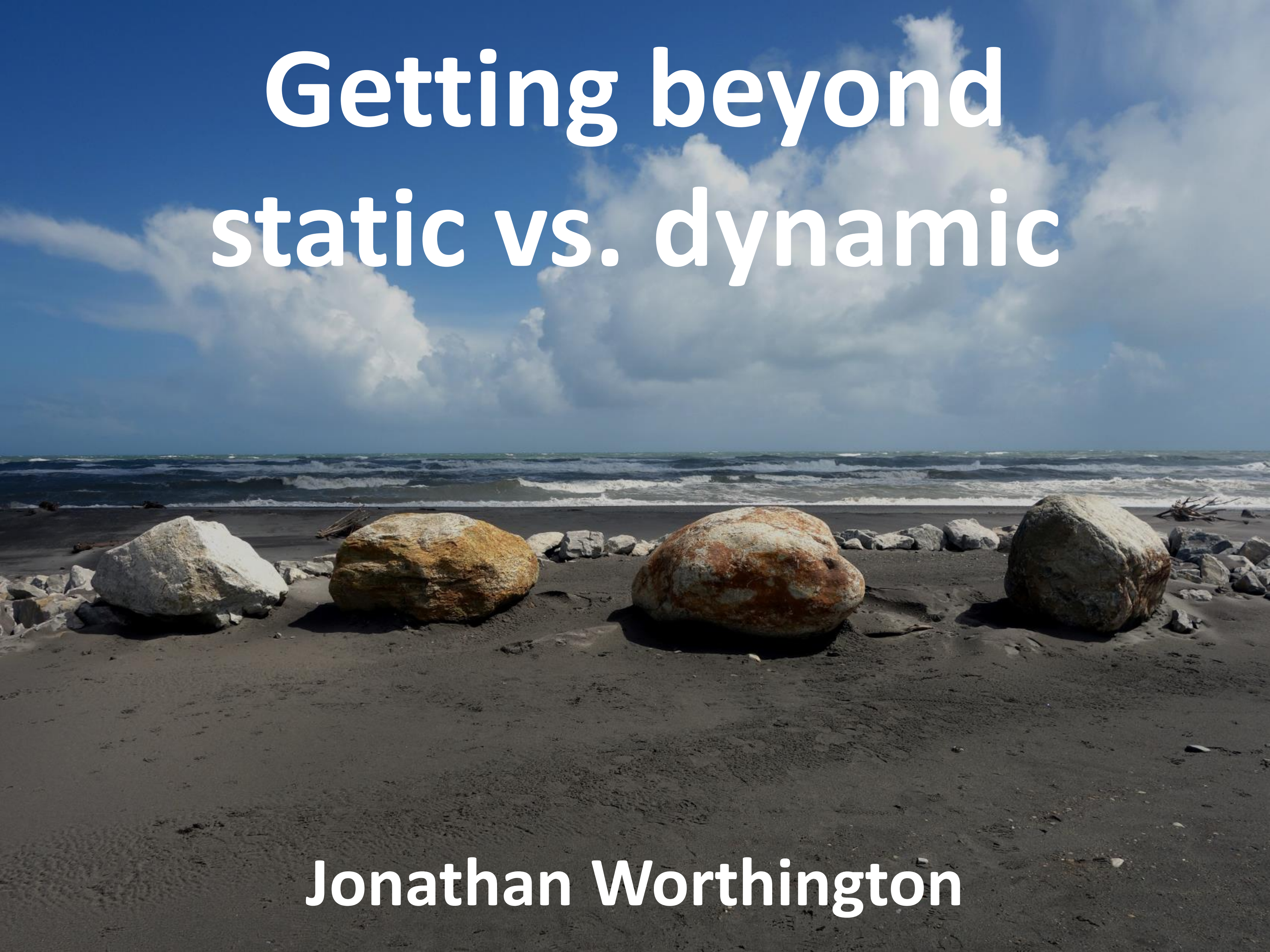
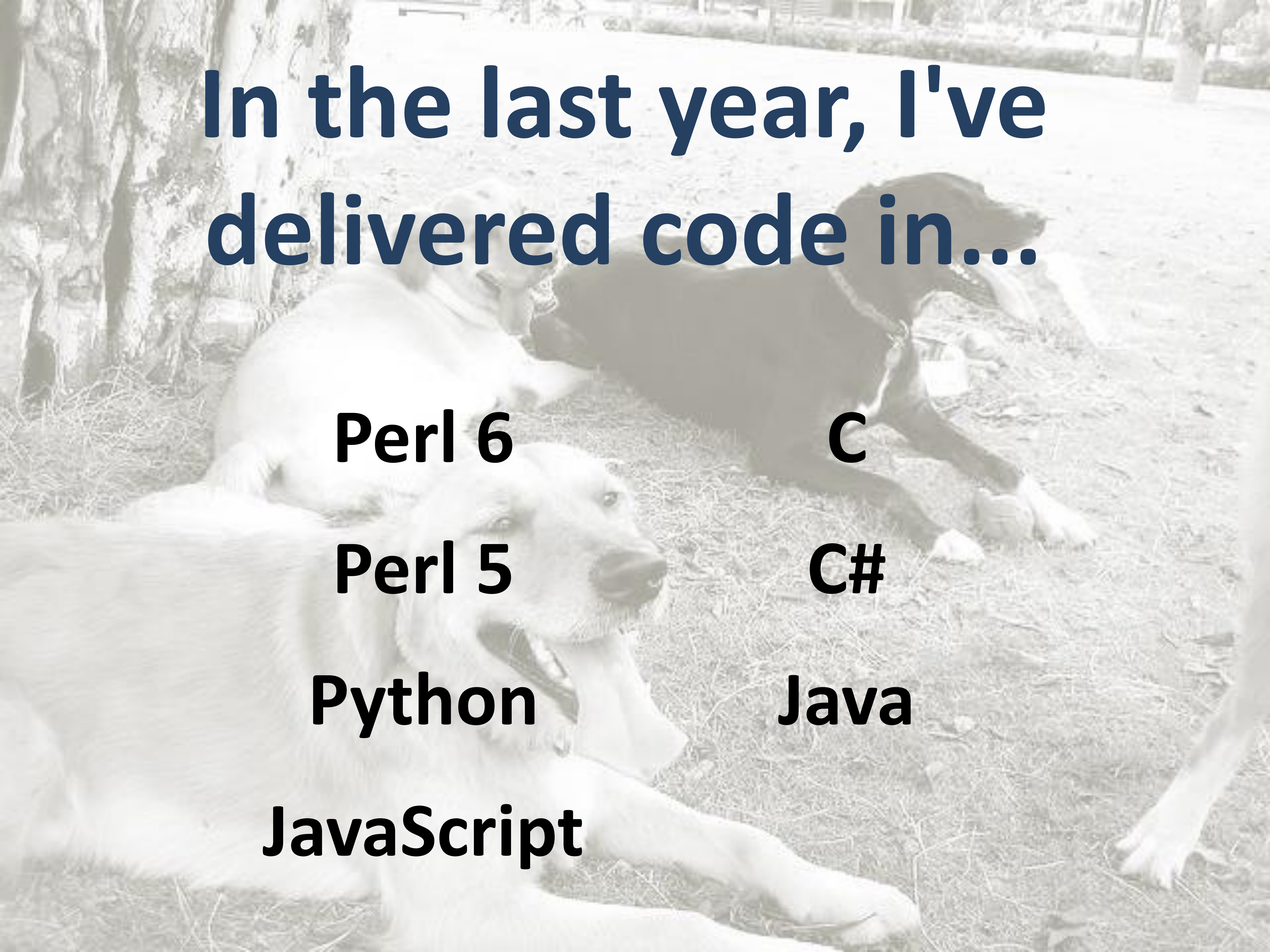
Getting beyond static vs. dynamic



Jonathan Worthington

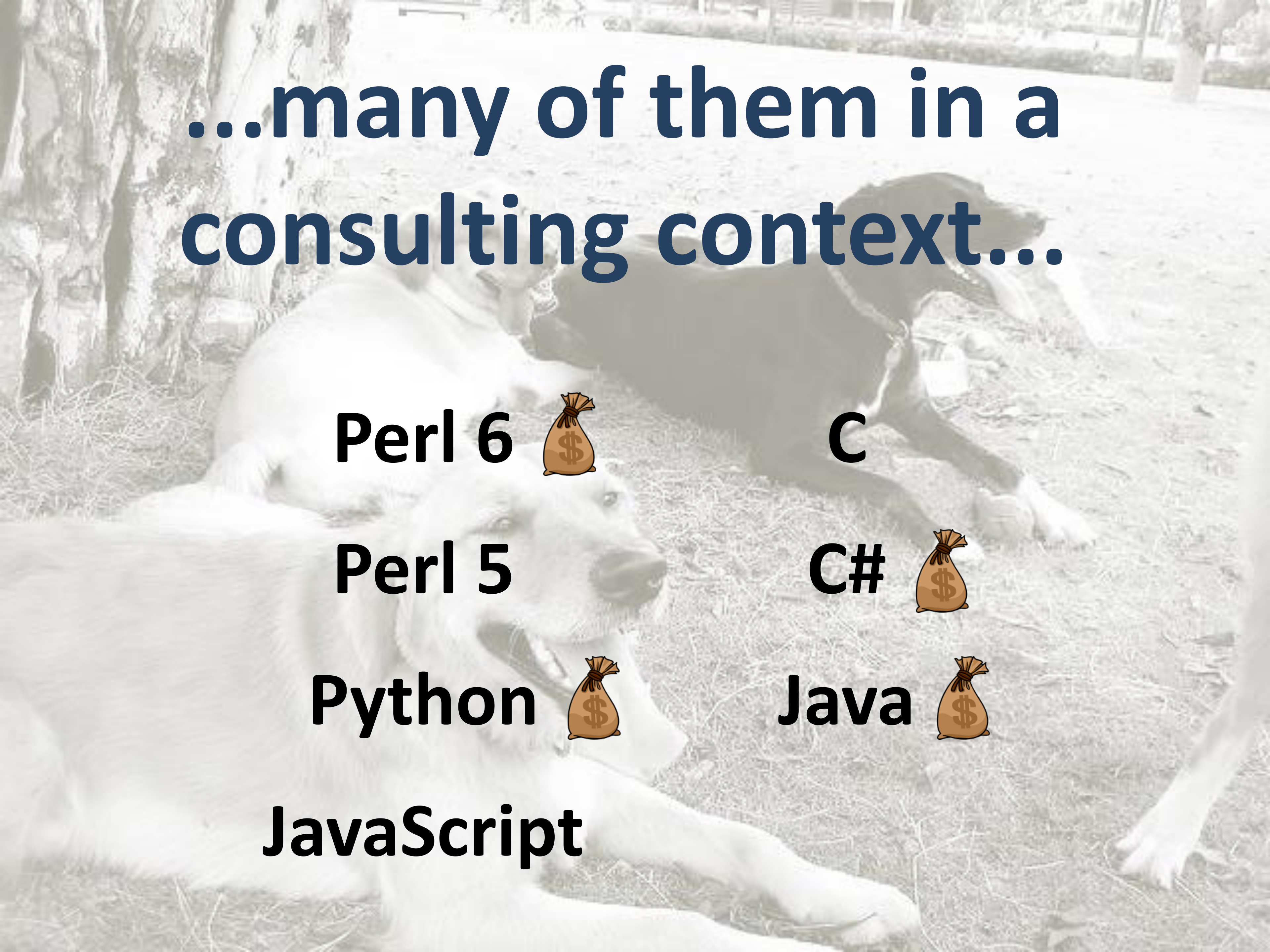
Hi.
I'm Jonathan.

I'm a polyglot programmer.



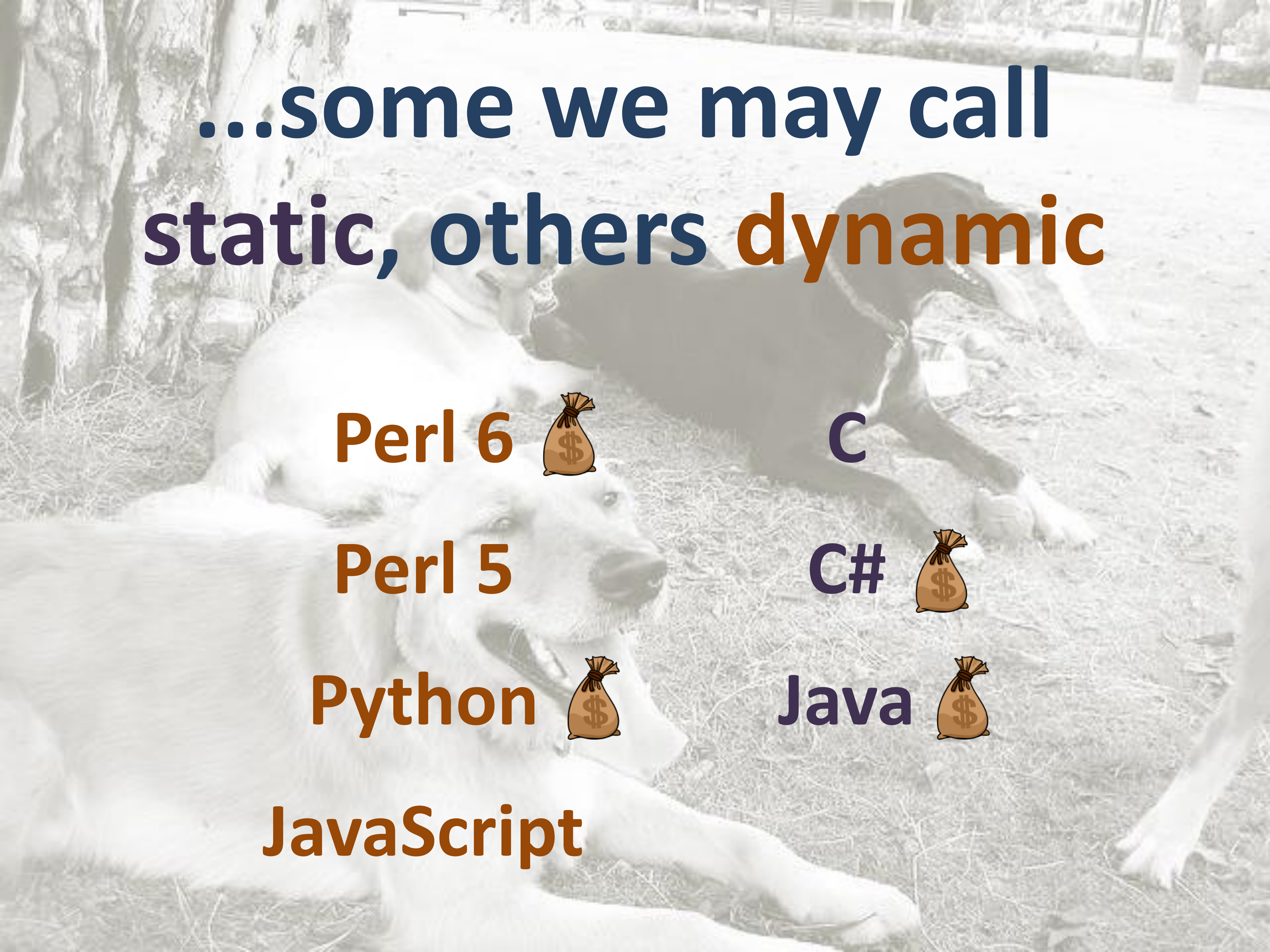
In the last year, I've delivered code in...

- Perl 6
- Perl 5
- Python
- JavaScript
- C
- C#
- Java



...many of them in a consulting context...

