this one?

Basically, a smart match against a hashref is true if the value is equal to any key in the hash. The default value itself is the lookup of what was passed in that same hash. (Note that we can't use `$msg` for the lookup, because we're in the process of defining it. But `@_` is still there, so we can use that.)

```
my %ERRORS =
(
    E100    =>  "The frobnobulator is all grizzed up.",
    E233    =>  "WARNING! Dagrilated toadthwacker down.",
    E666    =>  "Satan.",
    E951    =>  "Please reset your gymnozzle.",
);

func fatal ($msg = $ERRORS{$_[0]} when \%ERRORS)
{
    say STDERR $msg;
    exit 1;
}


fatal('E100');
```

So if you pass a code in the hash, it uses the corresponding error message.

```
my %ERRORS =
(
    E100    =>  "The frobnobulator is all grizzed up.",
    E233    =>  "WARNING! Dagrilated toadthwacker down.",
    E666    =>  "Satan.",
    E951    =>  "Please reset your gymnozzle.",
);

func fatal ($msg = $ERRORS{$_[0]} when \%ERRORS)
{
    say STDERR $msg;
    exit 1;
}


fatal('E100');
fatal("These are stupid error messages.");
```

Or you can specify your own message as an override.

Now, this isn't perfect, certainly. What happens if our caller passes us a code, but it isn't a code in the hash? or if they pass nothing at all? But it's still pretty cool.

Is there anything else we might want to do in terms of unpacking? What about not unpacking at all? That is, what is the advantage of accessing a value in `@_` directly?

```
sub f
{
    my $bar = $_[1];

    # do stuff modifying $_[0]
}
```

That's right: you can change it. Now, this is not something you should do lightly. I'm not even saying you should *ever* do it, really. But at least with `@_` you *can* do it, if you should ever need to. So can you do it with Method::Signatures?

```
f($x, $y);


func f ($foo, $bar)
{
    # do stuff
}
```

Of course you can.

When you write this, this is a shortcut too. This code is actually shorthand for

```
f($x, $y);


func f ($foo is copy, $bar is copy)
{
    # do stuff
}
```

this code. In other words, the default type of a parameter is a copy, which is what people expect. But that's not the only type of parameter you can have.

```
f($x, $y);


func f ($foo is alias, $bar is copy)
{
    # do stuff modifying $foo
}
```

You can have a parameter which is an alias, and then you can modify the caller's value.

The bit with the `is` is called a "trait." (Why do we use `is`? Did you say "because Larry says so"? Good for you.) This is similar to how `is` is used in Moose. Right now there aren't very many traits; perhaps we'll add more. Besides `copy` (which you'd hardly ever use explicitly) and `alias`, there's `rw` (which you also don't need to specify explicitly), and

```
f($x, $y);


func f ($foo is ro, $bar is copy)
{
    # do stuff; not allowed to modify $foo
}
```

`ro`, which might be useful to help yourself remember not to change the value within the function.

* UNPACKING

* Validating

So that's the unpacking taken care of.

* Unpacking

* VALIDATING

Now let's look at the validating.

First of all, we want to make sure we get enough parameters. If we have a required parameter and our caller doesn't provide a value, that's a problem. But how do we know what parameters are required and which are optional?

```
f($x, $y);


func f ($foo, $bar, :$baz)
{
    # do stuff
}


# no errors
```

Positional parameters are required. Named parameters are optional. This is, generally, what people expect.

```
f($x);


func f ($foo, $bar, :$baz)
{
    # do stuff
}


In call to main::f(), missing required argument $bar at reqpos.pl line 14.
```

If you omit a required parameter, you get an error. Obviously this is a run-time error: all parameter validation errors are run-time errors. The only compile-time errors you'll get out of Method::Signatures are if your signature syntax doesn't parse.

```
f($x);


func f ($foo, $bar = 0)
{
    # do stuff
}


# no errors
```

Of course, parameters with default values are optional too. So, more precisely, positional parameters are required unless they have a default value, and named parameters are optional whether they have default values or not.

Even more precisely, positional parameters are required *by default*, unless they have default values, and named parameters are optional *by default* regardless of default value. So what if we need to change that? What if we need to declare a positional parameter as optional? Can we do that?

```
f();


func f ($foo?, $bar = 0)
{
    # do stuff
}


# no errors
```

Of course we can! Just put a question mark at the end. (Why is it a question mark? You betcha.)