- Perl 6 💰
- Perl 5
- Python 💰
- JavaScript
- C
- C# 💰
- Java 💰



…some we may call static, others dynamic

Perl 6 💰

Perl 5

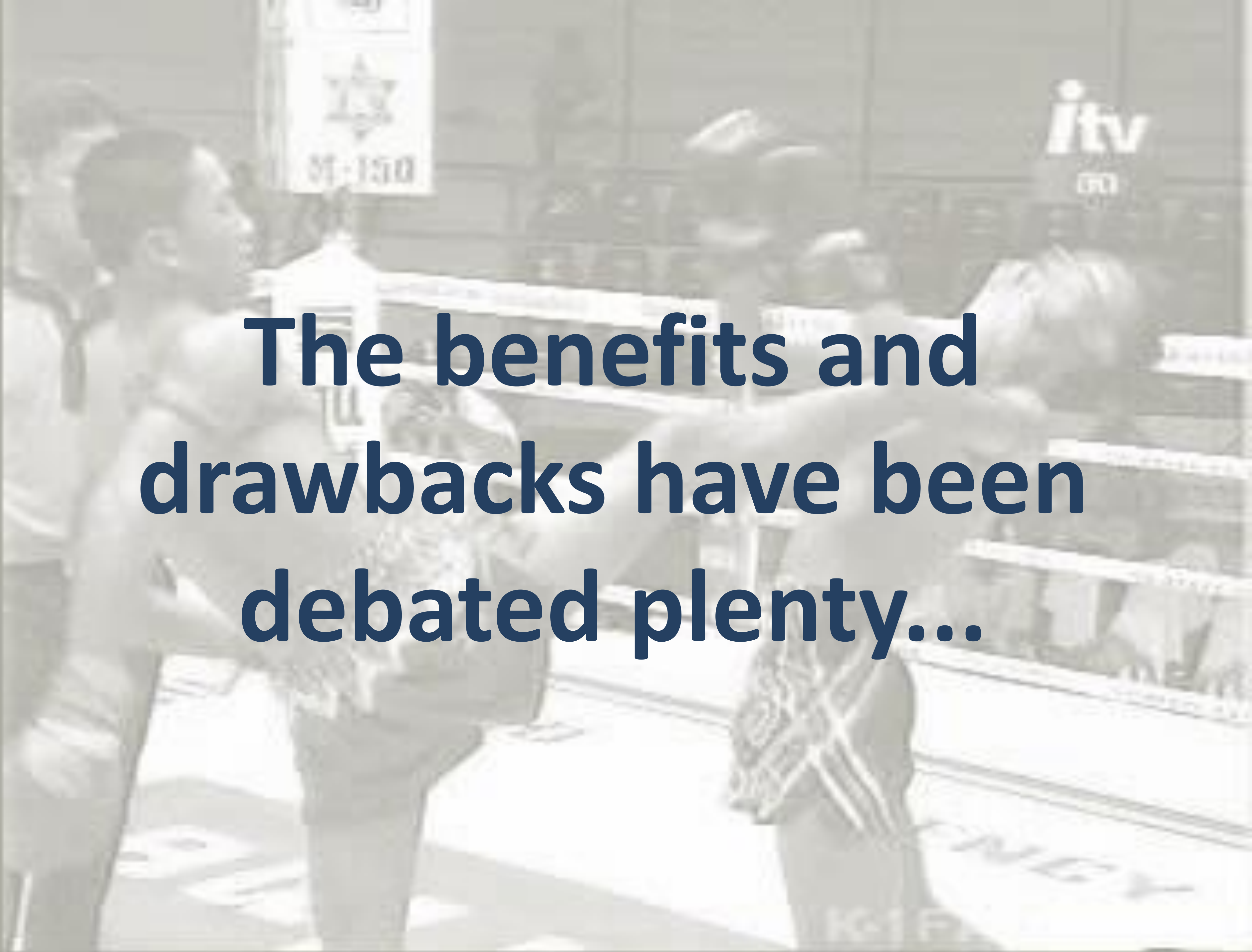
Python 💰

JavaScript

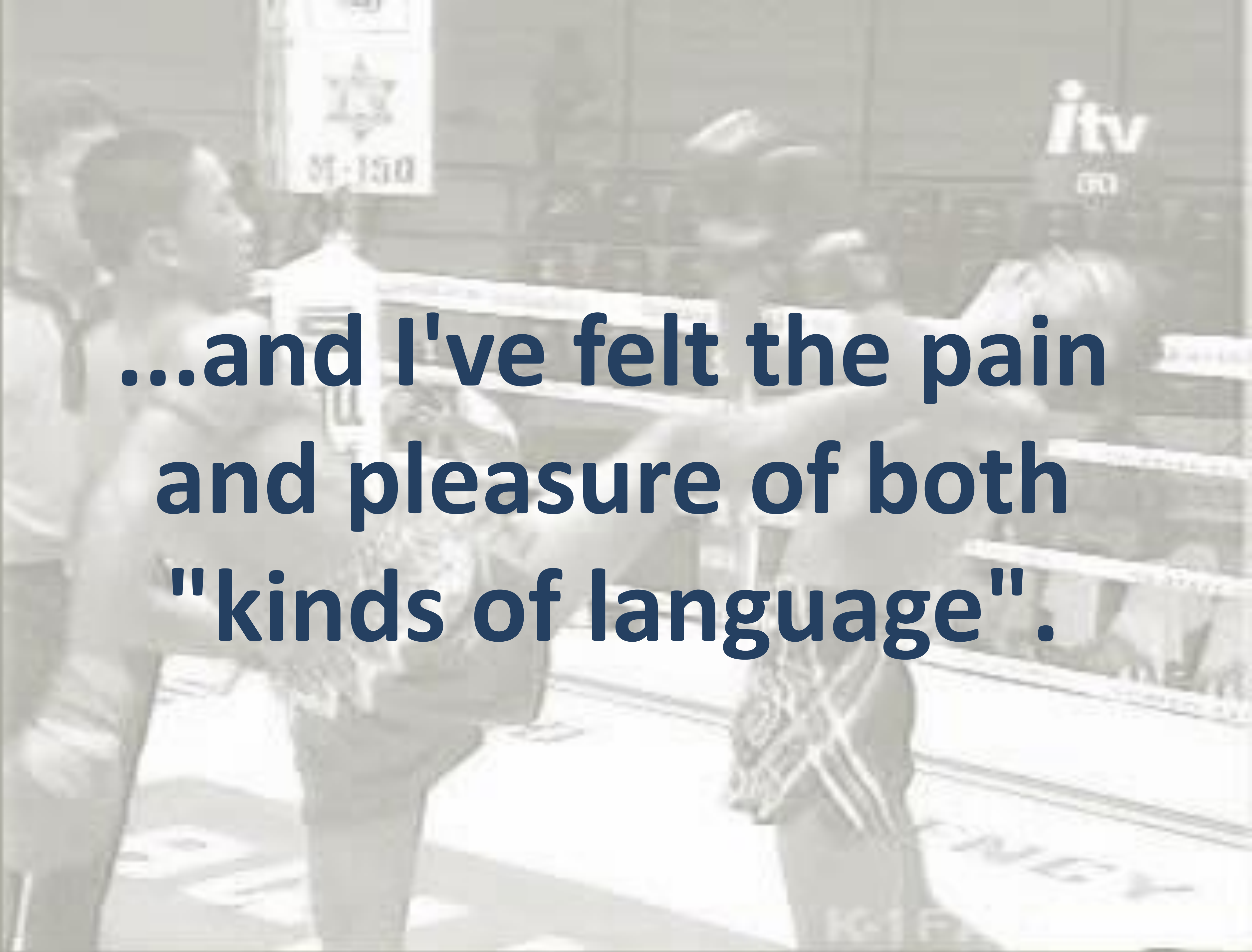
C

C# 💰

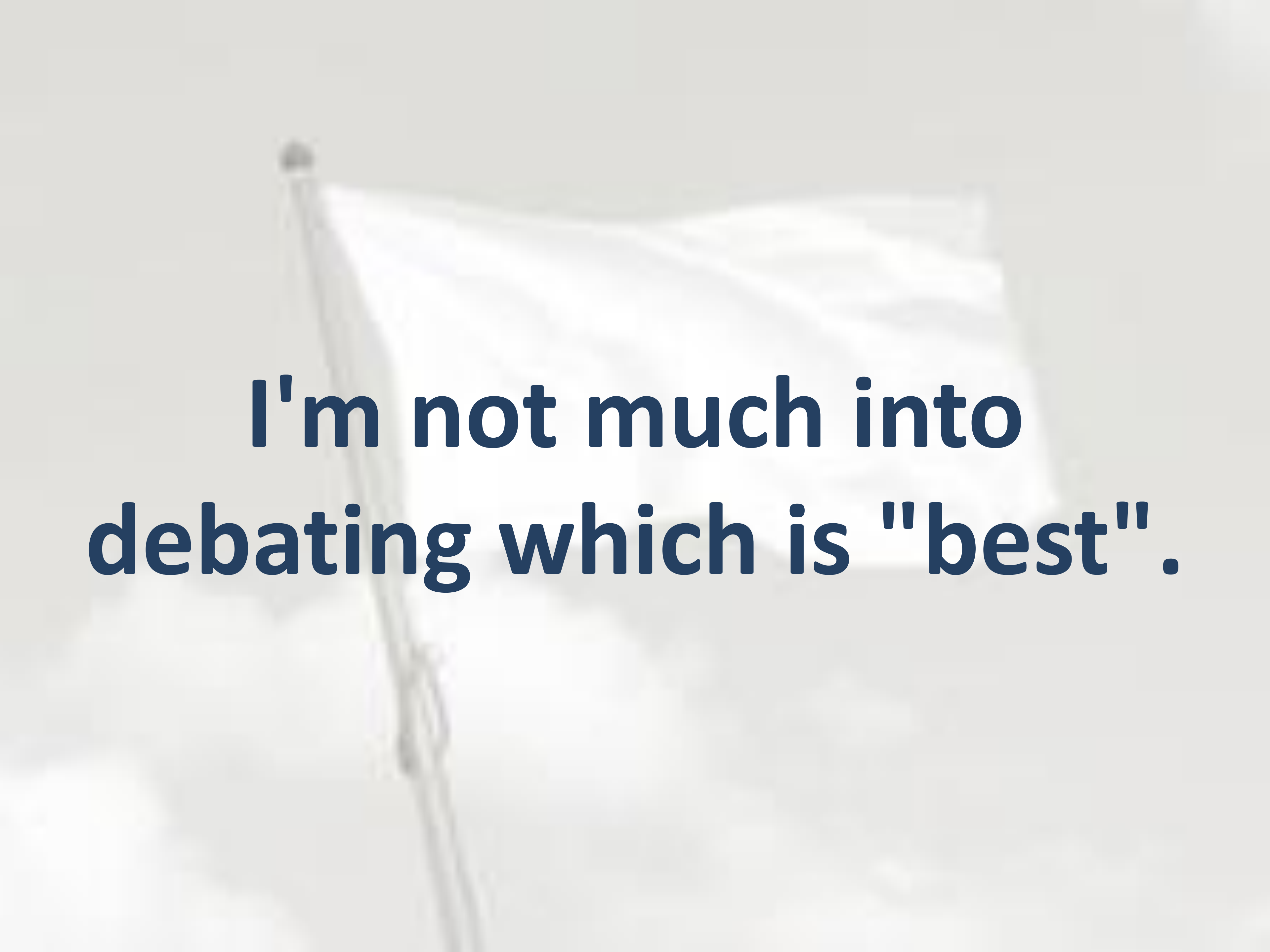
Java 💰



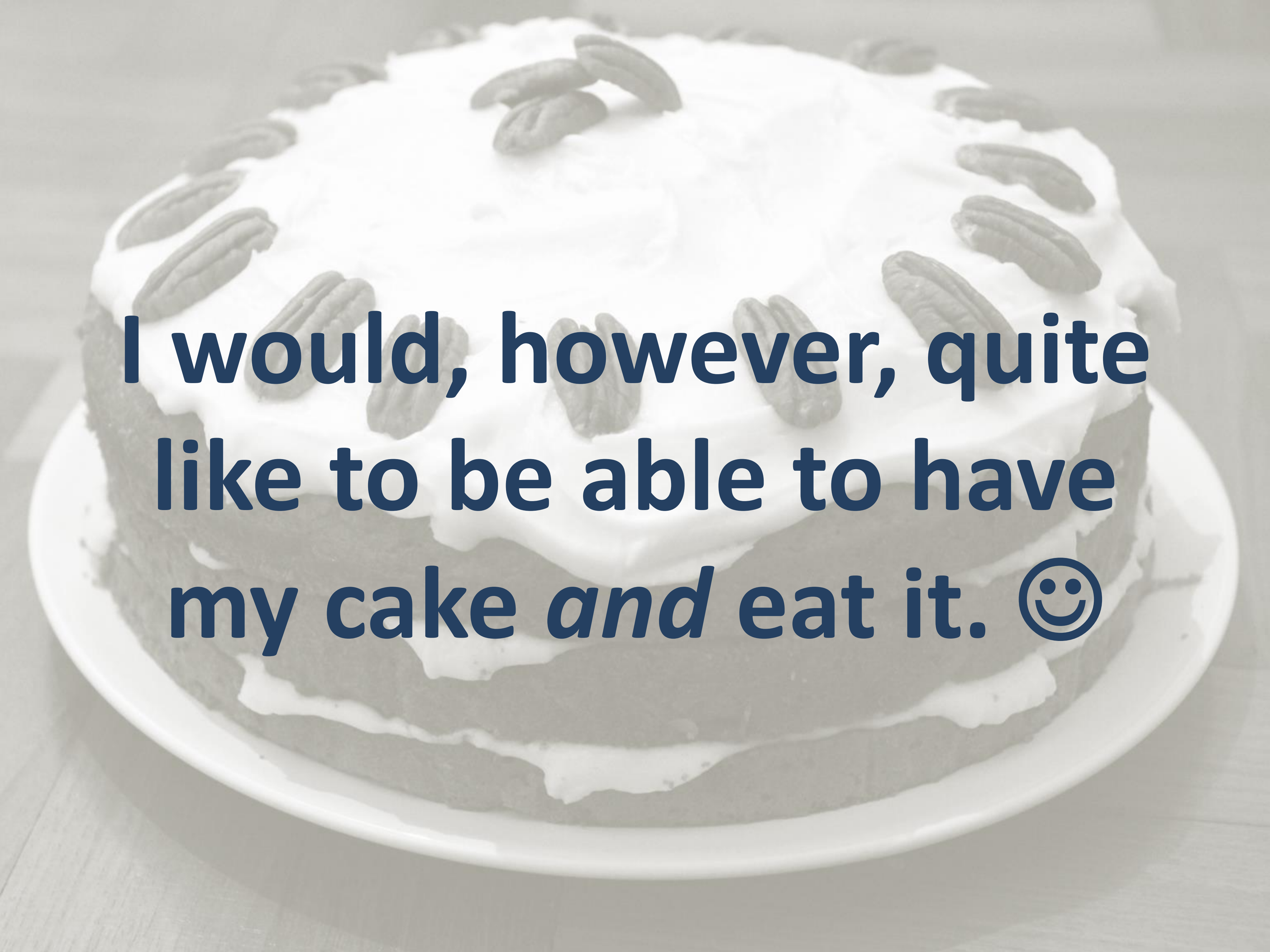
The benefits and drawbacks have been debated plenty…



...and I've felt the pain and pleasure of both "kinds of language".



I'm not much into debating which is "best".



I would, however, quite like to be able to have my cake *and* eat it. ☺

C#

```
class Program
{
    static void Main(string[] args)
    {
        var morning = "9am-12pm";
        Console.WriteLine("Opening hours:");
        Console.WriteLine(moroning);
    }
}
```

C#, etc.

```
class Program
{
    static void Main(string[] args)
    {
        var morning = "9am-12pm";
        Console.WriteLine("Opening hours:");
        Console.WriteLine(moroning);
    }
}
```

```
The name 'moroning' does not exist in the current
context
  -- The C# compiler
```

Python, Ruby, etc.

```
morning = "9am-12pm"
print("Opening hours:")
print(moroning)
```

```
morning = "9am-12pm"
puts "Opening times:"
puts moroning
```

Python, Ruby, etc.

```
morning = "9am-12pm"
print("Opening hours:")
print(moroning)
```

```
Opening hours:
Traceback (most recent call last):
  File "python", line 3, in <module>
NameError: name 'moroning' is not defined
```

```
morning = "9am-12pm"
puts "Opening times:"
puts moroning
```

```
Opening times:
undefined local variable or method `moroning' for
#<Context:0x00000002778f88>
(repl):3:in `initialize'
```

Perl 6

```
my $morning = "9am-12pm";
say "Opening hours:";
say $moroning;
```

Perl 6

```
my $morning = "9am-12pm";
say "Opening hours:";
say $moroning;
```

```
===SORRY!===
Variable '$moroning' is not declared. Did you
mean '$morning'?
at x.p6:3
------> say $moroning⏏;
```

Perl 6

```
my $morning = "9am-12pm";
say "Opening hours:";
say $moroning;
```

```
===SORRY!===
Variable '$moroning' is not declared. Did you
mean '$morning'?
at x.p6:3
------> say $moroning⏏;
```

The compiler says sorry for your moronic typo!

Perl 6

```
my $morning = "9am-12pm";
say "Opening hours:";
say $moroning;
```

```
===SORRY!===
Variable '$moroning' is not declared. Did you
mean '$morning'?
at x.p6:3
------> say $moroning⏏;
```

And it suggests what you probably meant to type…

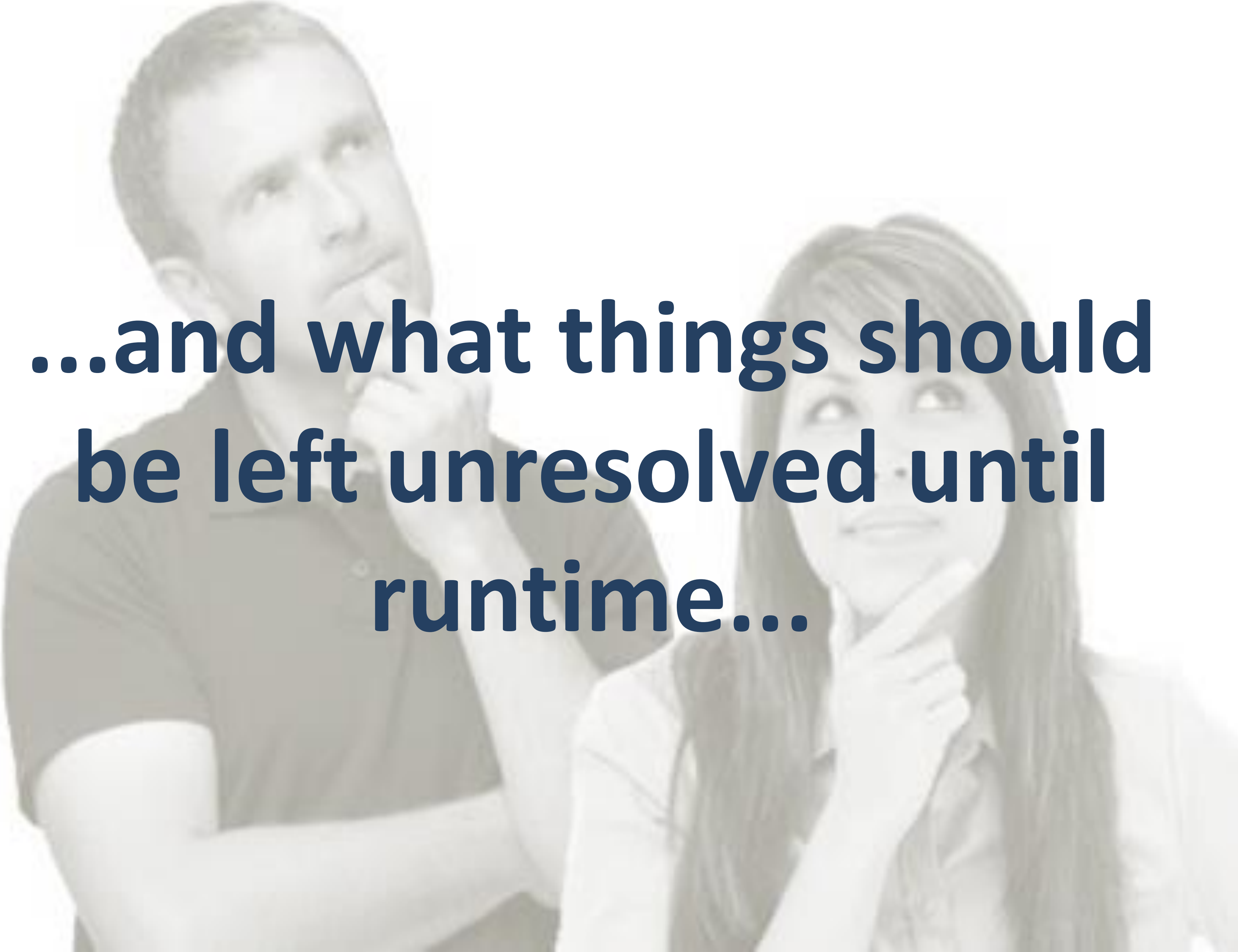
Perl 6

```
my $morning = "9am-12pm";
say "Opening hours:";
say $moroning;
```

```
===SORRY!===
Variable '$moroning' is not declared. Did you
mean '$morning'?
at x.p6:3
------> say $moroning⏏;
```

...and points out precisely where the problem is.

In Perl 6, we've thought carefully about what it's possible to know at compile time...



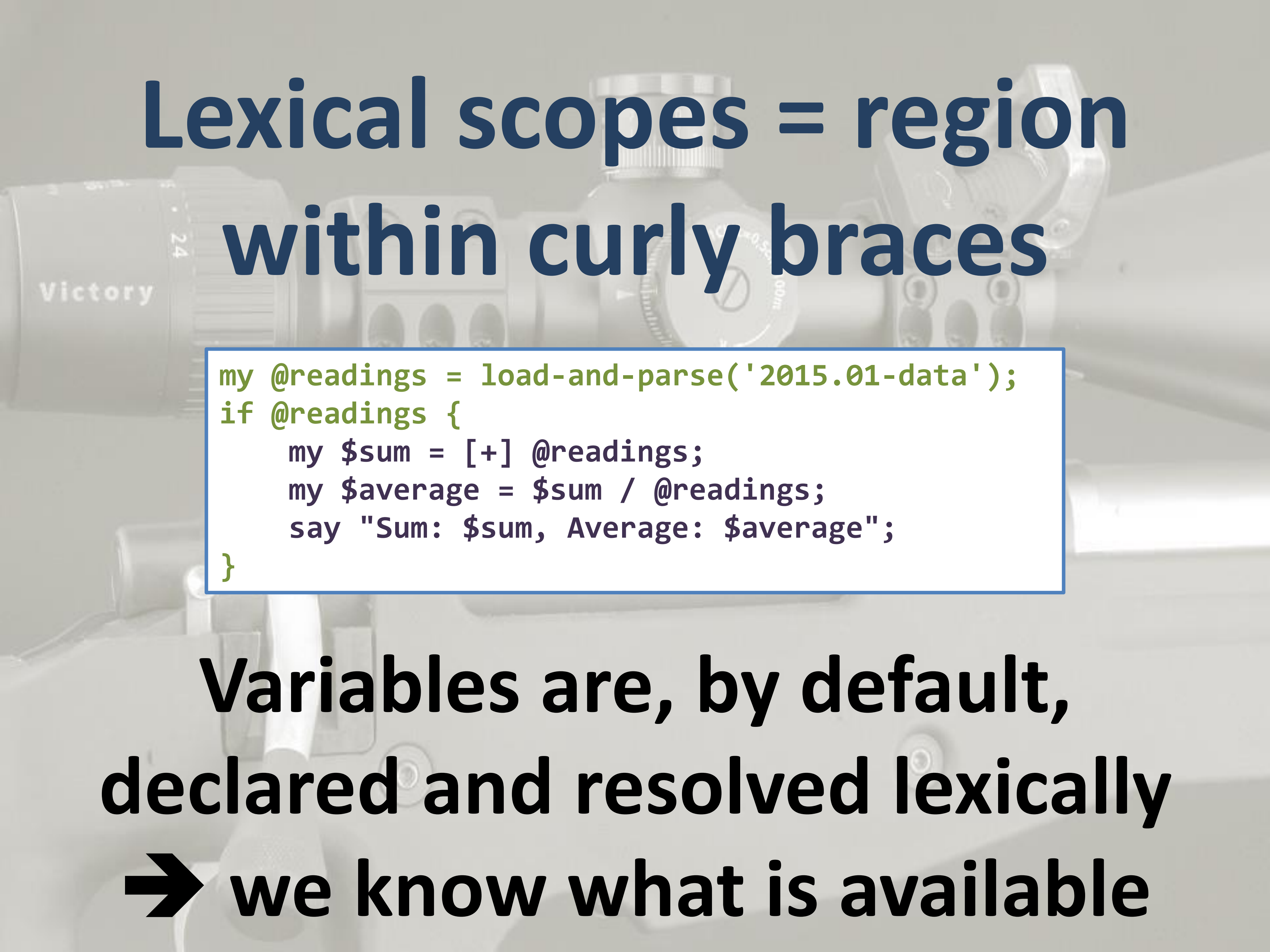
…and what things should be left unresolved until runtime…

...and made sure there are "escape valves" for the compile-time things.

Perl 6

❤

Lexical Scoping



Lexical scopes = region within curly braces

```
my @readings = load-and-parse('2015.01-data');
if @readings {
    my $sum = [+] @readings;
    my $average = $sum / @readings;
    say "Sum: $sum, Average: $average";
}
```

Variables are, by default, declared and resolved lexically ➔ we know what is available

Subroutines

Subs and calls to them are also lexically scoped by default

```
sub abbreviate($text, $chars) {
    $text.chars > $chars
        ?? $text.substr(0, $chars) ~ "..."
        !! $text
}
say abreviate("Long string is really long", 10);
```

```
===SORRY!===
Undeclared routine:
    abreviate used at line 6. Did you mean
    'abbreviate'?
```

Subroutines

Compiler knows what you call, so can check the arguments

```
sub abbreviate($text, $chars) {
    $text.chars > $chars
        ?? $text.substr(0, $chars) ~ "..."
        !! $text
}
say abbreviate("Long string is really long");
```

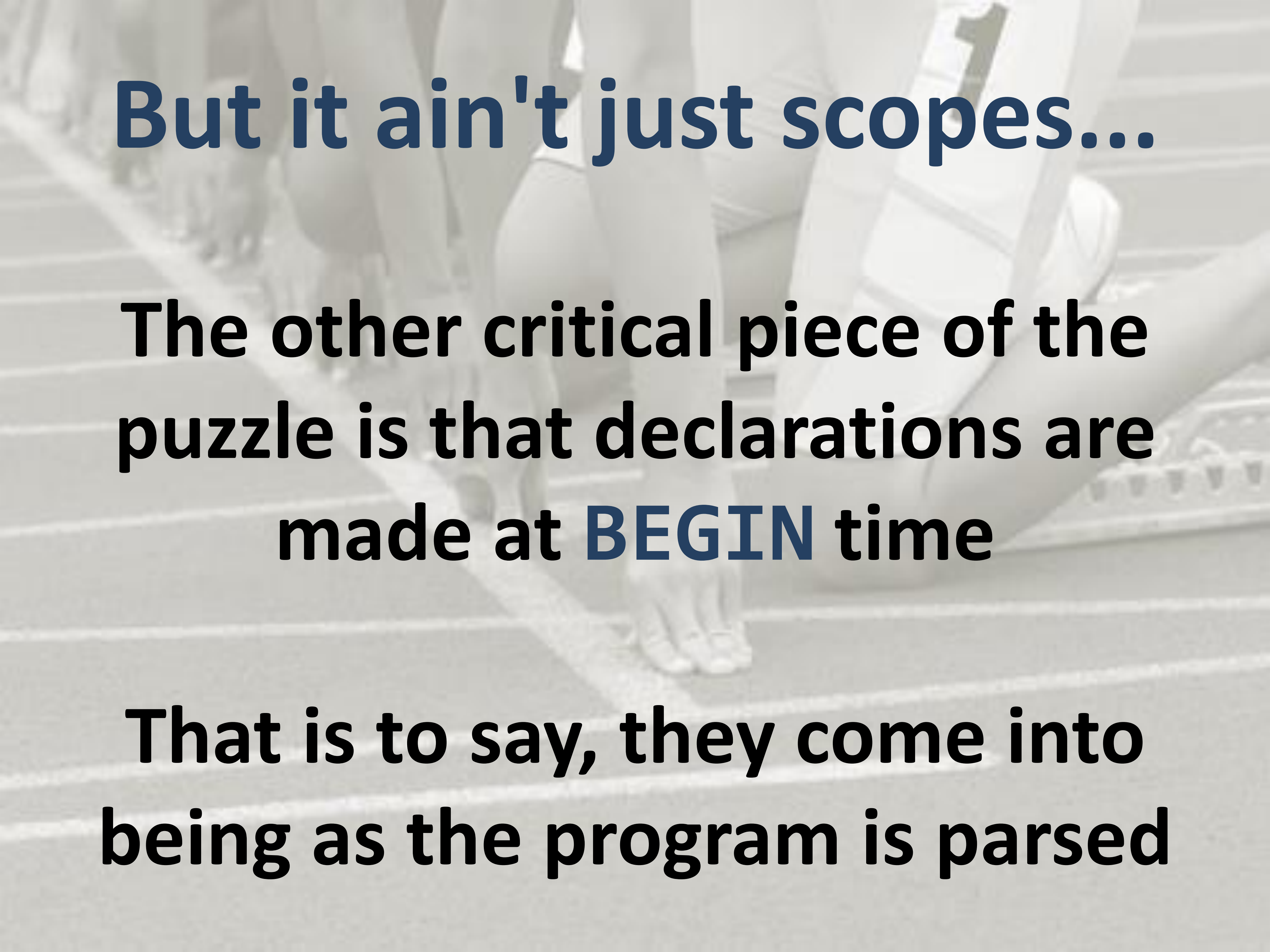
```
===SORRY!===
Calling 'abbreviate' will never work with
argument types (str)
    Expected: :(Any $text, Any $chars)
```

Subroutines

It can even do some basic type analysis on the arguments

```
sub abbreviate(Str $text, Int $chars) {
    $text.chars > $chars
        ?? $text.substr(0, $chars) ~ "..."
        !! $text
}
say abbreviate(10, "Long string is really long");
```

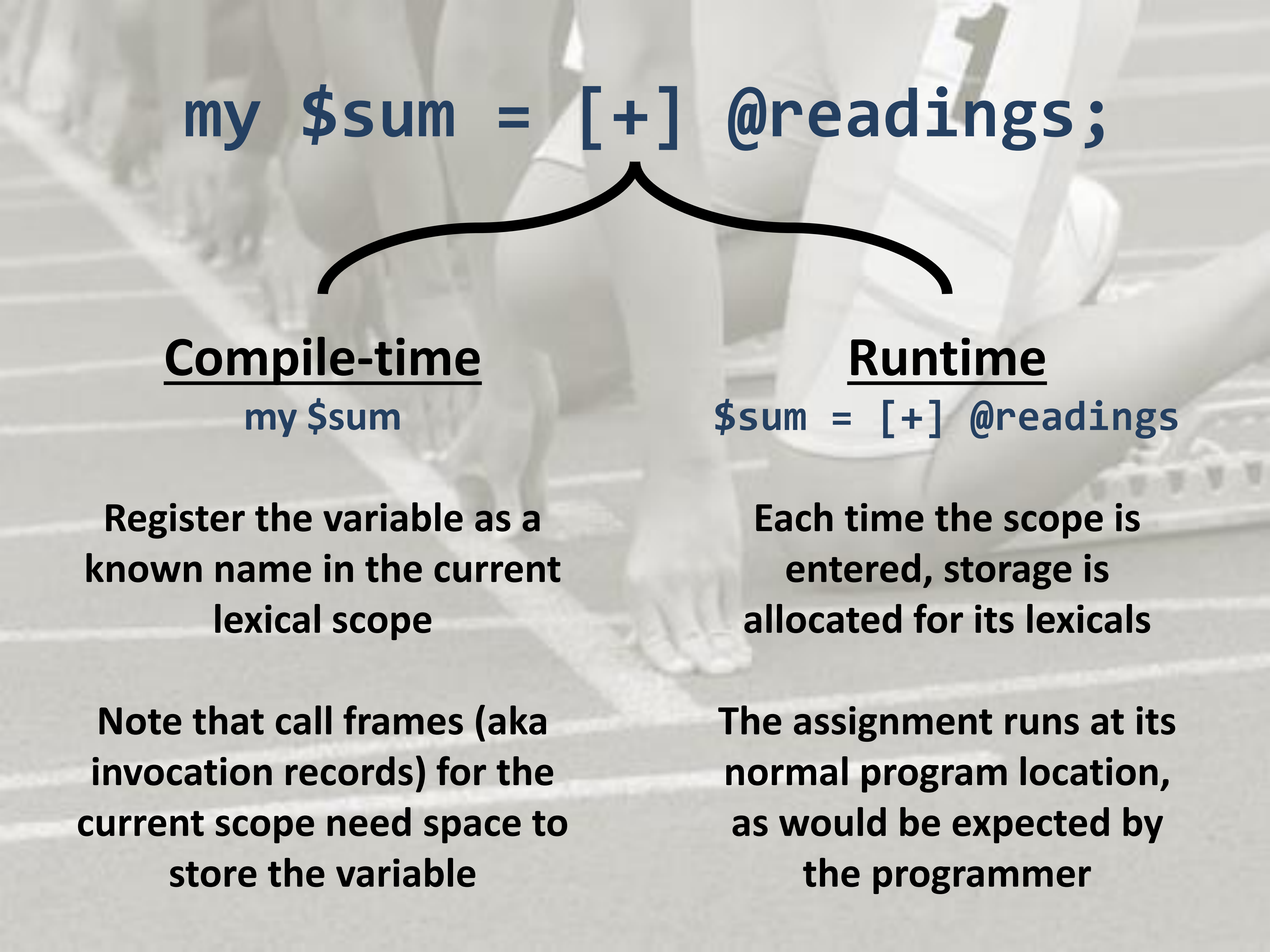
```
===SORRY!===
Calling 'abbreviate' will never work with
argument types (Int, Str)
    Expected: :(Str $text, Int $chars)
```



But it ain't just scopes...

The other critical piece of the puzzle is that declarations are made at `BEGIN` time

That is to say, they come into being as the program is parsed



```
my $sum = [+] @readings;
```

Compile-time

```
my $sum
```

Register the variable as a known name in the current lexical scope

Note that call frames (aka invocation records) for the current scope need space to store the variable

Runtime

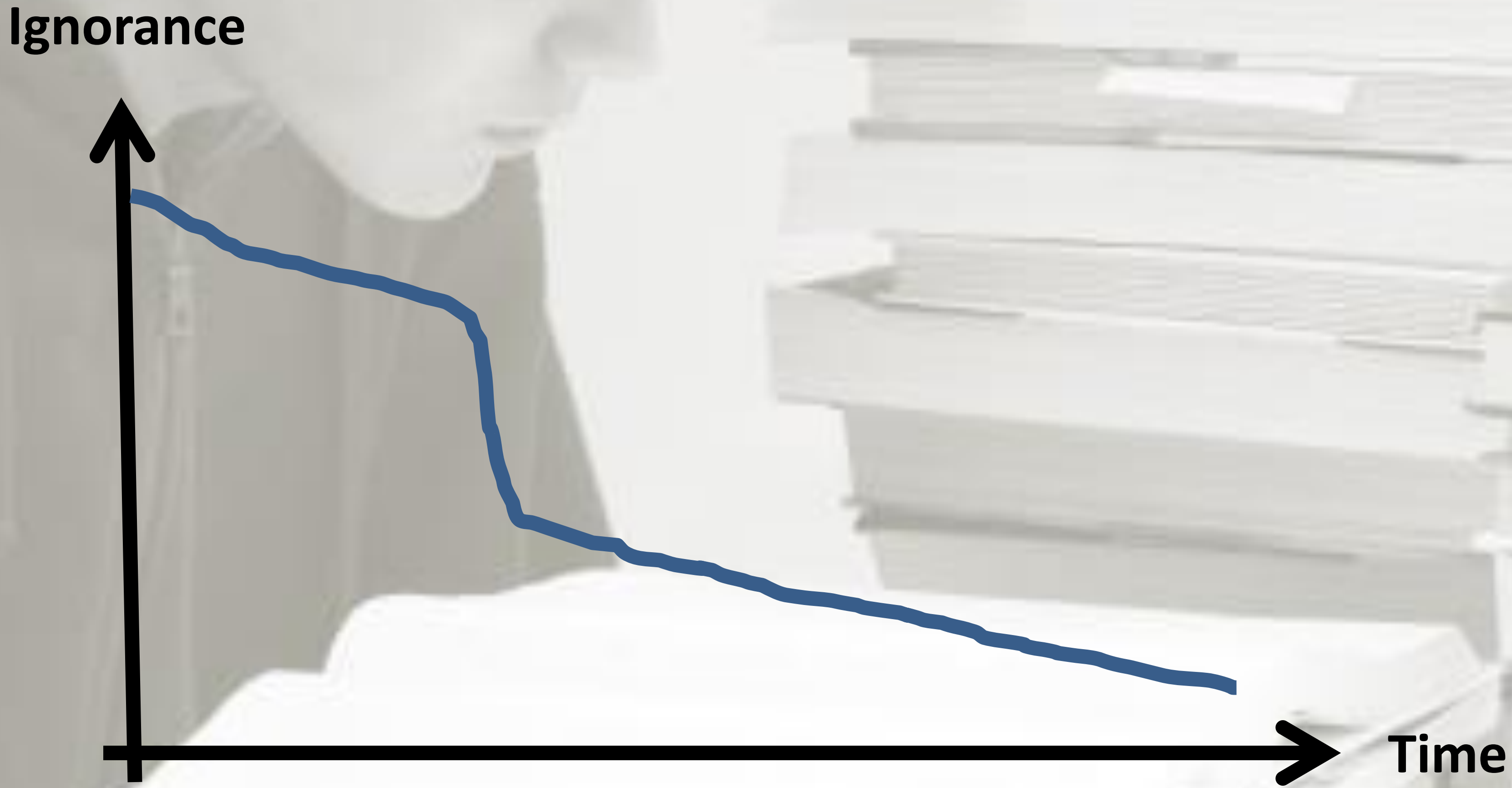
```
$sum = [+] @readings
```

Each time the scope is entered, storage is allocated for its lexicals

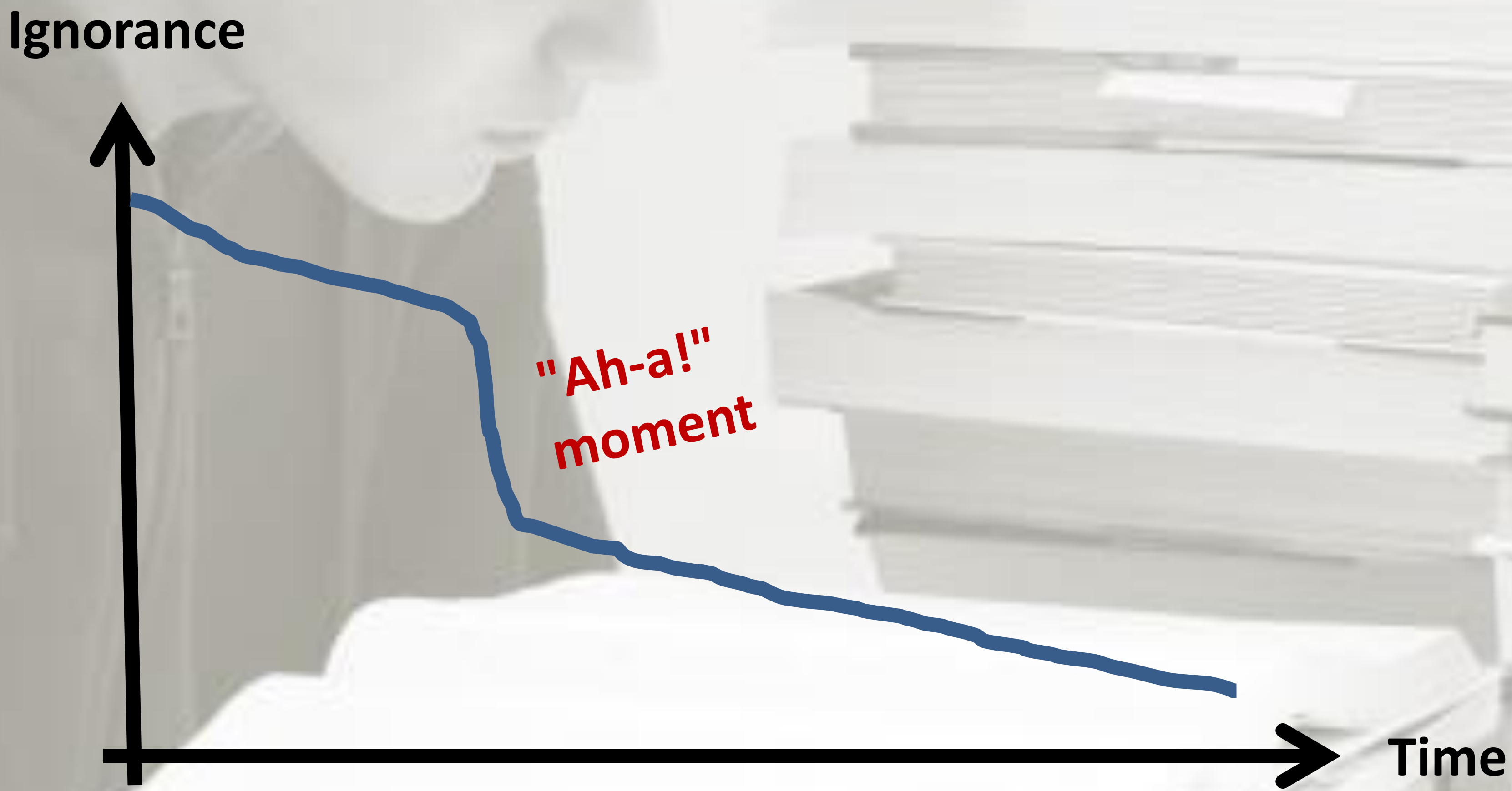
The assignment runs at its normal program location, as would be expected by the programmer

A historical aside…

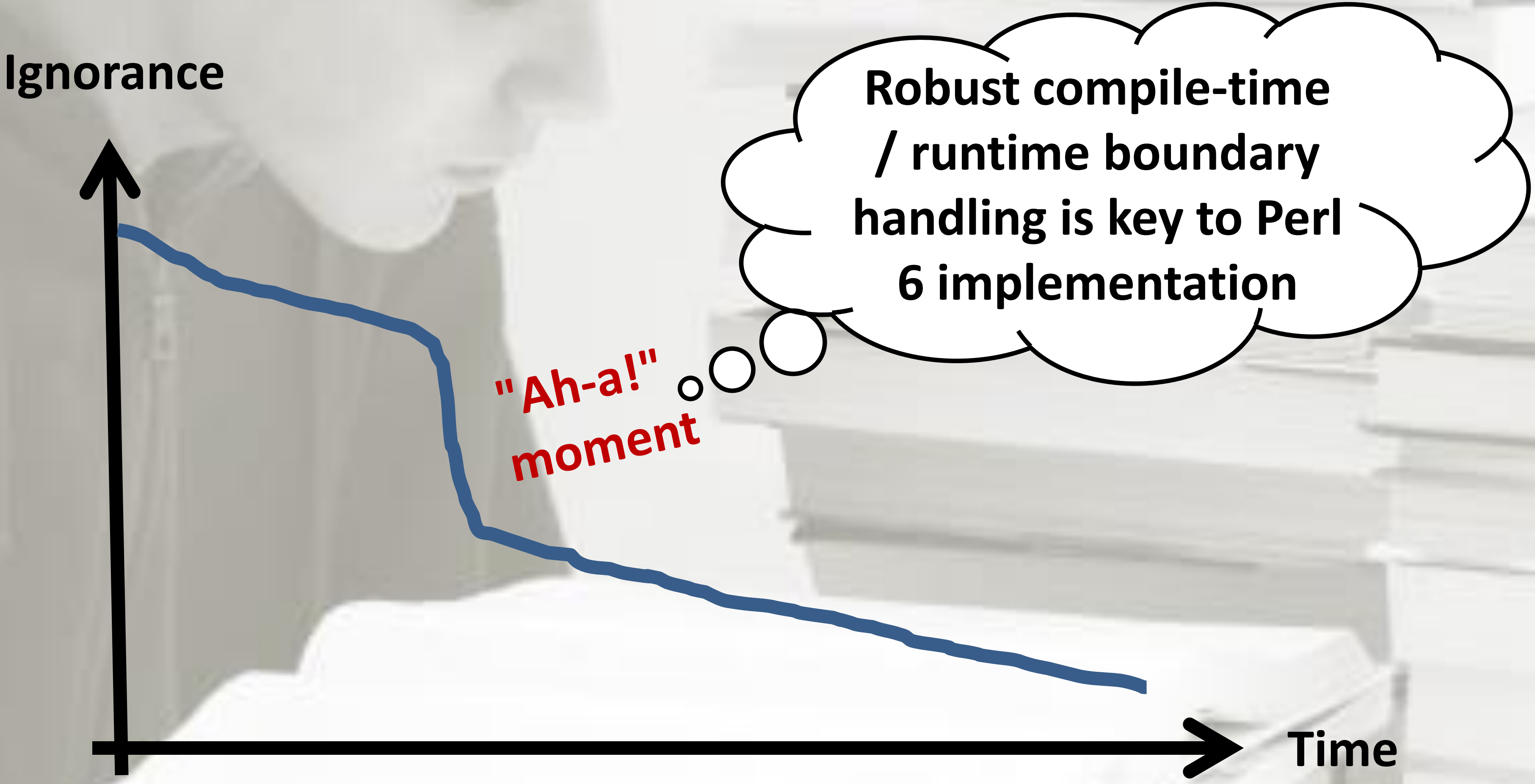
The ignorance curve



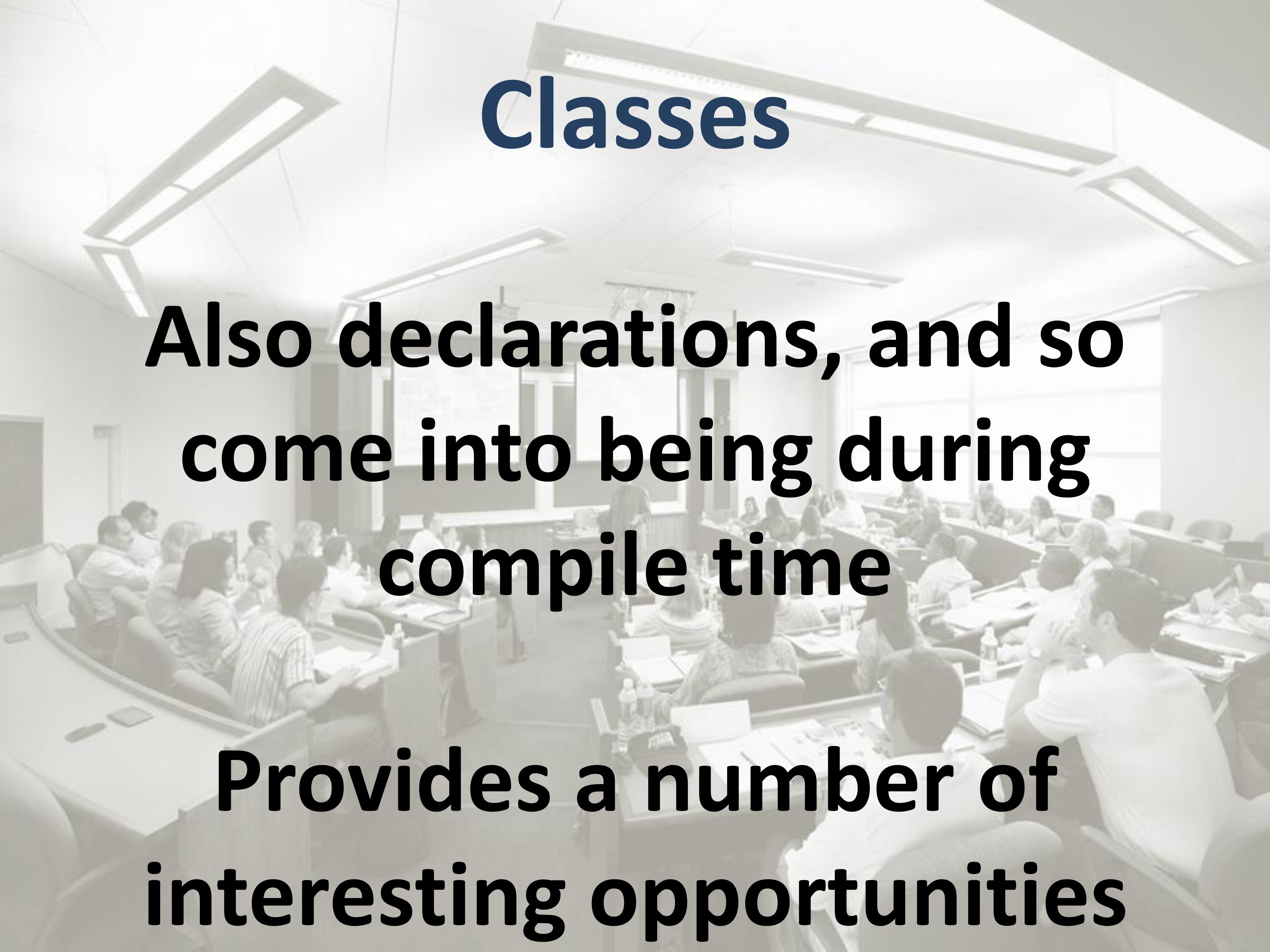
The ignorance curve



The ignorance curve

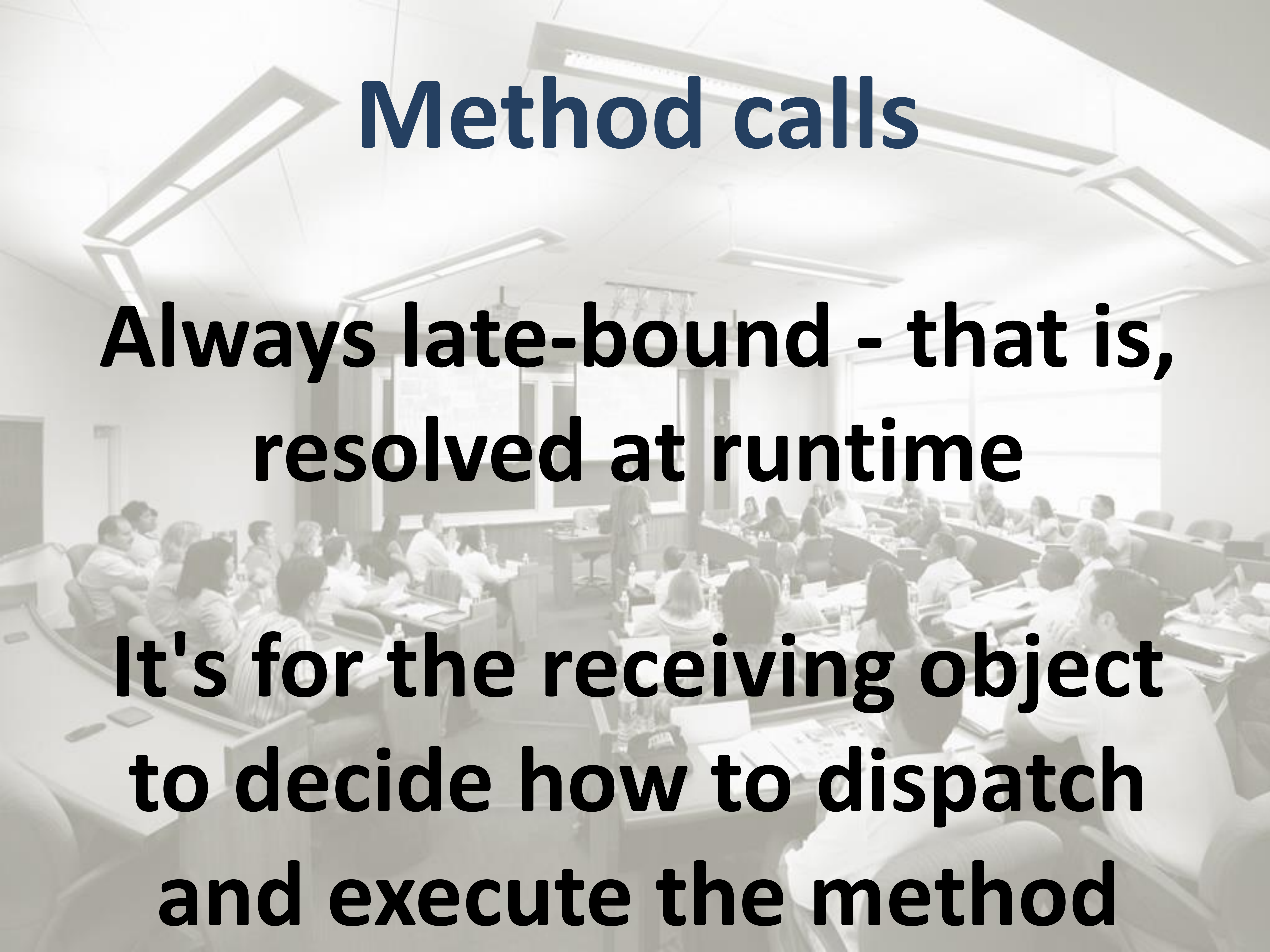


Classes



Also declarations, and so come into being during compile time

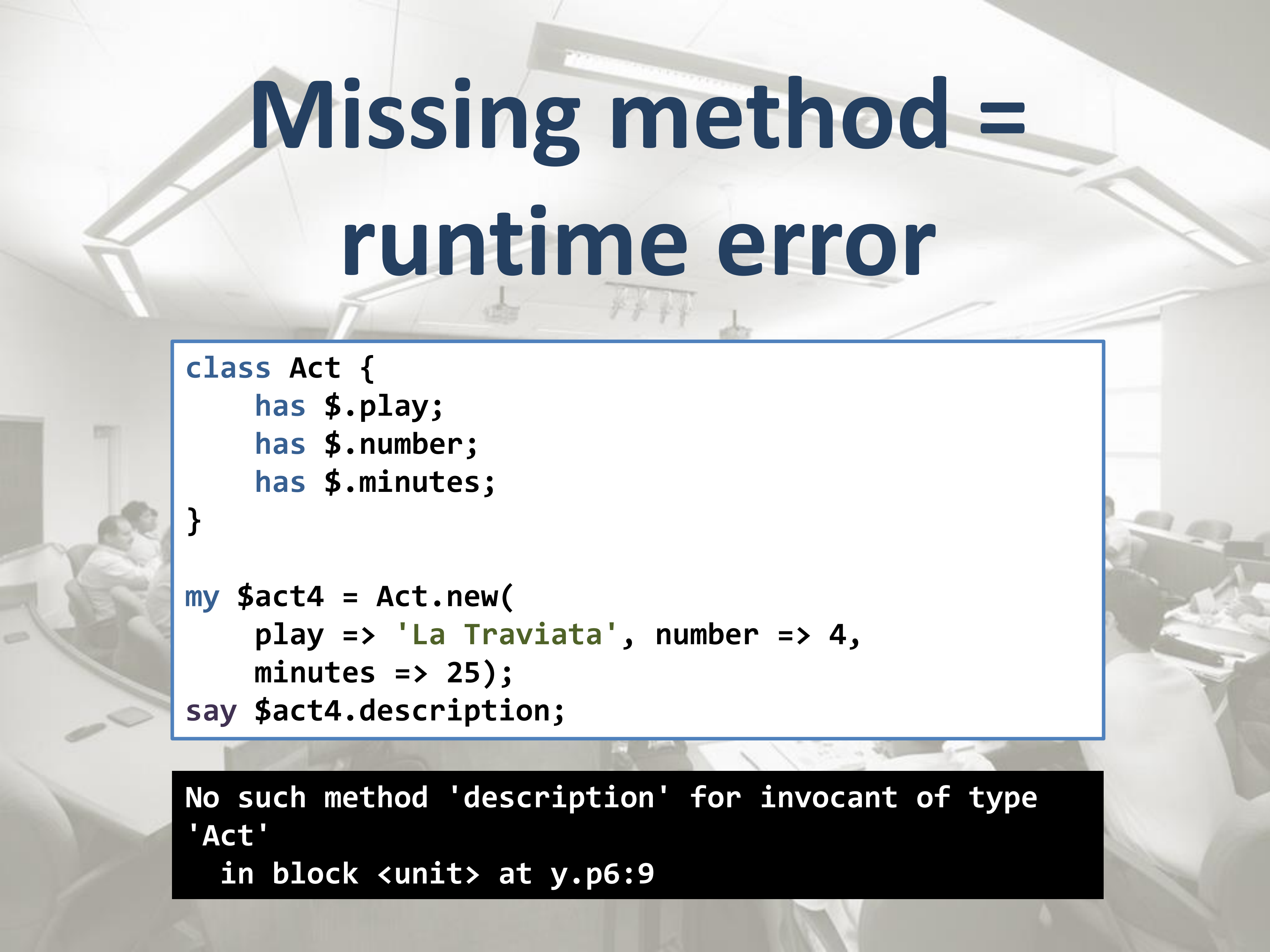
Provides a number of interesting opportunities