Now, remember when I said that, when you mix positional parameters and named parameters, the named parameters have to go at the end? I oversimplified again. It's more correct to say that *optional* parameters have to go at the end. Doesn't really matter why they're optional, they just have to go at the end. Oh, and you can't mix named parameters with optional positional parameters because it creates ambiguity.

```
f( foo => $x );


func f (:$foo, :$bar)
{
    # do stuff
}


# no errors
```

Conversely, what if we need to declare a named parameter as required? Can we do that?

```
f( foo => $x );


func f (:$foo, :$bar!)
{
    # do stuff
}


In call to main::f(), missing required argument $bar at reqnamed.pl line 14.
```

Of course we can! Just put an exclamation point after it. (Why is it ... oh, you know the drill by now.)

What about other combinations? Can you mark a positional parameter as required?

```
f($x);


func f ($foo!, $bar = 0)
{
    # do stuff
}


# no errors
```

Sure. Redundant, but legal. How about marking a named parameter as optional?

```
f( foo => $x );


func f (:$foo, :$bar?)
{
    # do stuff
}


# no errors
```

Still redundant, but still legal. What about marking a parameter with a default value as required?

```
f($x);


func f ($foo, $bar! = 0)
{
    # do stuff
}


In call to main::f(), missing required argument $bar at combopos.pl line 14.
```

Silly, but still legal. Silly, because your default value can never get used, since failing to supply a value for it causes your function to blow up. (Of course, if you use `when` then your default value *could* get used ...)

So all those combinations are useless, but legal. What about trying to declare something both required and optional at the same time?

```
f($x);


func f ($foo?!, $bar = 0)
{
    # do stuff
}


Unexpected extra code after parameter specification: '!' in
declaration at both.pl line 10.
```

Not legal. And note that this one is a compile-time error, because it's illegal signature syntax.

Okay, fine. So Method::Signatures validates not enough parameters. What about too many?

```
f($x, $y, $z);


func f ($foo, $bar)
{
    # do stuff
}


In call to main::f(), was given too many arguments; it expec
ts 2 at extrapos.pl line 14.
```

Yep, it validates that too.

```
f($x);


func f ()
{
    # do stuff
}


In call to main::f(), was given too many arguments; it expec
ts 0 at extraempty.pl line 14.
```

Even if you give it an empty signature. An empty signature says "I'm expecting not to receive any parameters at all." So, if you get any, that's an error.

But what if you don't care about whatever parameters people pass you? This can happen if you're intercepting or wrapping a call to someone else's code, and you're just going to throw the parameters away, or pass them through. But in those cases you don't want Method::Signatures whinging about extra parameters.

```
f($x);


func f (...)
{
    # do stuff
}


# no errors
```

This is called the "yada yada" syntax, and it also comes from Perl 6. It basically says "whatever comes next, I really don't care."

```
f($x, $y);


func f ($foo, ...)
{
    # do stuff
}


# no errors
```

The yada yada doesn't have to be the only thing in the signature either. You can mix it with paramaters that you *do* care about, as long as the yada yada is at the end.

```
f( foo => $x, baz => $z );


func f (:$foo, :$bar)
{
    # do stuff
}


# ???
```

What about this one? Technically, that's not too many parameters ... but it's still wrong.

```
f( foo => $x, baz => $z );


func f (:$foo, :$bar)
{
    # do stuff
}


In call to main::f(), does not take baz as named argument(s)
 at extranamed.pl line 14.
```

And Method::Signatures knows it.

So, what else do we want to do when we validate parameters?

```
sub f
{
    my ($foo) = @_;
    die("foo must be an int") unless $foo =~ /^\d+$/;

    # do stuff
}
```

How often have you written code that looks like this? And how often have you realized *this* code has a bug too? What if you're passed a negative number? What if it's in scientific notation?

```
sub f
{
    use Regexp::Common;

    my ($foo) = @_;
    die("foo must be an int") unless $foo =~ /^$RE{num}{int}$/;

    # do stuff
}
```

Probably this is better. But starting to get a bit verbose. Can Method::Signatures help us with this one?

```
f($x);


func f (Int $foo)
{
    # do stuff
}
```

Of course it can! Just use a type.

Now, it's important to stress that these types aren't like types in other languages. In C++, for example, a type is a compile-time constraint. What we're calling a "type" here is really just a run-time validation check (just like the code it's replacing).