Method calls

Always late-bound - that is, resolved at runtime

It's for the receiving object to decide how to dispatch and execute the method



Missing method = runtime error

```
class Act {
    has $.play;
    has $.number;
    has $.minutes;
}

my $act4 = Act.new(
    play => 'La Traviata', number => 4,
    minutes => 25);
say $act4.description;
```

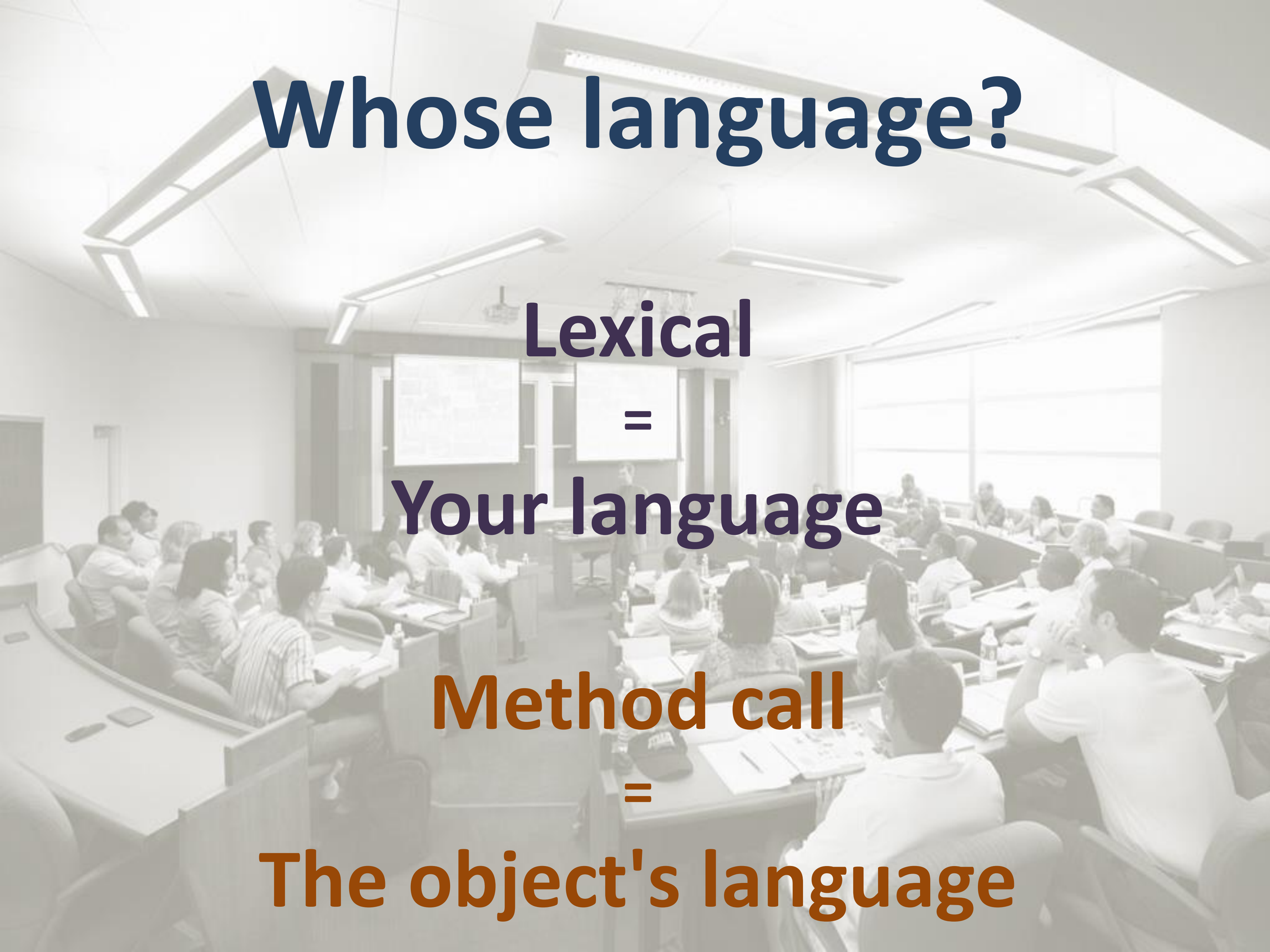
```
No such method 'description' for invocant of type
'Act'
  in block <unit> at y.p6:9
```

Handling missing methods

```
class Html {
    method FALLBACK($tag, *@kids, *%attrs) {
        my $kids-str = @kids.join('');
        my $attr-str = %attrs.fmt(' %s="%s"', '');
        "<$tag" ~ $attr-str ~ ">" ~ $kids-str ~ "</$tag>"
    }
}

say Html.p(
    'Omg, ',
    Html.a('a link', href => 'http://perl6.org/'),
    '!'
);
```

```
<p>Omg, <a href="http://perl6.org/">a link</a>!</p>
```

Whose language?



Lexical

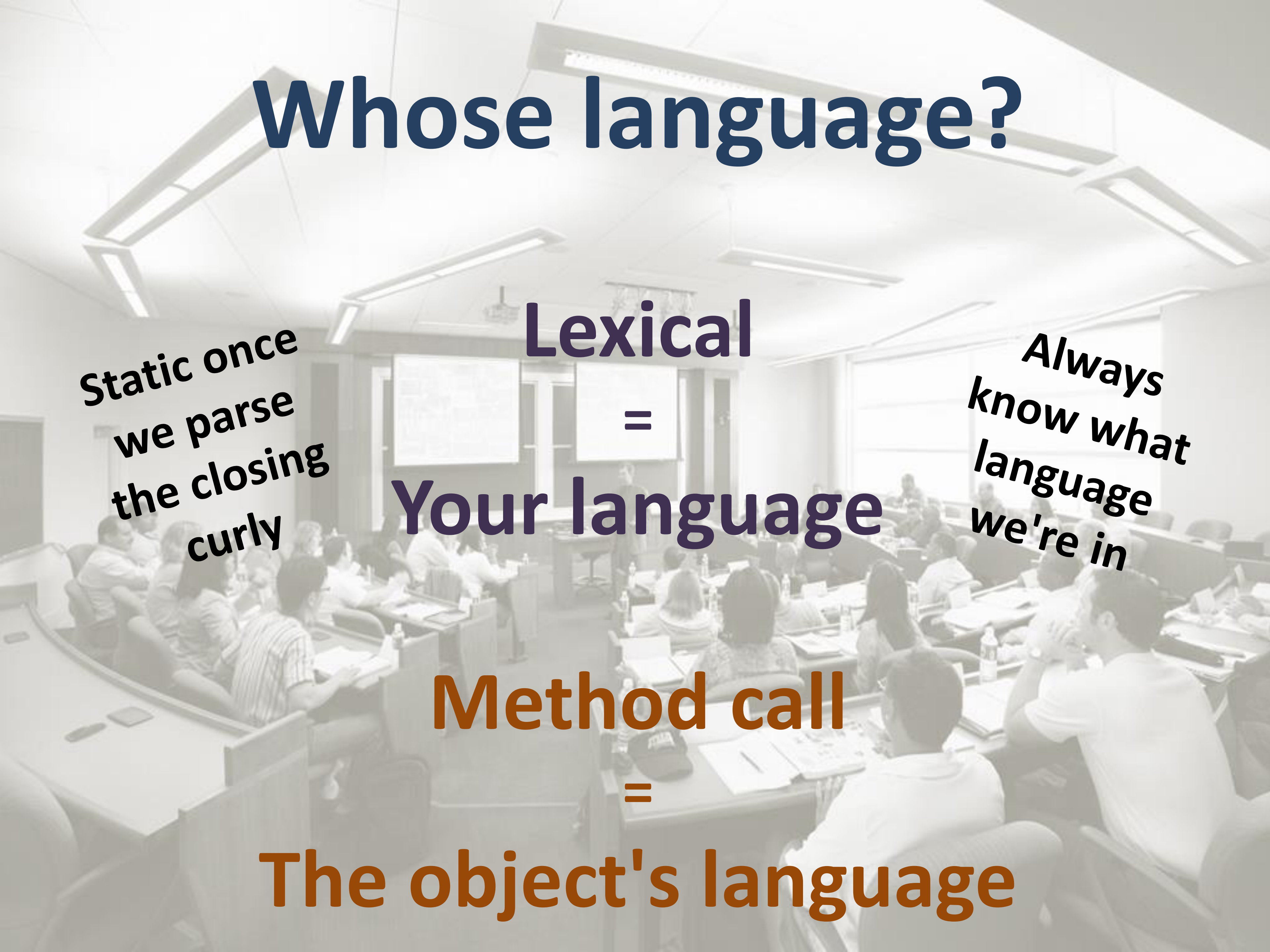
=

Your language

Method call

=

The object's language



Whose language?

Lexical

=

Your language

Static once we parse the closing curly

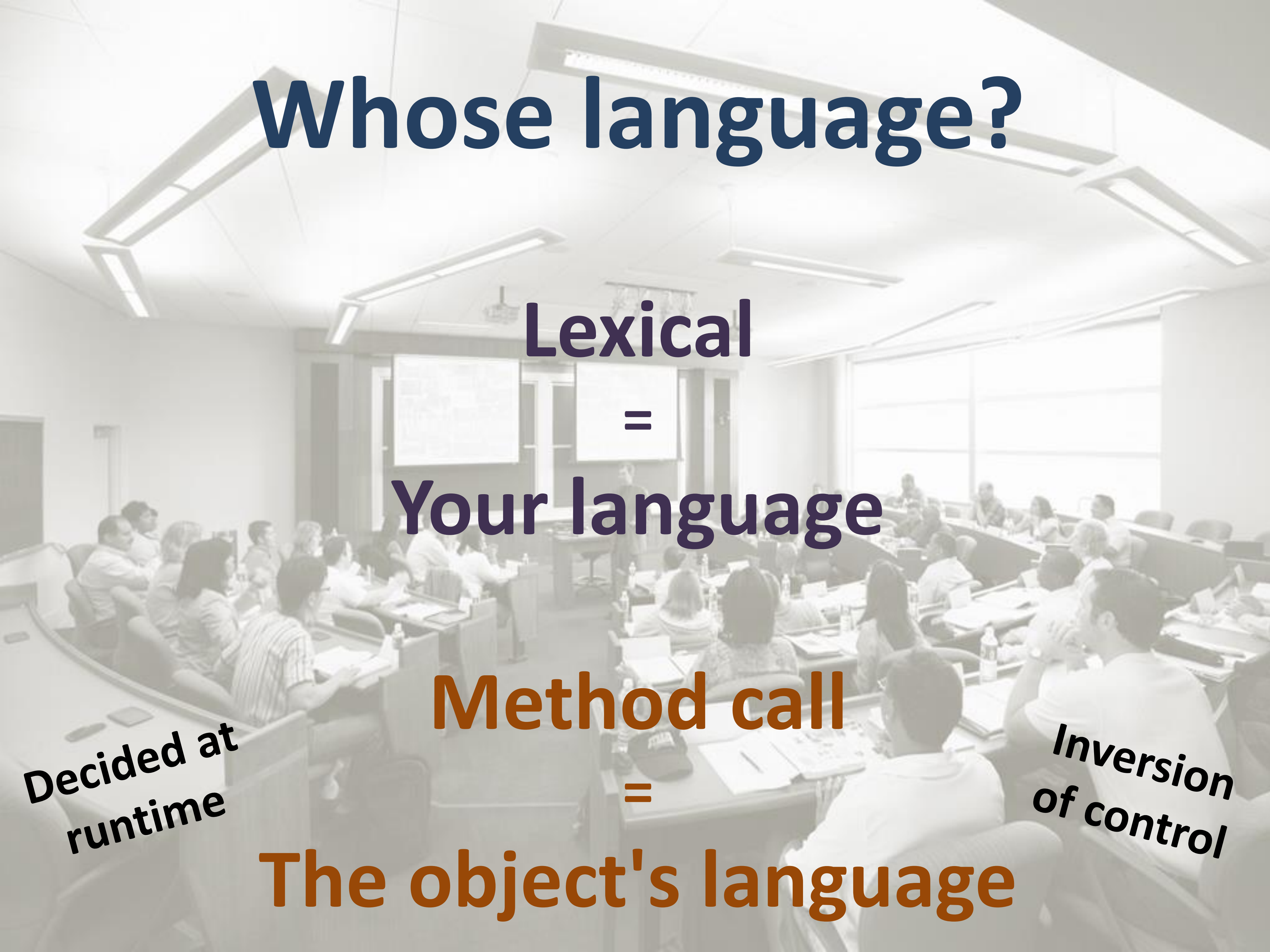
Always know what language we're in

Method call

=

The object's language

Whose language?



Lexical

=

Your language

Method call

=

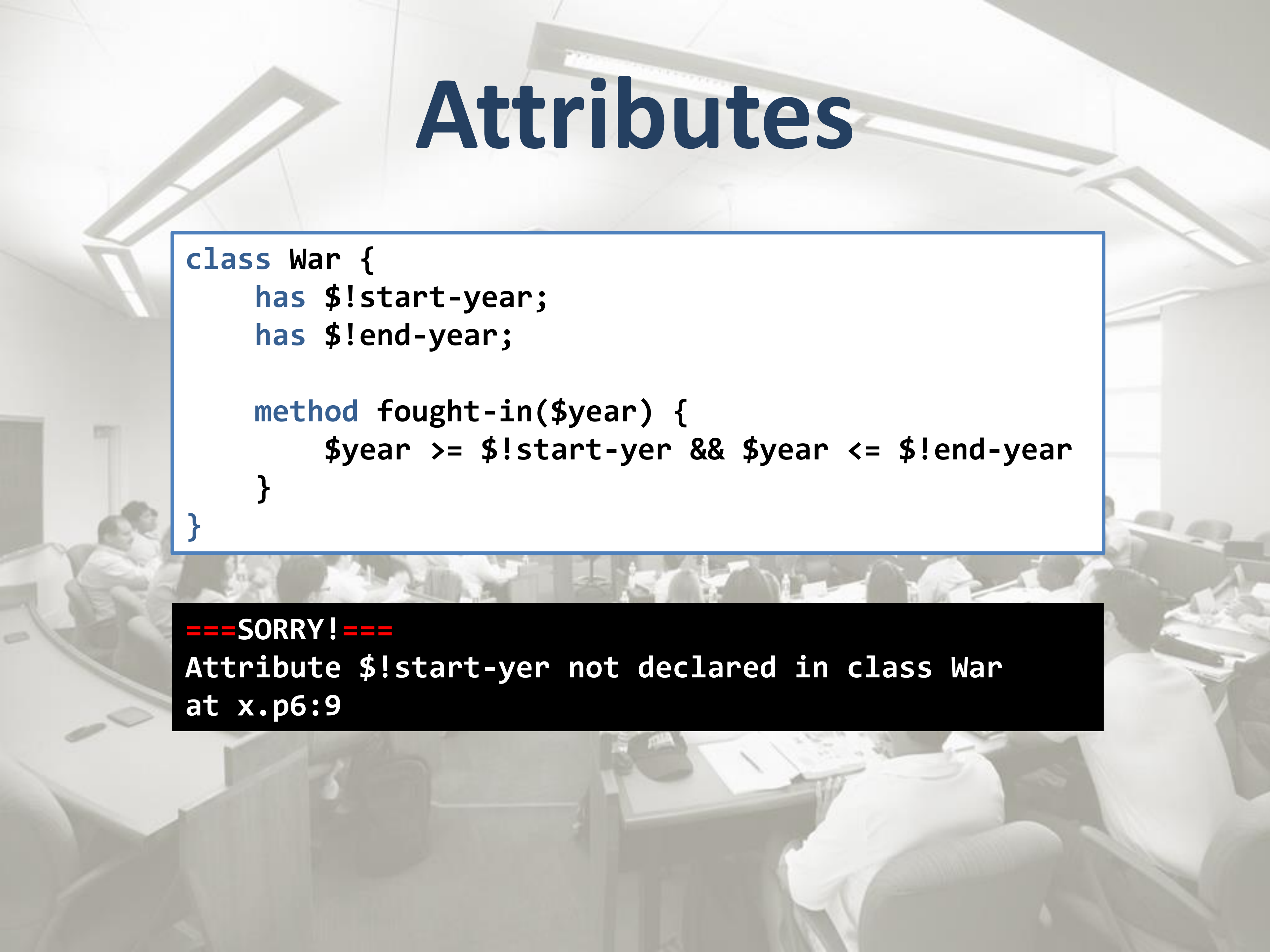
The object's language

Decided at runtime

Inversion of control

Attributes

```
class War {
    has $!start-year;
    has $!end-year;

    method fought-in($year) {
        $year >= $!start-yer && $year <= $!end-year
    }
}
```



Attributes

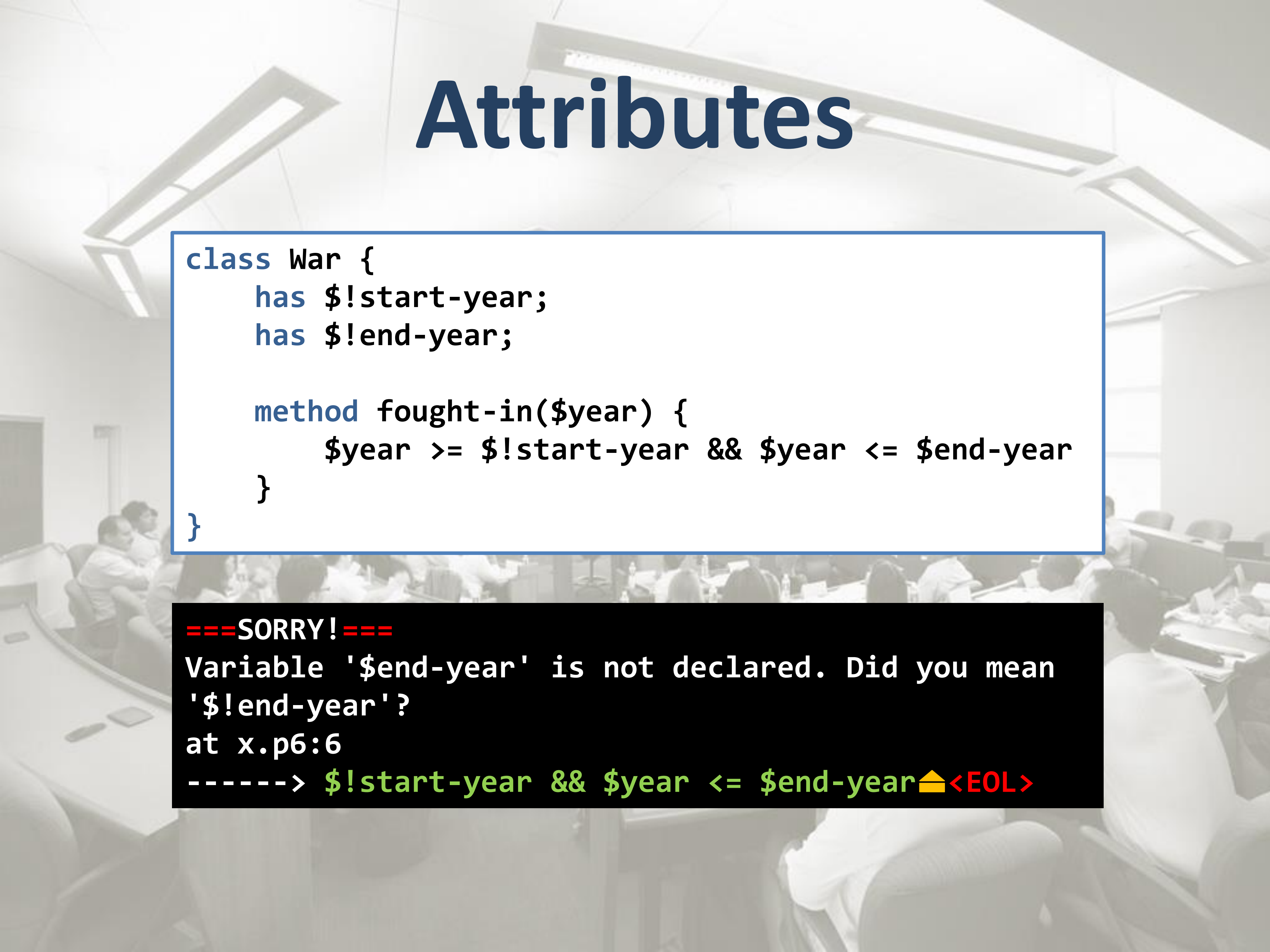
```
class War {
    has $!start-year;
    has $!end-year;

    method fought-in($year) {
        $year >= $!start-yer && $year <= $!end-year
    }
}
```

```
===SORRY!===
Attribute $!start-yer not declared in class War
at x.p6:9
```

Attributes

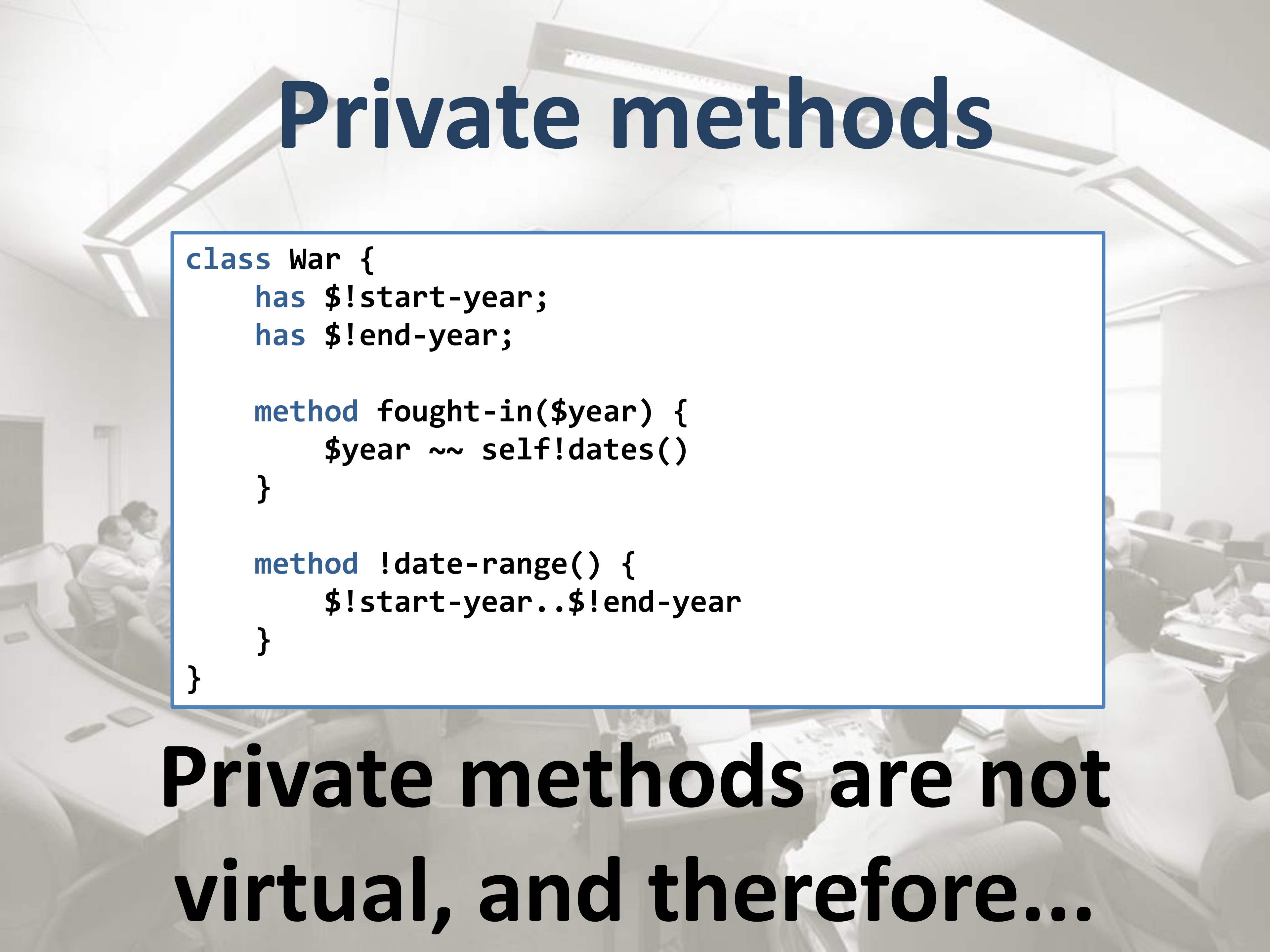
```
class War {
    has $!start-year;
    has $!end-year;

    method fought-in($year) {
        $year >= $!start-year && $year <= $end-year
    }
}
```

Attributes



```
class War {
    has $!start-year;
    has $!end-year;

    method fought-in($year) {
        $year >= $!start-year && $year <= $end-year
    }
}
```

```
===SORRY!===
Variable '$end-year' is not declared. Did you mean
'$!end-year'?
at x.p6:6
------> $!start-year && $year <= $end-year⏏<EOL>
```



Private methods

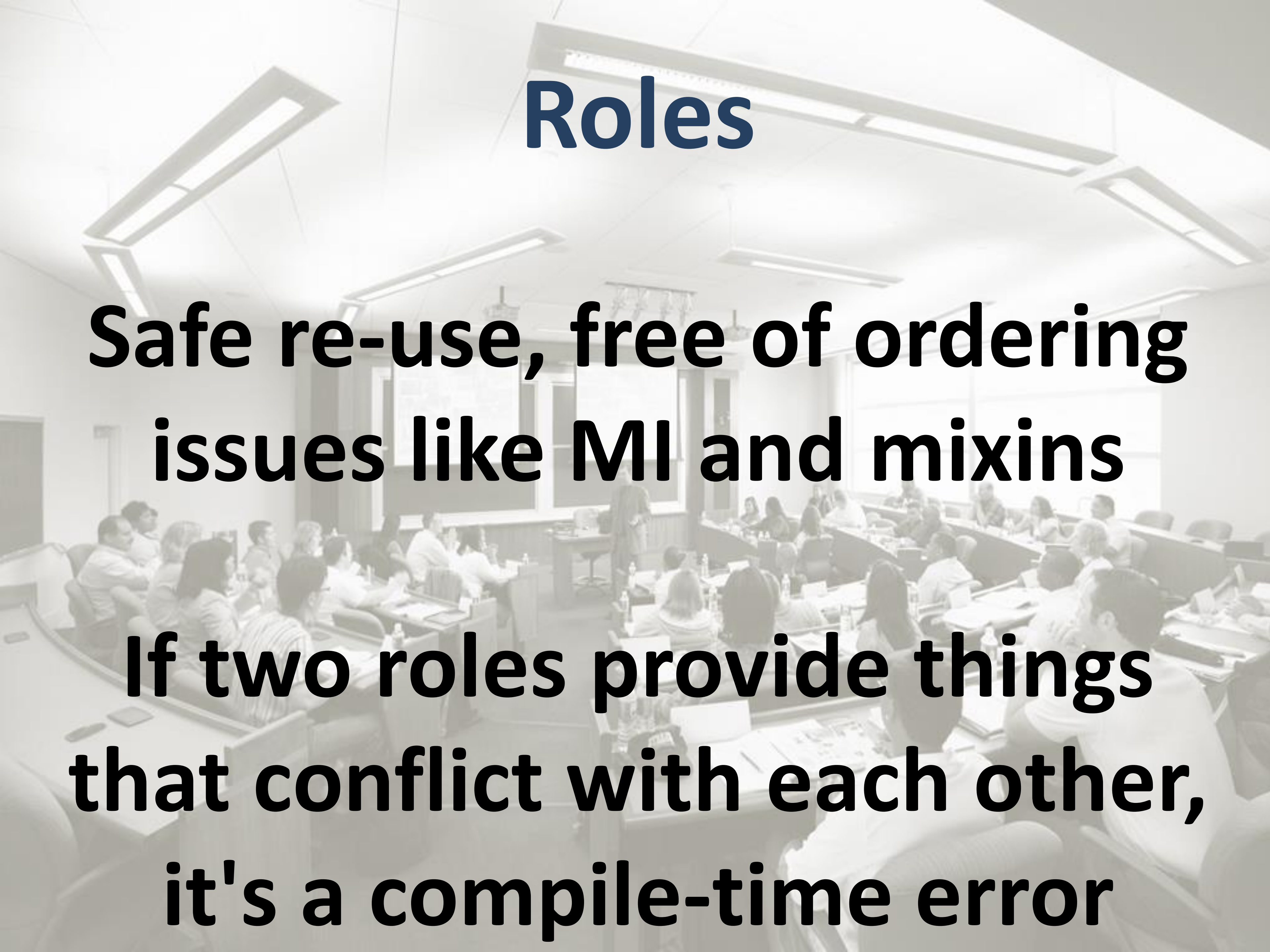
```
class War {
    has $!start-year;
    has $!end-year;

    method fought-in($year) {
        $year ~~ self!dates()
    }

    method !date-range() {
        $!start-year..$!end-year
    }
}
```

Private methods are not virtual, and therefore…

Private methods

```
class War {
    has $!start-year;
    has $!end-year;

    method fought-in($year) {
        $year ~~ self!dates()
    }

    method !date-range() {
        $!start-year..$!end-year
    }
}
```

```
===SORRY!===
No such private method 'dates' for invocant of type
'War'
at x.p6:6
------>         $year ~~ self!dates(⏏)
```

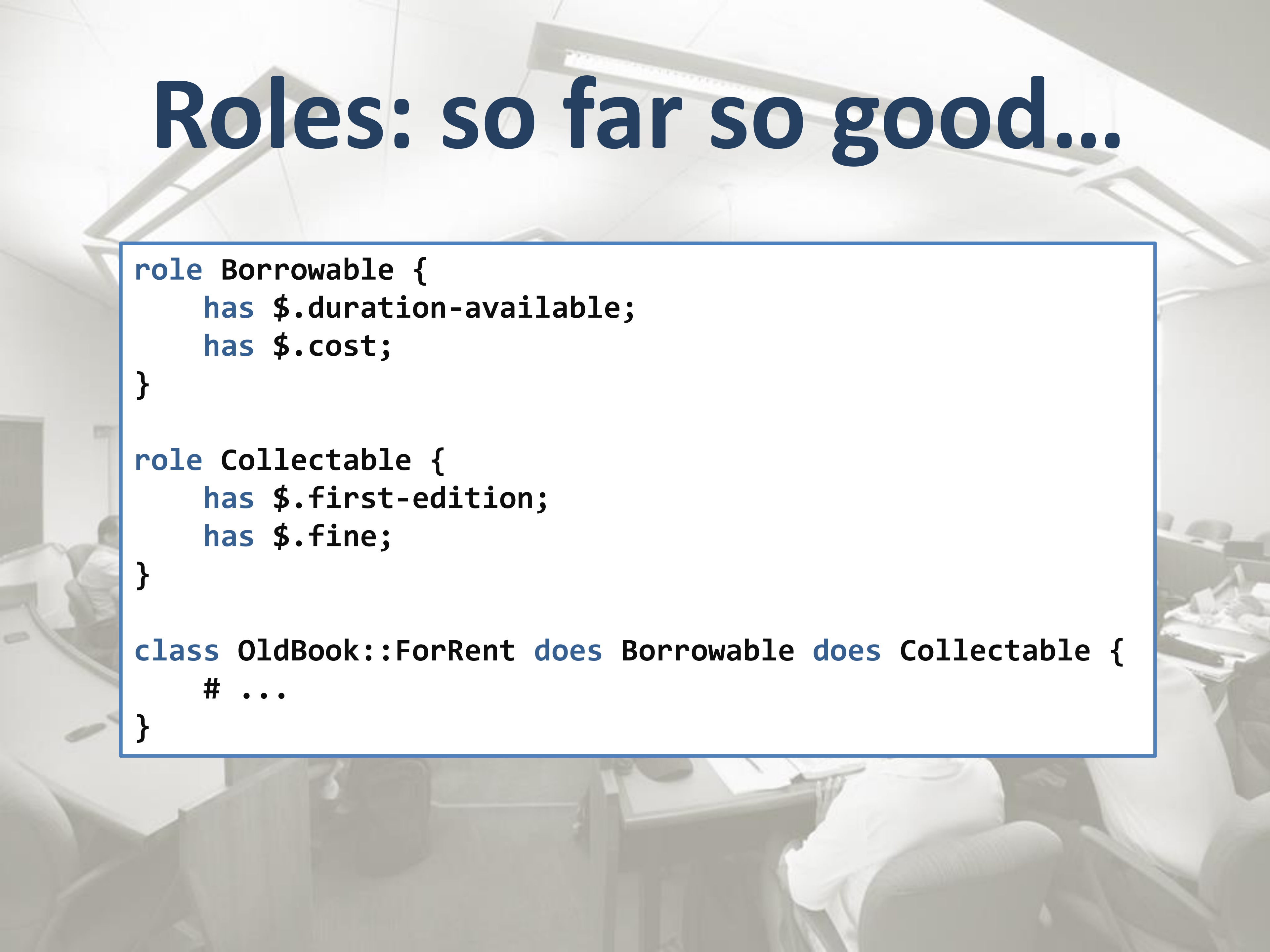


Roles

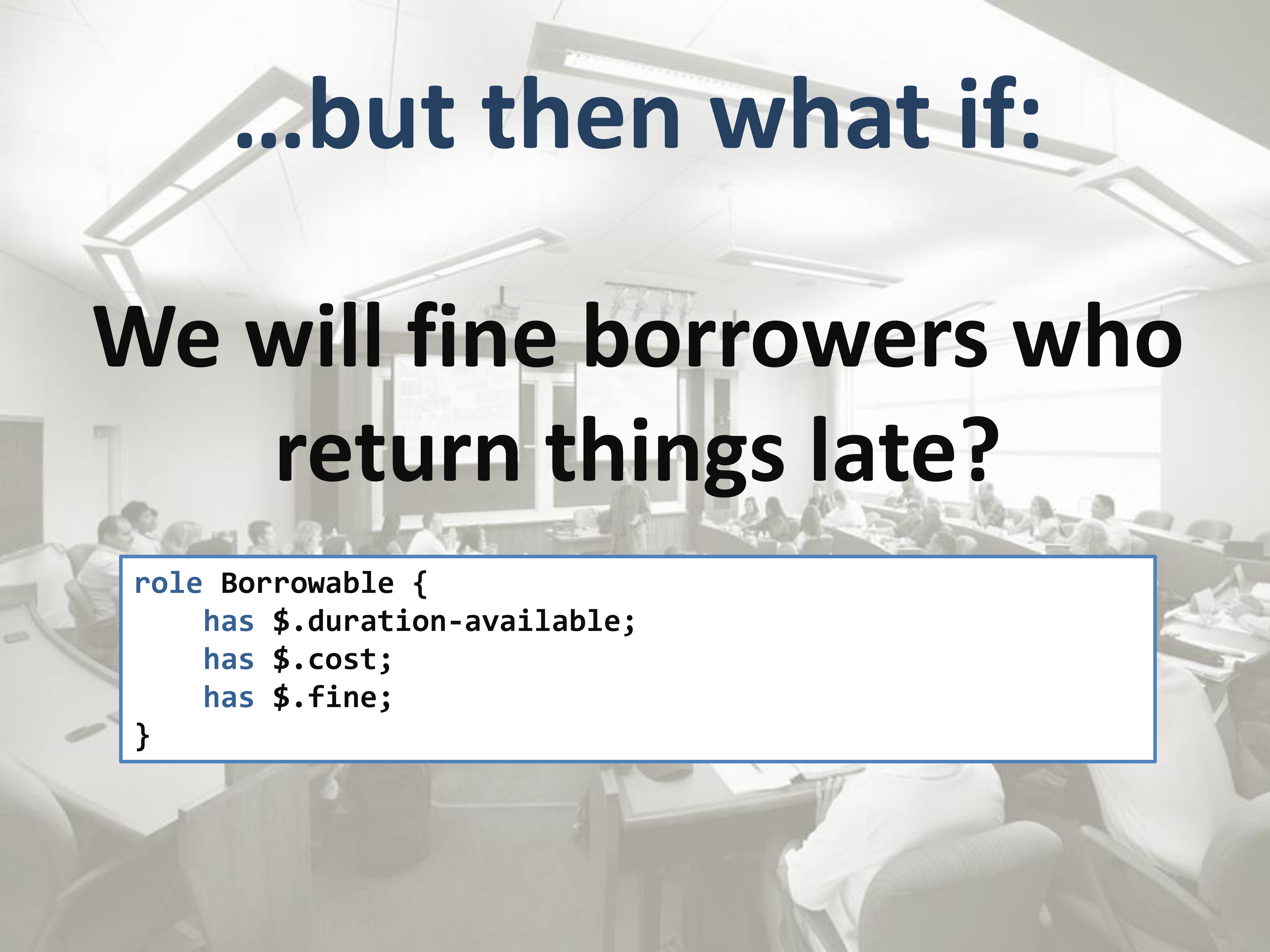
Safe re-use, free of ordering issues like MI and mixins

If two roles provide things that conflict with each other, it's a compile-time error

Roles: so far so good…



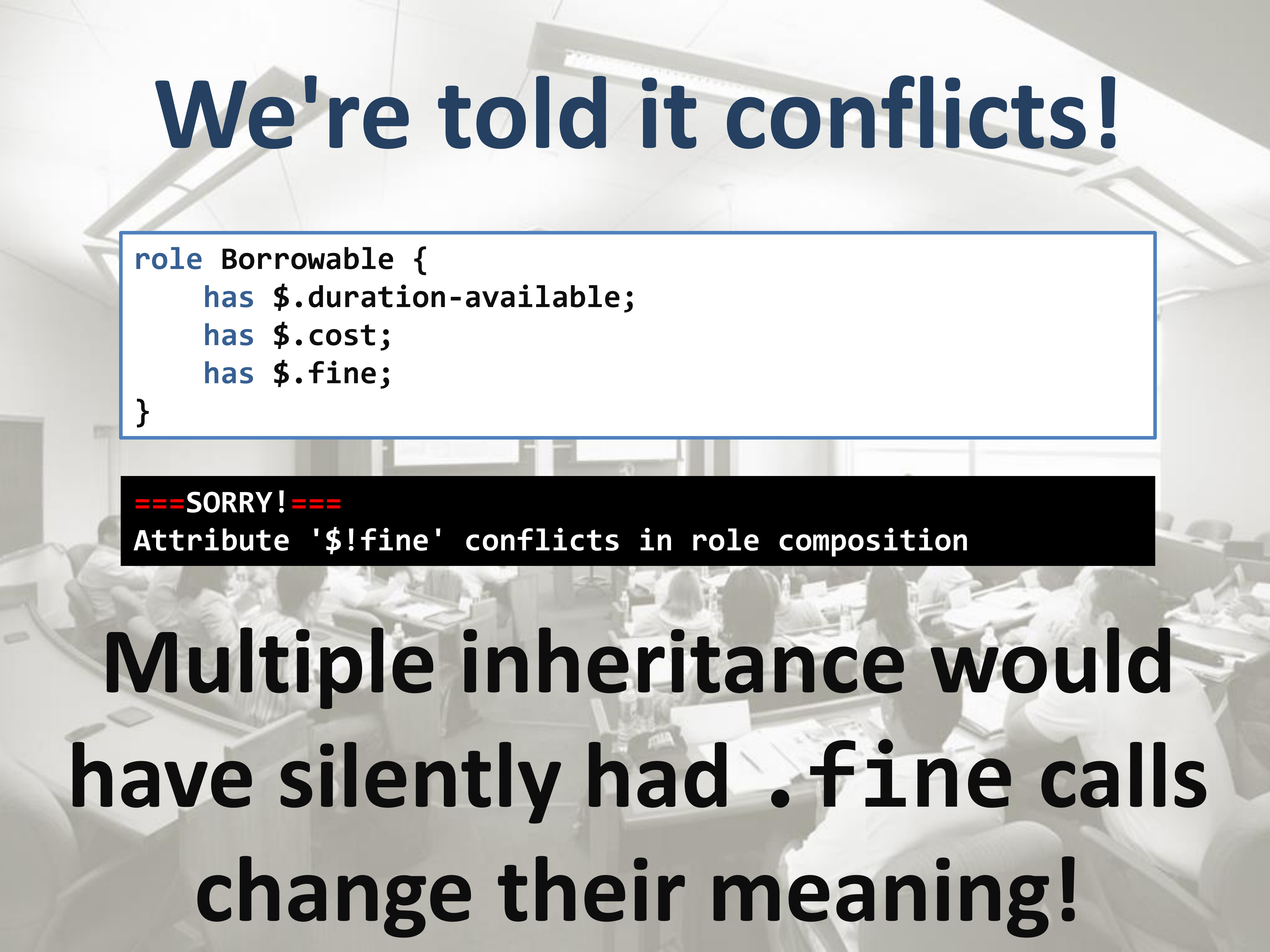
```
role Borrowable {
    has $.duration-available;
    has $.cost;
}

role Collectable {
    has $.first-edition;
    has $.fine;
}

class OldBook::ForRent does Borrowable does Collectable {
    # ...
}
```



...but then what if:

We will fine borrowers who return things late?

```
role Borrowable {
    has $.duration-available;
    has $.cost;
    has $.fine;
}
```

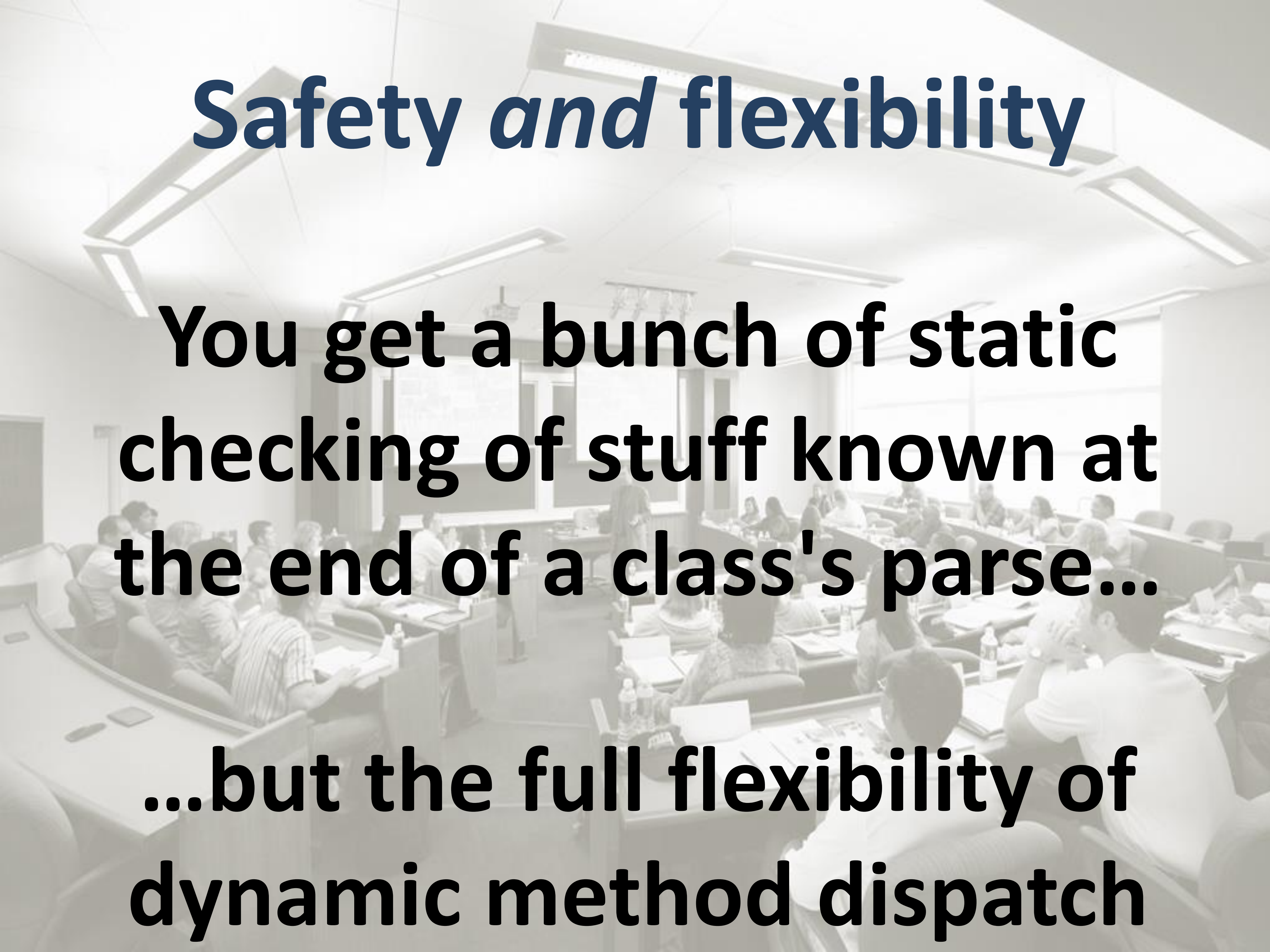


We're told it conflicts!

```
role Borrowable {
    has $.duration-available;
    has $.cost;
    has $.fine;
}
```

```
===SORRY!===
Attribute '$!fine' conflicts in role composition
```