Also, it's important to ask the question: where do types come from? The answer is: Moose, or Mouse. (Getting types out of Moo is a bit more of a challenge, but it's on our radar.) So, if you're not using Moose (or Mouse), then you can't use types, unfortunately. But that still covers a healthy number of folks.

The type goes in front of the parameter name, as you can see. What you can put in front of the parameter name is one of three things (and they're checked in this order). First is "anything Moose (or Mouse) recognizes as a type." So, int is okay.

```
f($x);


func f (Str $foo)
{
    # do stuff
}
```

Or string.

```
f($x);


func f (Bool $foo)
{
    # do stuff
}
```

Or boolean.

```
f($x);


func f (ArrayRef $foo)
{
    # do stuff
}
```

Or arrayref.

```
f($x);


func f (HashRef $foo)
{
    # do stuff
}
```

Or hashref. Or many other things. See the manual for Moose (or Mouse) for a complete list.

```
f($x);


func f (Some::Class $foo)
{
    # do stuff
}
```

The second thing you can put there is a class name. The validation passes as long as the value is an object of that type (or derived from that type).

```
f($x);


func f (Some::Role $foo)
{
    # do stuff
}
```

And the last thing you can put there is a role name. As expected, the validation passes as long as the value is an object which "does" (i.e. composes) that role.

So what happens when you pass something that doesn't match the type?

```
f('bmoogle');


func f (Int $foo)
{
    # do stuff
}


In call to main::f(), the 'foo' parameter ("bmoogle") is not
 of type Int at typecheck.pl line 14.
```

Pretty basic.

But what if you wanted to accept only integers under 10? Well, you could make a whole new type ...

```
use Moose::Util::TypeConstraints;

subtype IntLessThanTen => as Int => where { $_ < 10 };


f($x);


func f (IntLessThanTen $foo)
{
    # do stuff
}
```

And, I suppose if you have a whole bunch of parameters you want to do like that, that might actually make sense. If it's just one parameter, though, it seems like a bit of overkill, doesn't it?

Can Method::Signatures help?

```
f($x);


func f (Int $foo where { $_ < 10 })
{
    # do stuff
}


# no errors
```

Of course it can. This is called a `where` constraint.

In this particular example, it's going to check both things:

```
f('bmoogle');


func f (Int $foo where { $_ < 10 })
{
    # do stuff
}


In call to main::f(), the 'foo' parameter ("bmoogle") is not
 of type Int at wherecheck.pl line 14.
```

first it checks the type,

```
f(100);


func f (Int $foo where { $_ < 10 })
{
    # do stuff
}


In call to main::f(), $foo value ("100") does not satisfy co
nstraint:  { $_ < 10 } at wherecheck.pl line 14.
```

and then it checks the `where` clause.

Now, just like with `when`, how this actually works is that the right-hand side of the `where` is smart-matched against the value passed in. So, again, you could do something like

```
f($x);


func f (Str $foo where \@LEGAL_VALUES)
{
    # do stuff
}


# no errors
```

this, or any of the other myriad of crazy things you can do with smart matching. Most often, though, you're going to want to use a code block.

Of course, the other cool thing about `where` is that it doesn't require Moose (or Mouse). So, if you wanted a version of this which could work without Moose/Mouse ...

```
f($x);


func f ($foo where { /^$RE{num}{int}$/ && $_ < 10 })
{
    # do stuff
}


# no errors
```

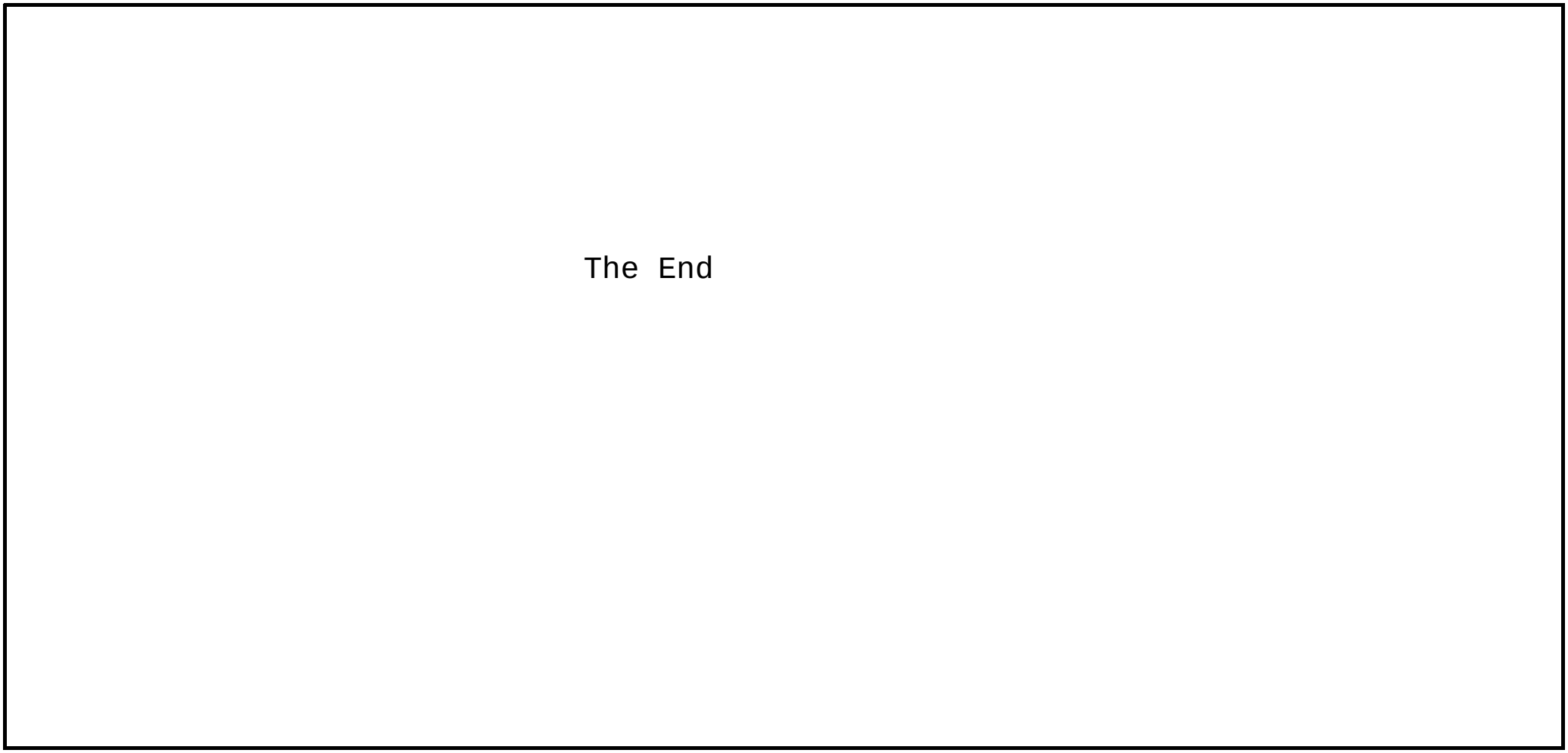
... you could do that too.

```
my %ERRORS =
(
    E100    =>  "The frobnobulator is all grizzed up.",
    E233    =>  "WARNING! Dagrilated toadthwacker down.",
    E666    =>  "Satan.",
    E951    =>  "Please reset your gymnozzle.",
);

func fatal ($msg = $ERRORS{$_[0]} when \%ERRORS)
{
    say STDERR $msg;
    exit 1;
}
```

So, remember this example? I said that it wasn't perfect: that it still had issues if someone passed us a code that wasn't in the list, or passed us nothing at all. Well, now that we know a little more, we can fix those things.

```
my %ERRORS =
(
    E100    =>  "The frobnobulator is all grizzed up.",
    E233    =>  "WARNING! Dagrilated toadthwacker down.",
    E666    =>  "Satan.",
    E951    =>  "Please reset your gymnozzle.",
);

func fatal ($msg! where { !/E\d{3}/ } = $ERRORS{$_[0]} when \%ERRORS)
{
    say STDERR $msg;
    exit 1;
}
```

Here's one instance where specifying that a parameter with a default value is required actually makes sense. And, in understanding how this works, it's also important to realize that the default value is applied *before* type and `where` constraints are checked.



So hopefully this has been a useful demonstration of how you can use Method::Signatures to make one of the easy things easy again. Damian Conway has said he's using it in "absolutely all [his] teaching." Perhaps this will someday be a standard part of Perl. In the meantime, it's ready for you right now.

The End

Questions?