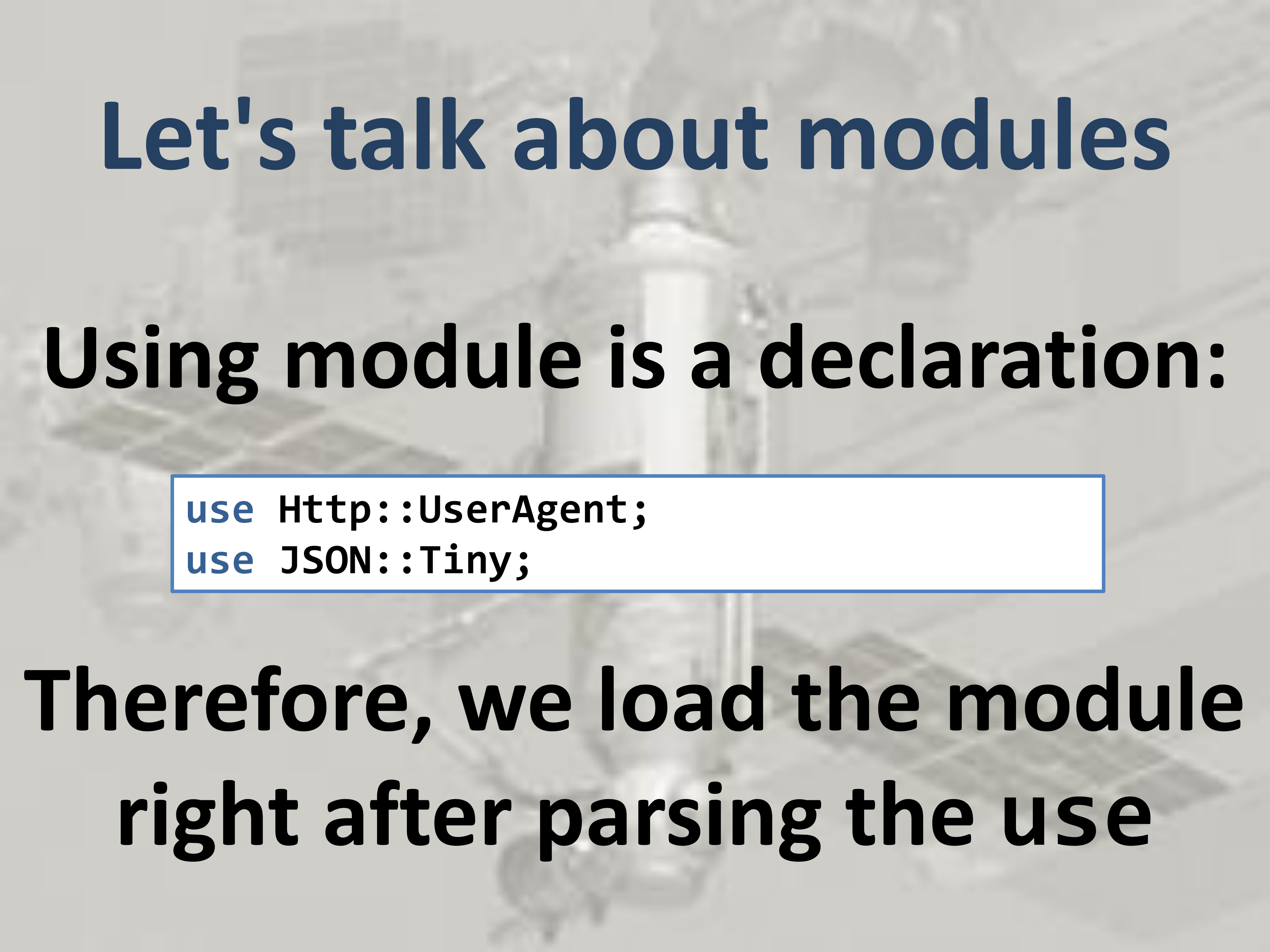
Multiple inheritance would have silently had .fine calls change their meaning!



Safety *and* flexibility

You get a bunch of static checking of stuff known at the end of a class's parse…

…but the full flexibility of dynamic method dispatch



Let's talk about modules

Using module is a declaration:

```
use Http::UserAgent;
use JSON::Tiny;
```

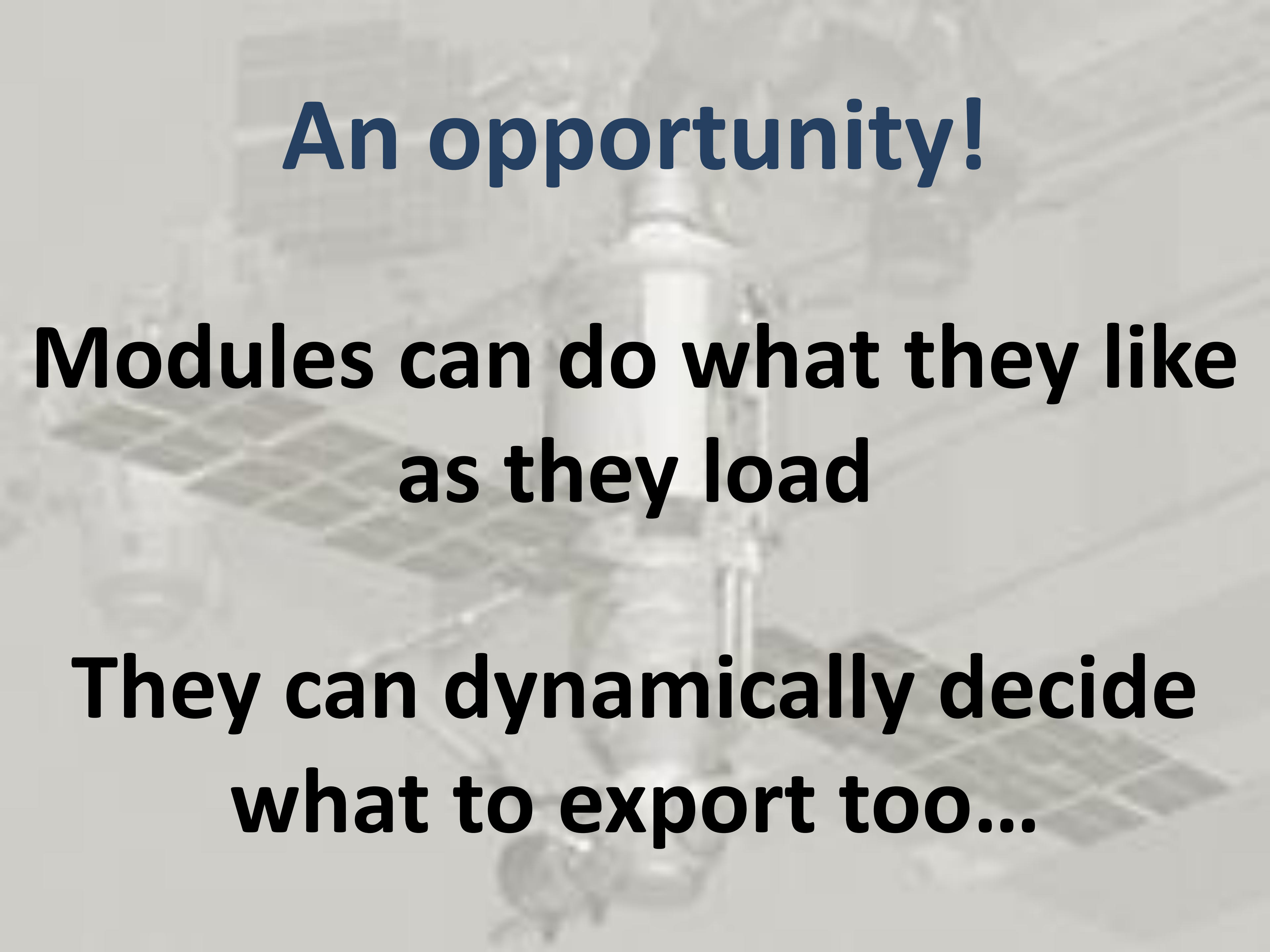
Therefore, we load the module right after parsing the use

Lexical import

By default, imports are lexical

```
{
    use Test;
    plan 42;
}
nok now, 0, "Time is non-zero";
```

```
===SORRY!===
Undeclared routine:
    nok used at line 5
```



An opportunity!

Modules can do what they like as they load

They can dynamically decide what to export too…

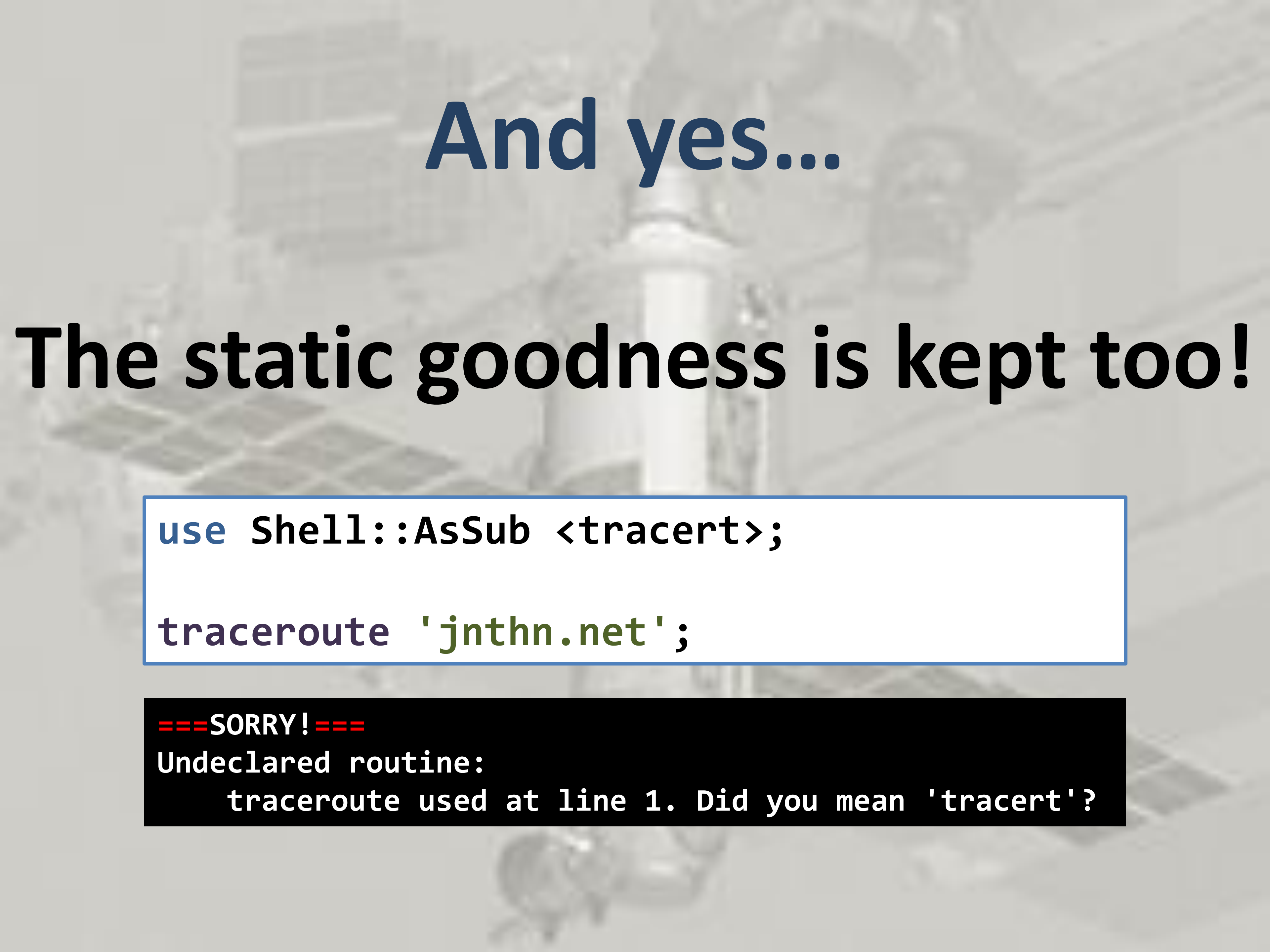
Dynamic subs

Let's write a module to export subs that, when called, shell out and run a command:

```
use Shell::AsSub <ping tracert>;

ping 'jnthn.net';
tracert 'jnthn.net';
```

Shell::AsSub

```
sub EXPORT(*@commands) {
    my %subs;
    for @commands -> $command {
        %subs{'&' ~ $command} = sub (*@args) {
            run $command, |@args;
        }
    }
    return %subs;
}
```

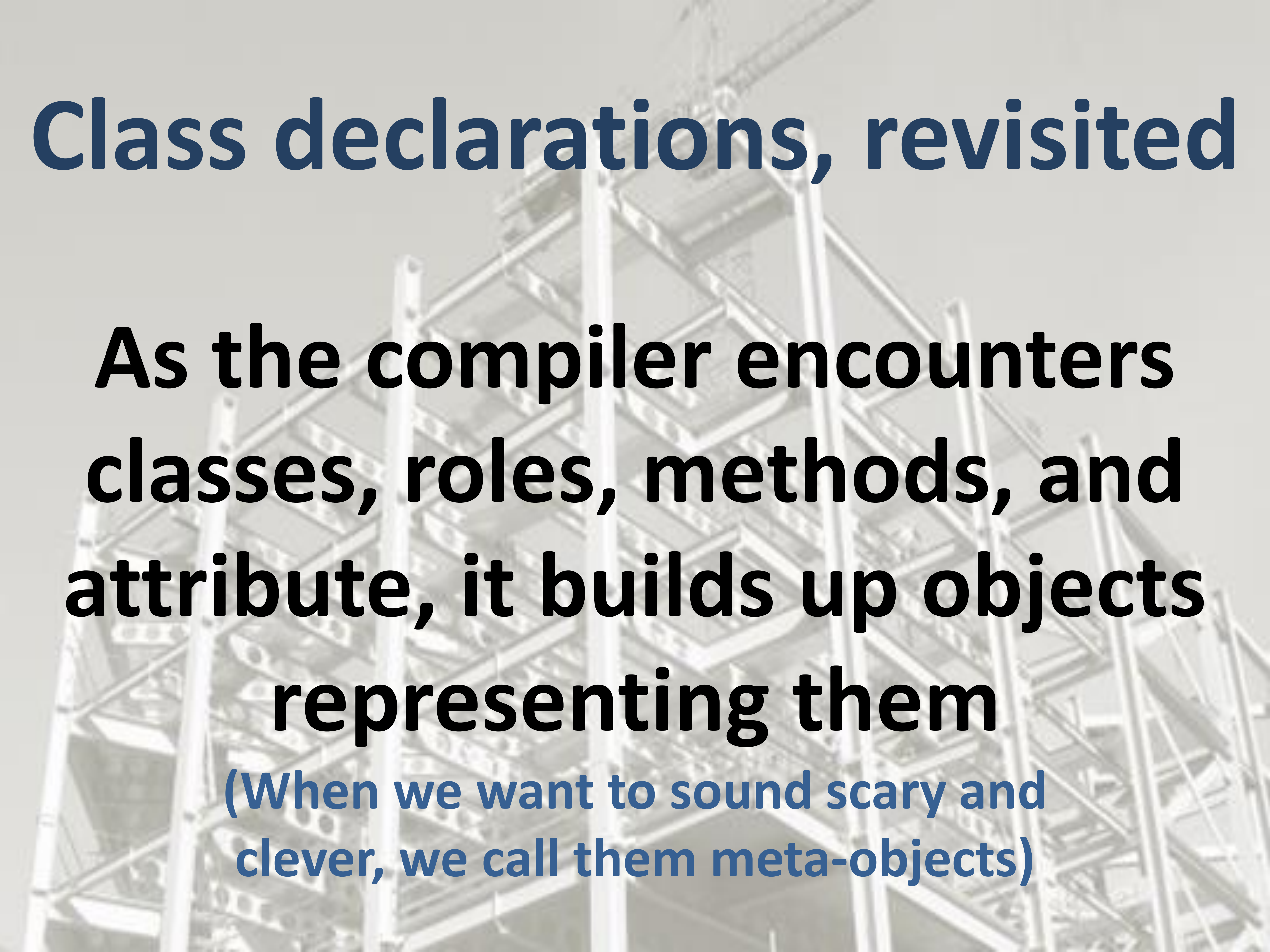


And yes...

The static goodness is kept too!

```
use Shell::AsSub <tracert>;

traceroute 'jnthn.net';
```

```
===SORRY!===
Undeclared routine:
    traceroute used at line 1. Did you mean 'tracert'?
```

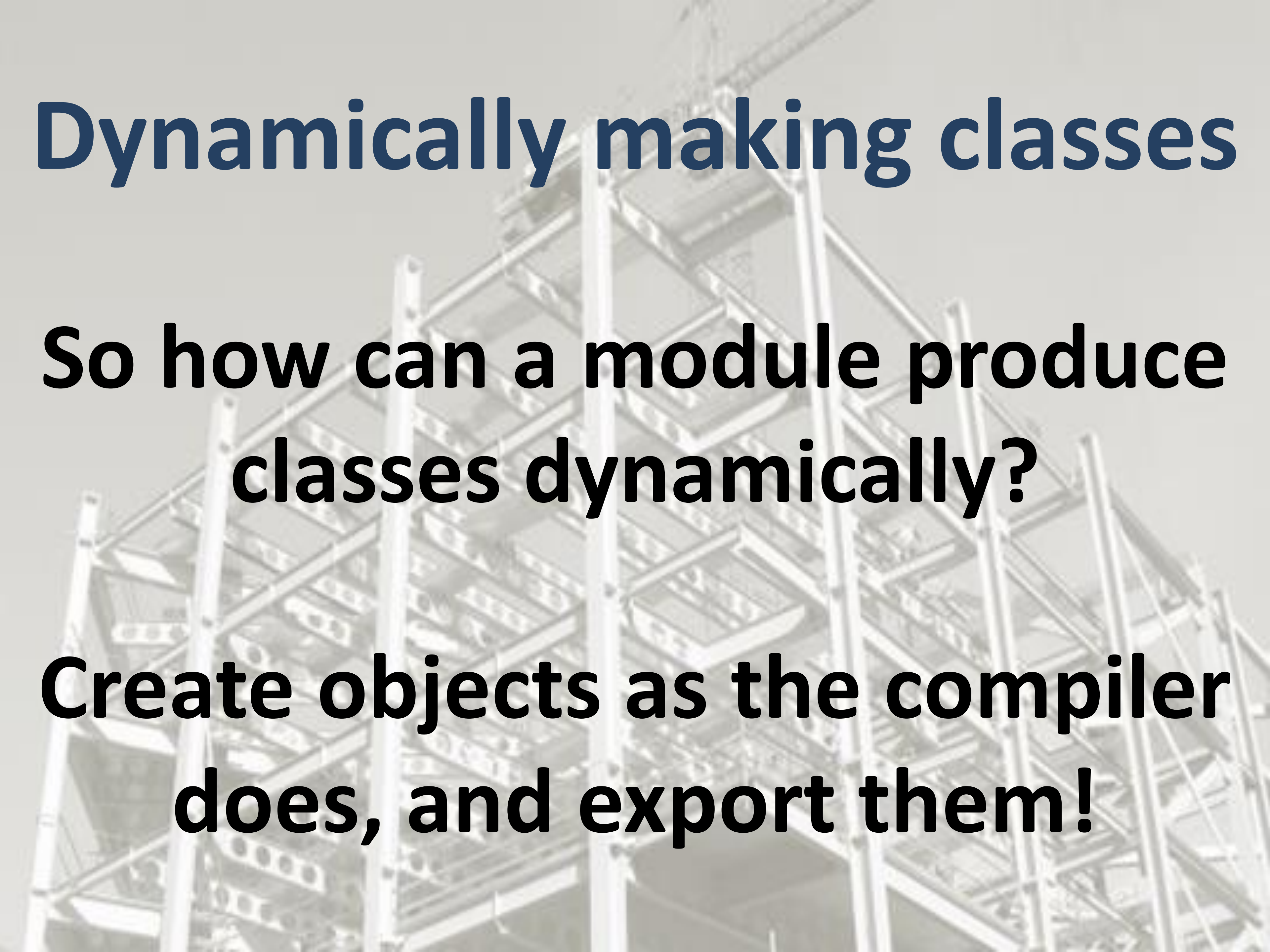


Class declarations, revisited

As the compiler encounters classes, roles, methods, and attribute, it builds up objects representing them

(When we want to sound scary and clever, we call them meta-objects)

Dynamically making classes



So how can a module produce classes dynamically?

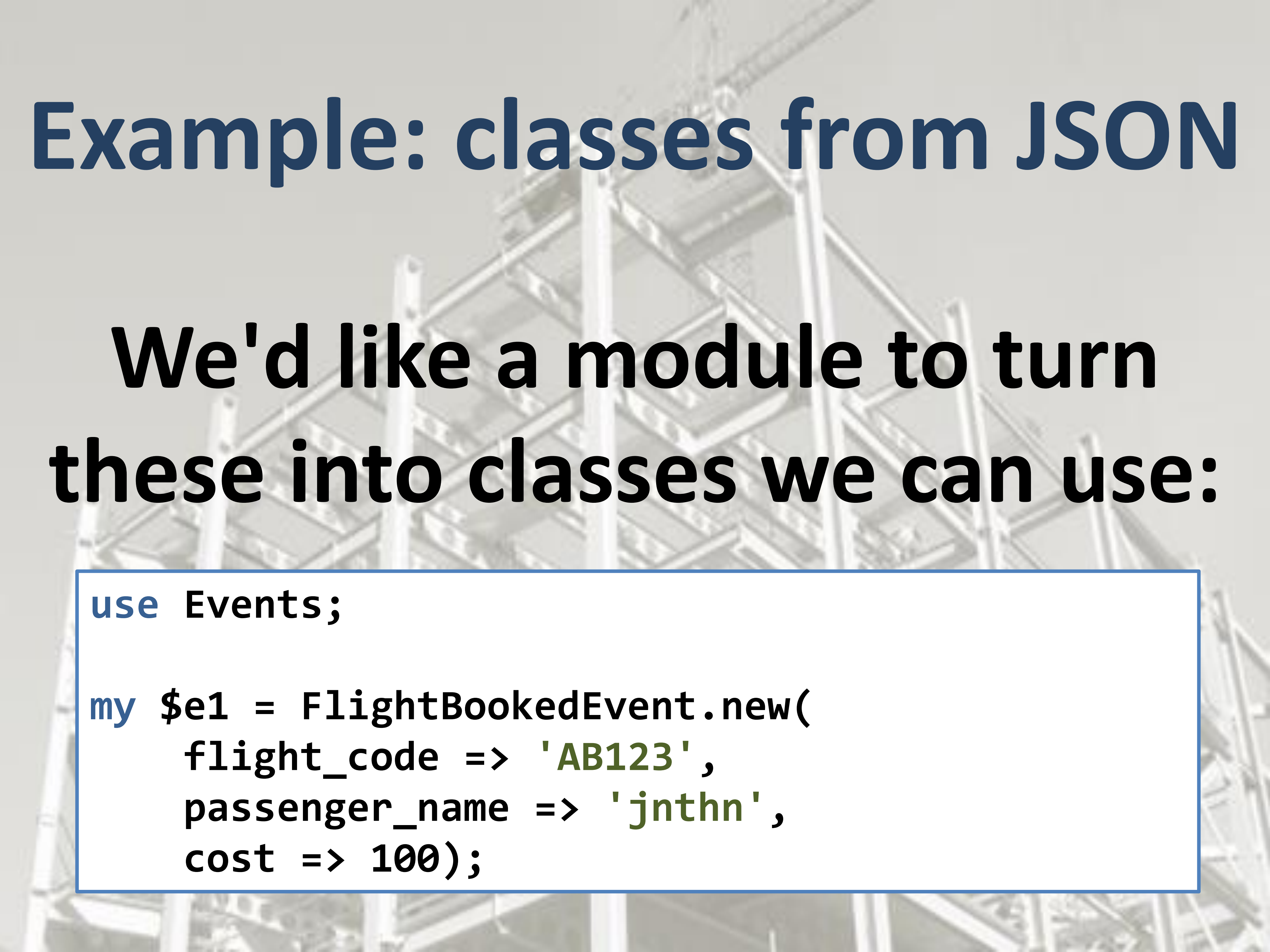
Create objects as the compiler does, and export them!

Example: classes from JSON

Here's a crazy simple schema:

```
[
    {
        "name": "FlightBookedEvent",
        "values": [ "flight_code", "passenger_name", "cost" ]
    },
    {
        "name": "FlightCancelledEvent",
        "values": [ "flight_code", "passenger_name" ]
    }
]
```

Example: classes from JSON

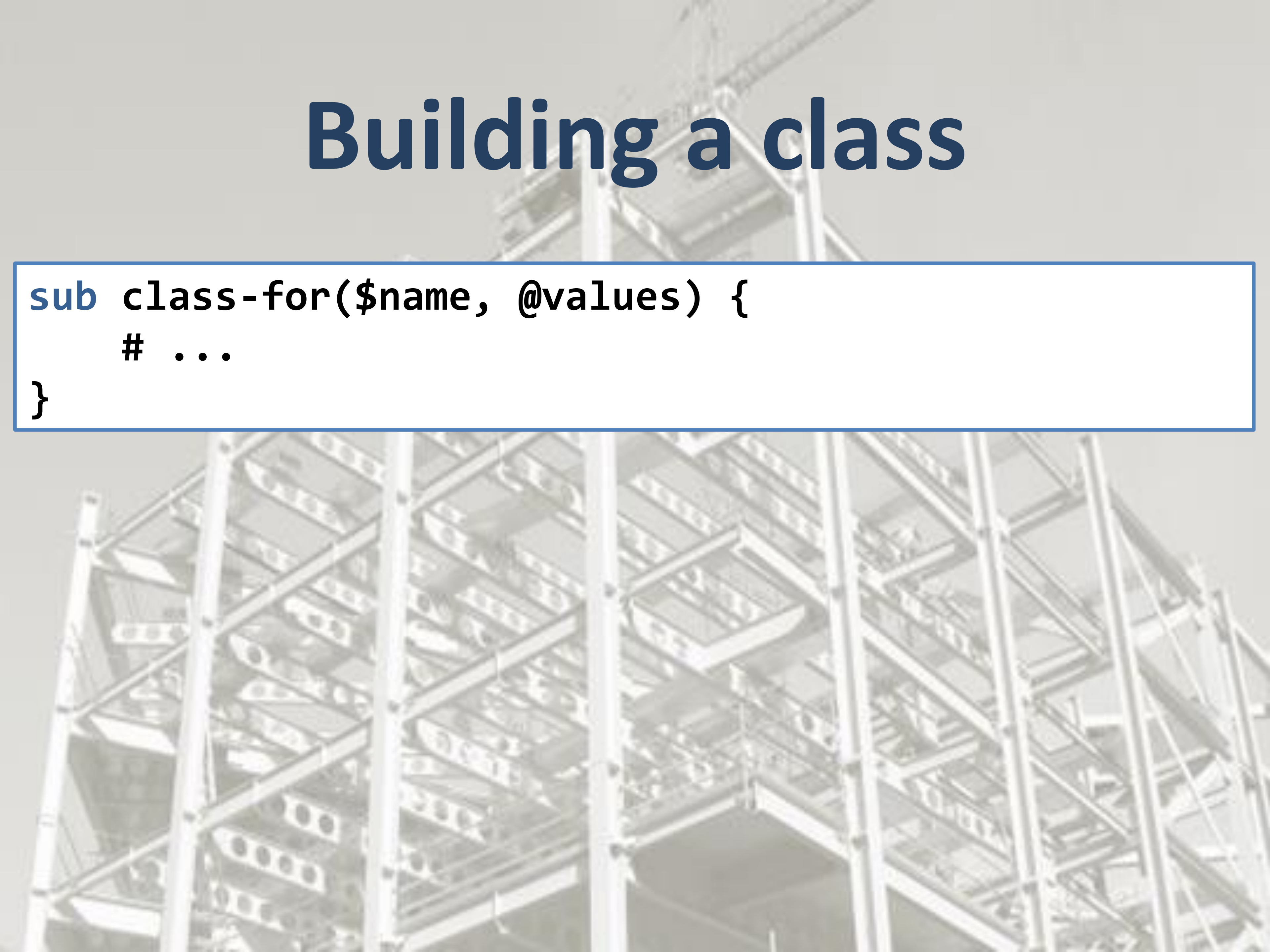


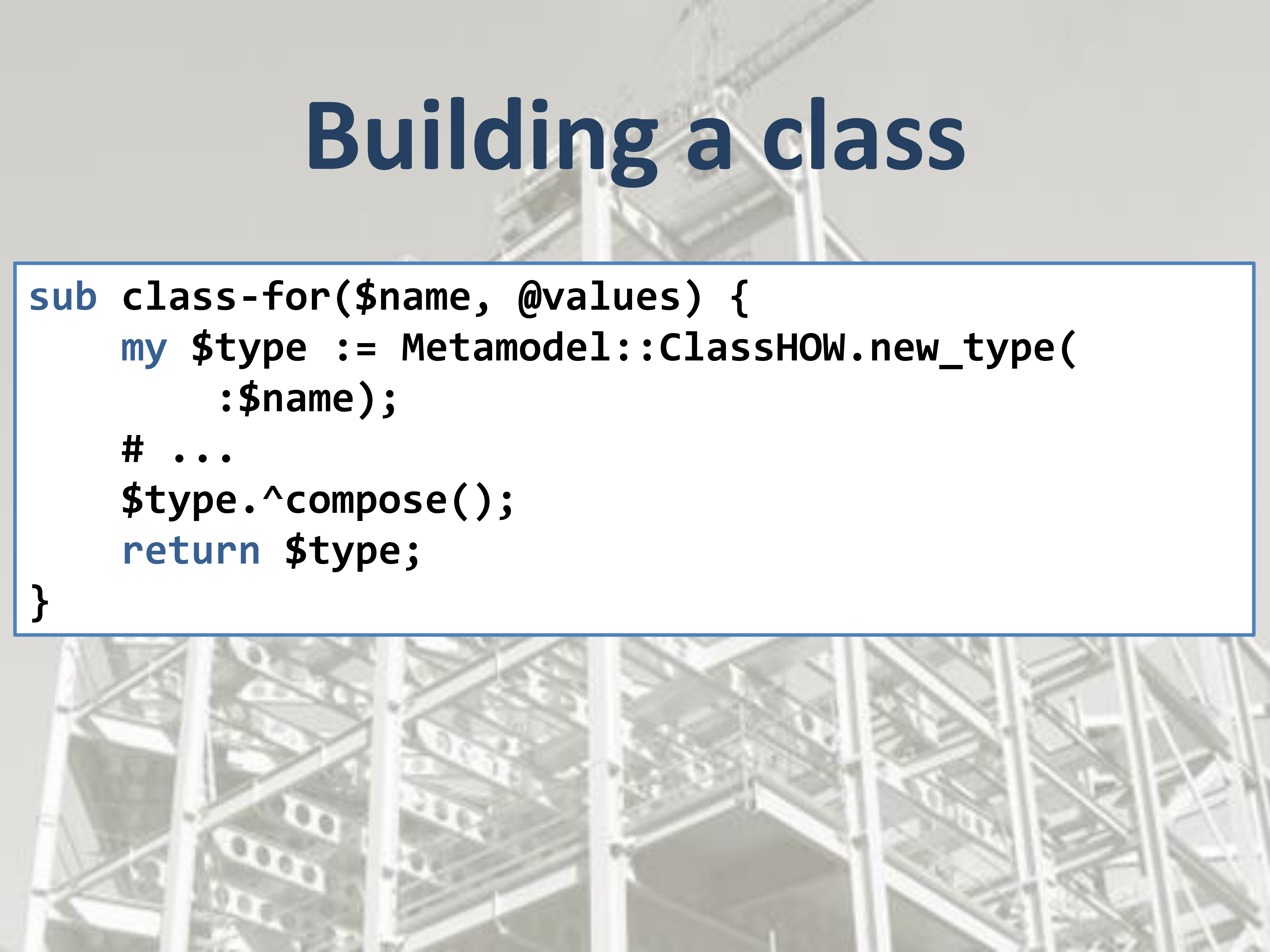
We'd like a module to turn these into classes we can use:

```
use Events;

my $e1 = FlightBookedEvent.new(
    flight_code => 'AB123',
    passenger_name => 'jnthn',
    cost => 100);
```

Building a class

```
sub class-for($name, @values) {
    # ...
}
```



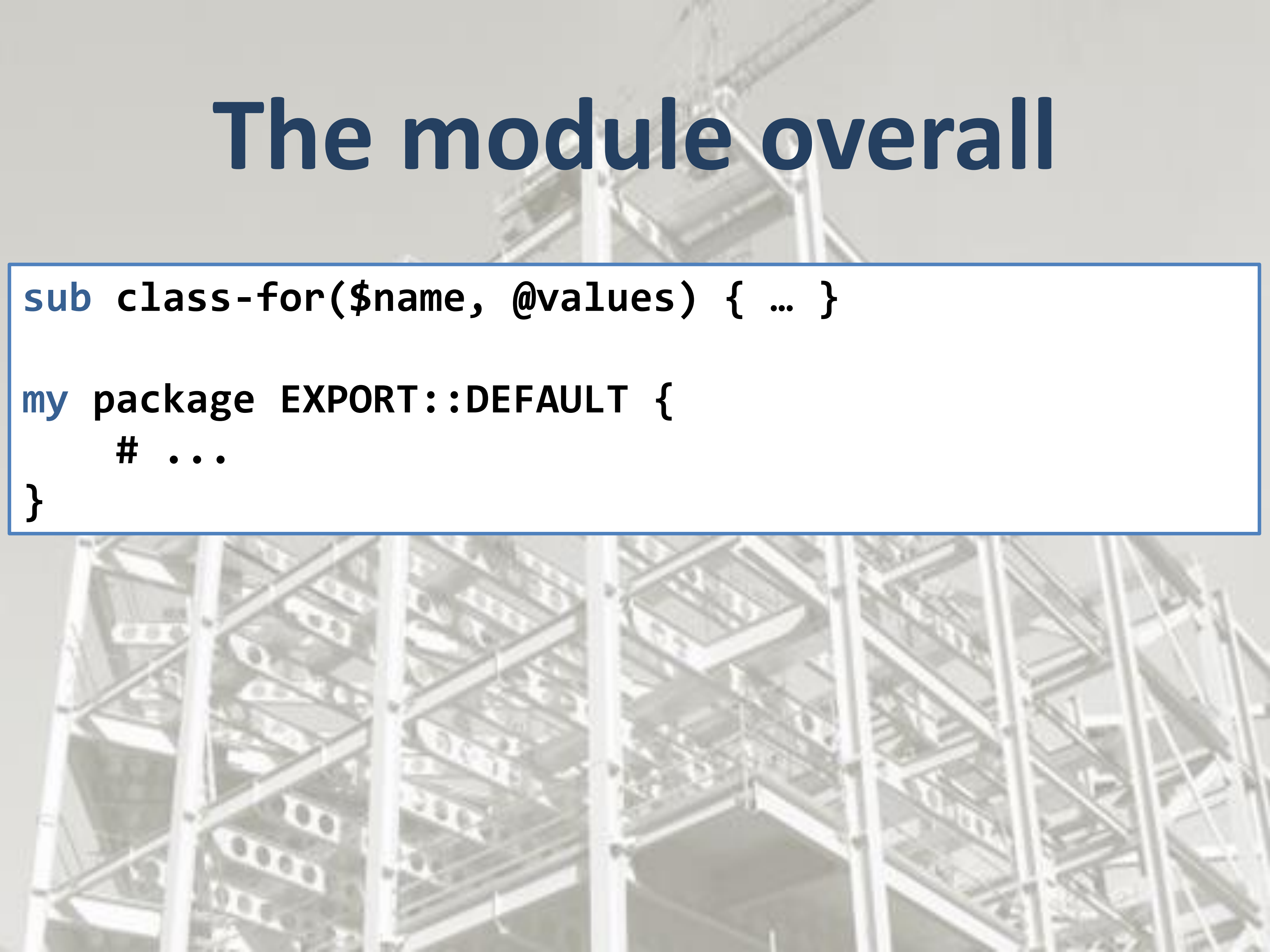


Building a class

```
sub class-for($name, @values) {
    my $type := Metamodel::ClassHOW.new_type(
        :$name);
    # ...
    $type.^compose();
    return $type;
}
```

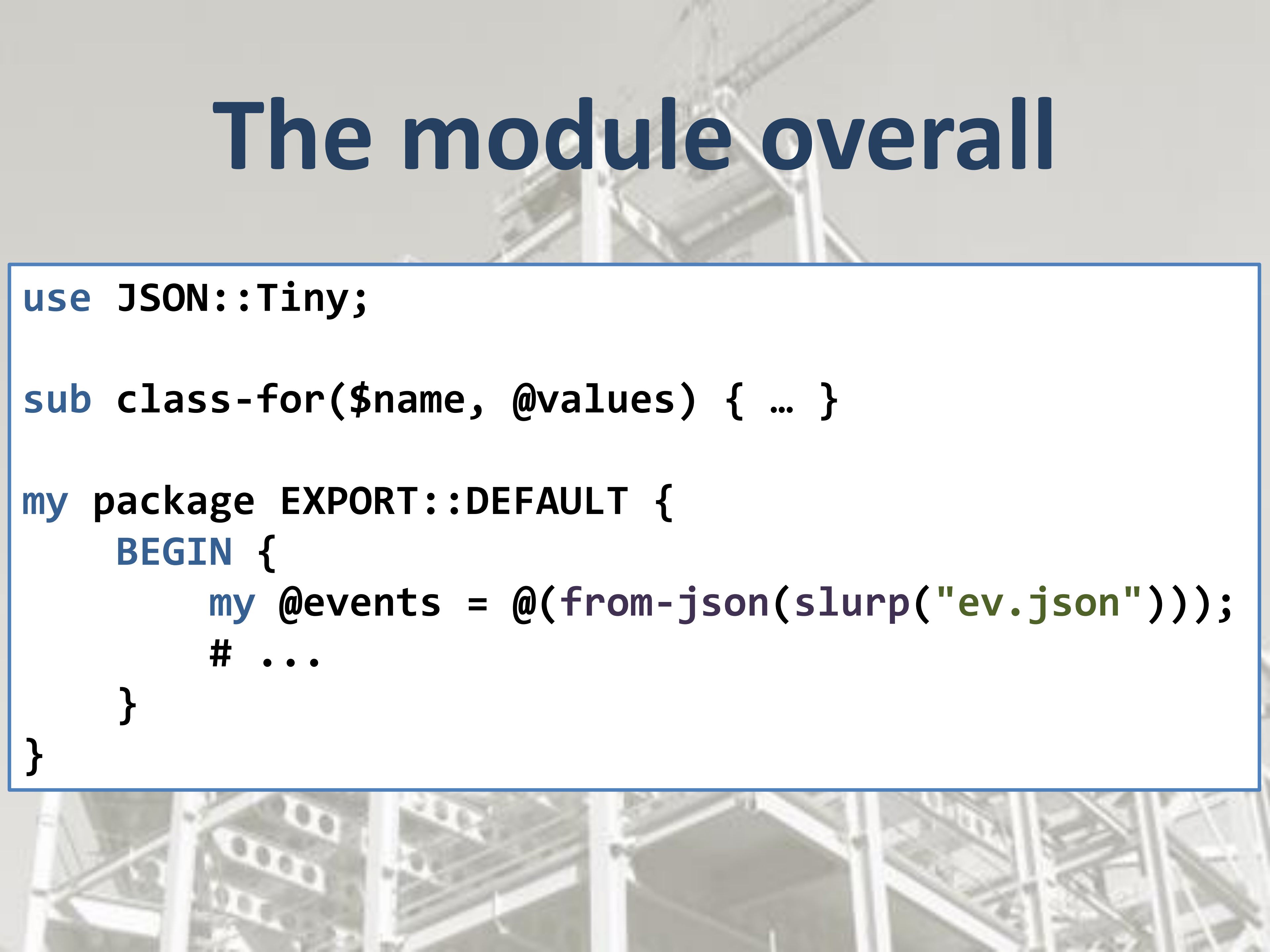
Building a class

```
sub class-for($name, @values) {
    my $type := Metamodel::ClassHOW.new_type(
        :$name);
    for @values -> $attr_name {
        $type.^add_attribute(Attribute.new(
            :name('$!' ~ $attr_name), :type(Mu),
            :has_accessor(1), :package($type)
        ));
    }
    $type.^compose();
    return $type;
}
```



The module overall

```
sub class-for($name, @values) { … }

my package EXPORT::DEFAULT {
    # ...
}
```



The module overall

```
use JSON::Tiny;

sub class-for($name, @values) { … }

my package EXPORT::DEFAULT {
    BEGIN {
        my @events = @(from-json(slurp("ev.json")));
        # ...
    }
}
```

The module overall

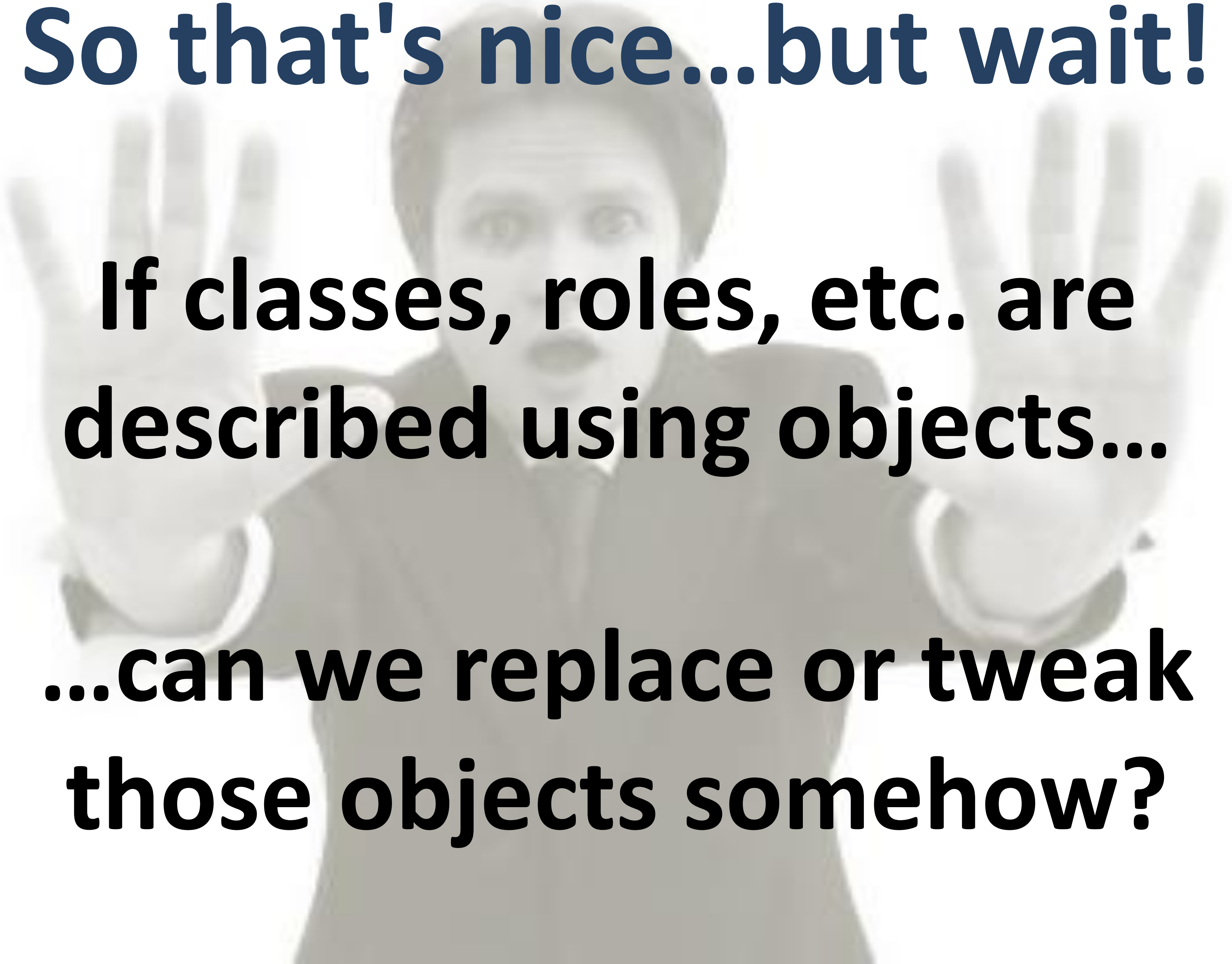
```
use JSON::Tiny;

sub class-for($name, @values) { … }

my package EXPORT::DEFAULT {
    BEGIN {
        my @events = @(from-json(slurp("ev.json")));
        for @events -> (:$name, :@values) {
            OUR::{$name} := class-for(
                $name, @values);
        }
    }
}
```

And that BEGIN…

```
use JSON::Tiny;

sub class-for($name, @values) { … }

my package EXPORT::DEFAULT {
    BEGIN {
        my @events = @(from-json(slurp("ev.json")));
        for @events -> (:$name, :@values) {
            OUR::{$name} := class-for(
                $name, @values);
        }
    }
}
```

If we pre-compile the module to VM bytecode, read the JSON just then and persist the classes that are produced

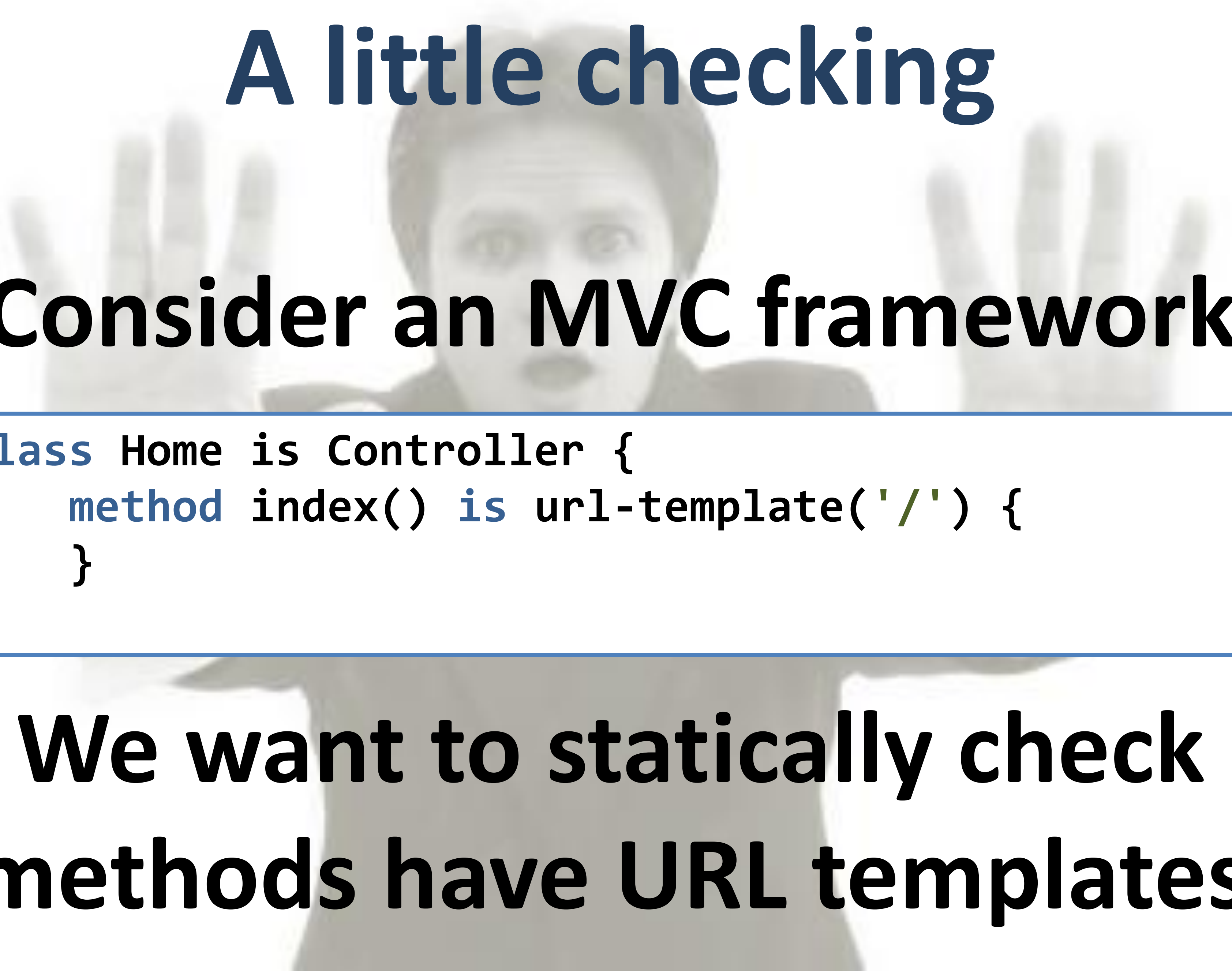
So that's nice...but wait!



If classes, roles, etc. are described using objects...

...can we replace or tweak those objects somehow?

A little checking

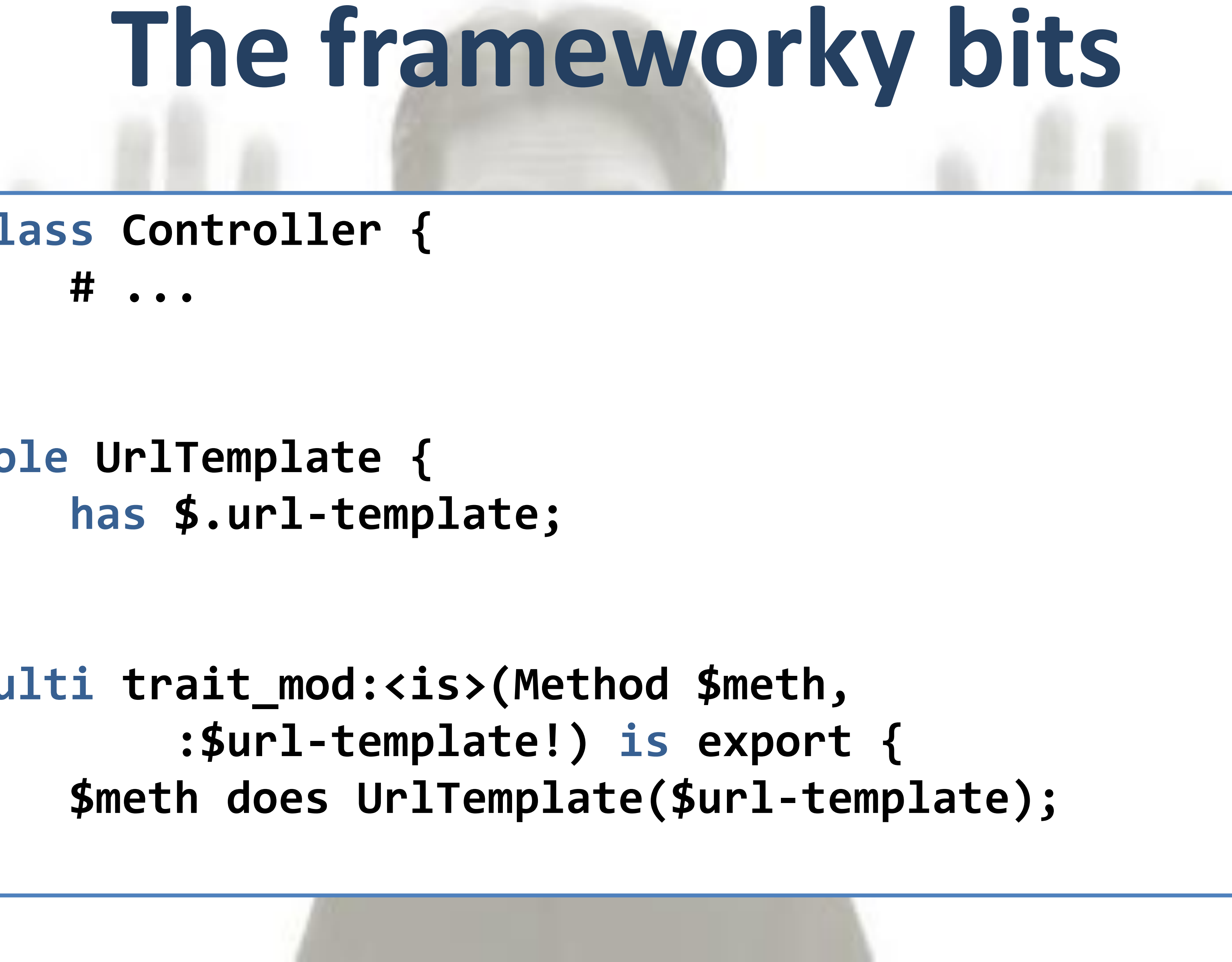


Consider an MVC framework

```
class Home is Controller {
    method index() is url-template('/') {
    }
}
```

We want to statically check methods have URL templates

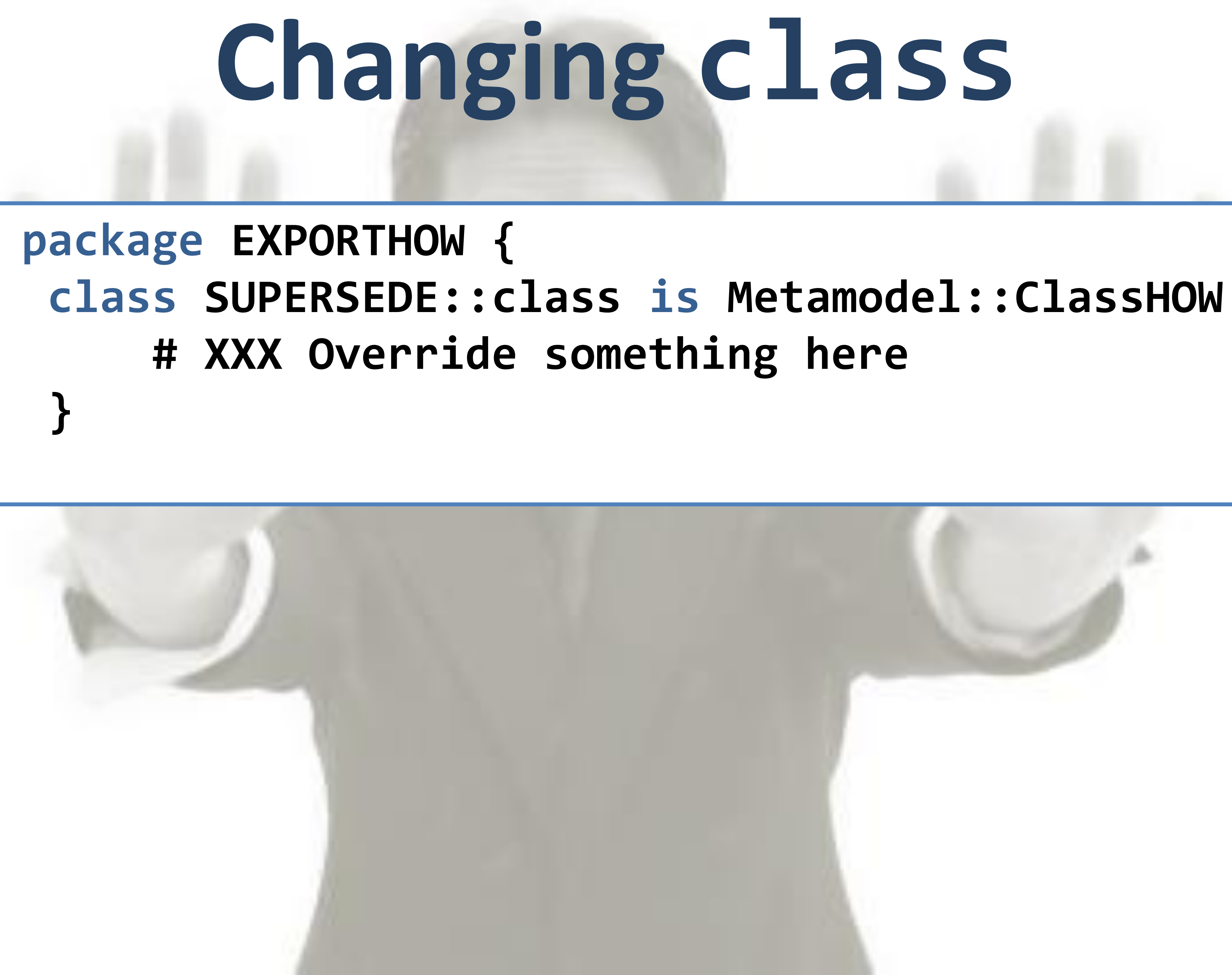
The frameworky bits



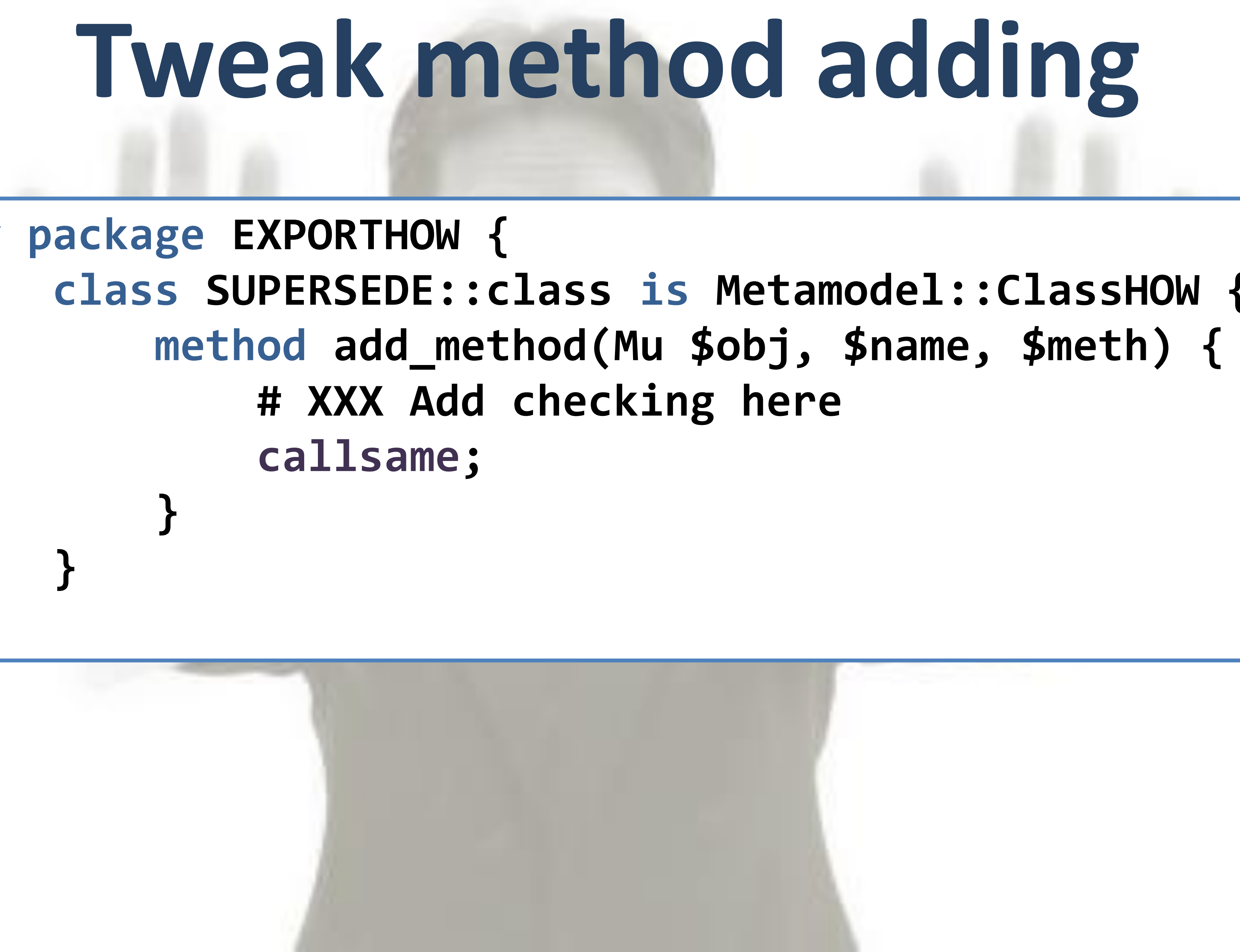
```
class Controller {
    # ...
}

role UrlTemplate {
    has $.url-template;
}

multi trait_mod:<is>(Method $meth,
        :$url-template!) is export {
    $meth does UrlTemplate($url-template);
}
```

Changing class

```
my package EXPORTHOW {
    class SUPERSEDE::class is Metamodel::ClassHOW {
        # XXX Override something here
    }
}
```

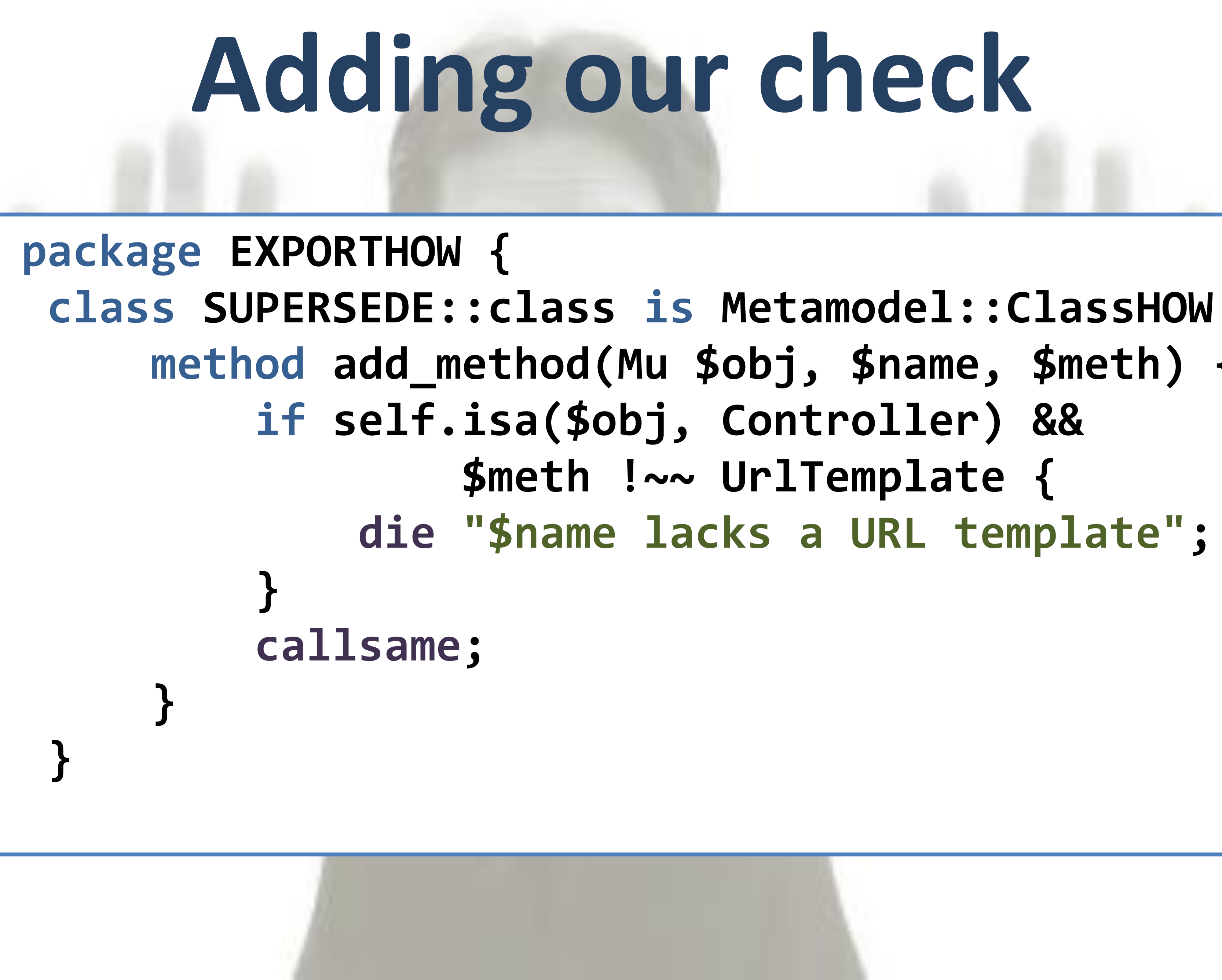


Tweak method adding



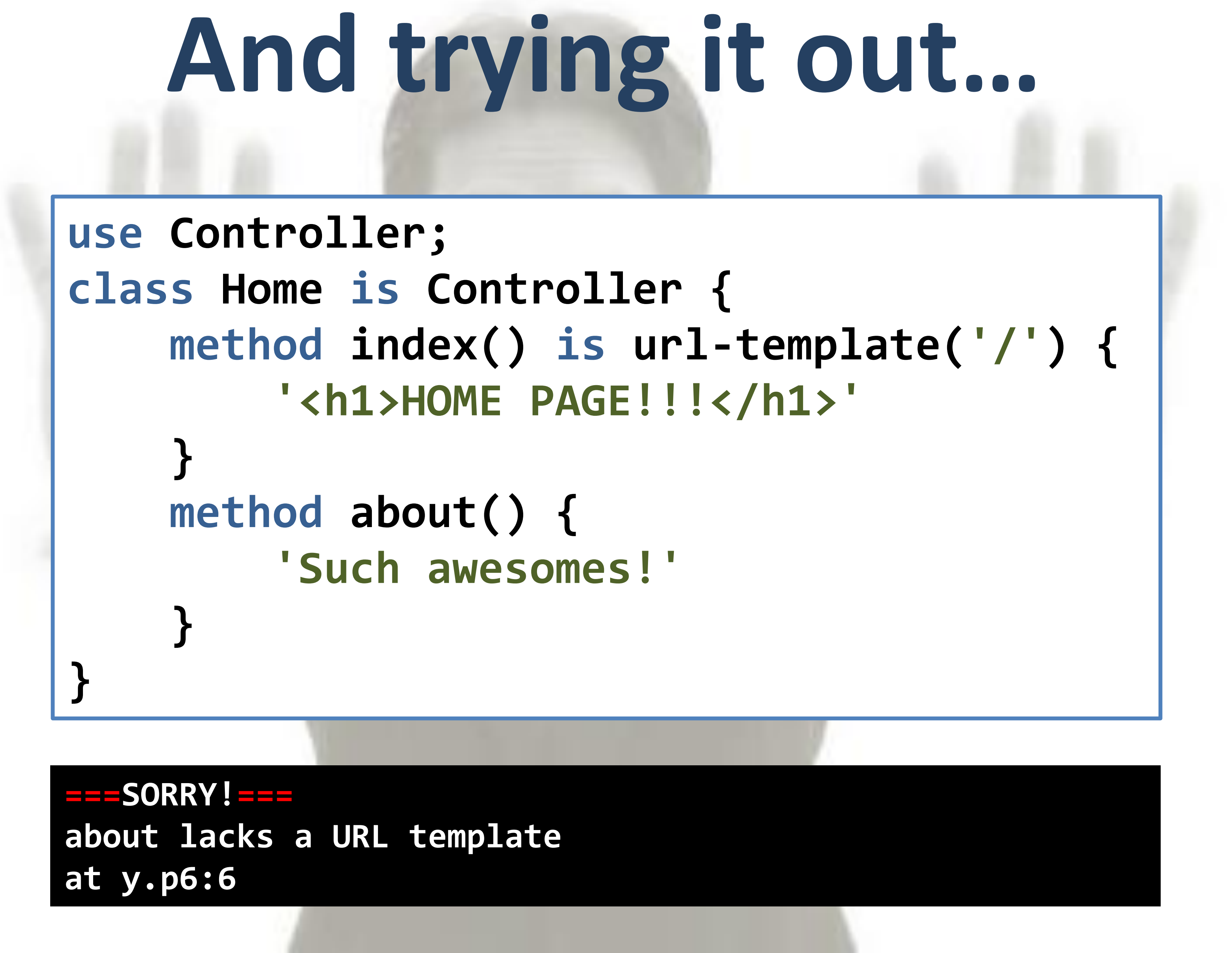
```
my package EXPORTHOW {
    class SUPERSEDE::class is Metamodel::ClassHOW {
        method add_method(Mu $obj, $name, $meth) {
            # XXX Add checking here
            callsame;
        }
    }
}
```

Adding our check



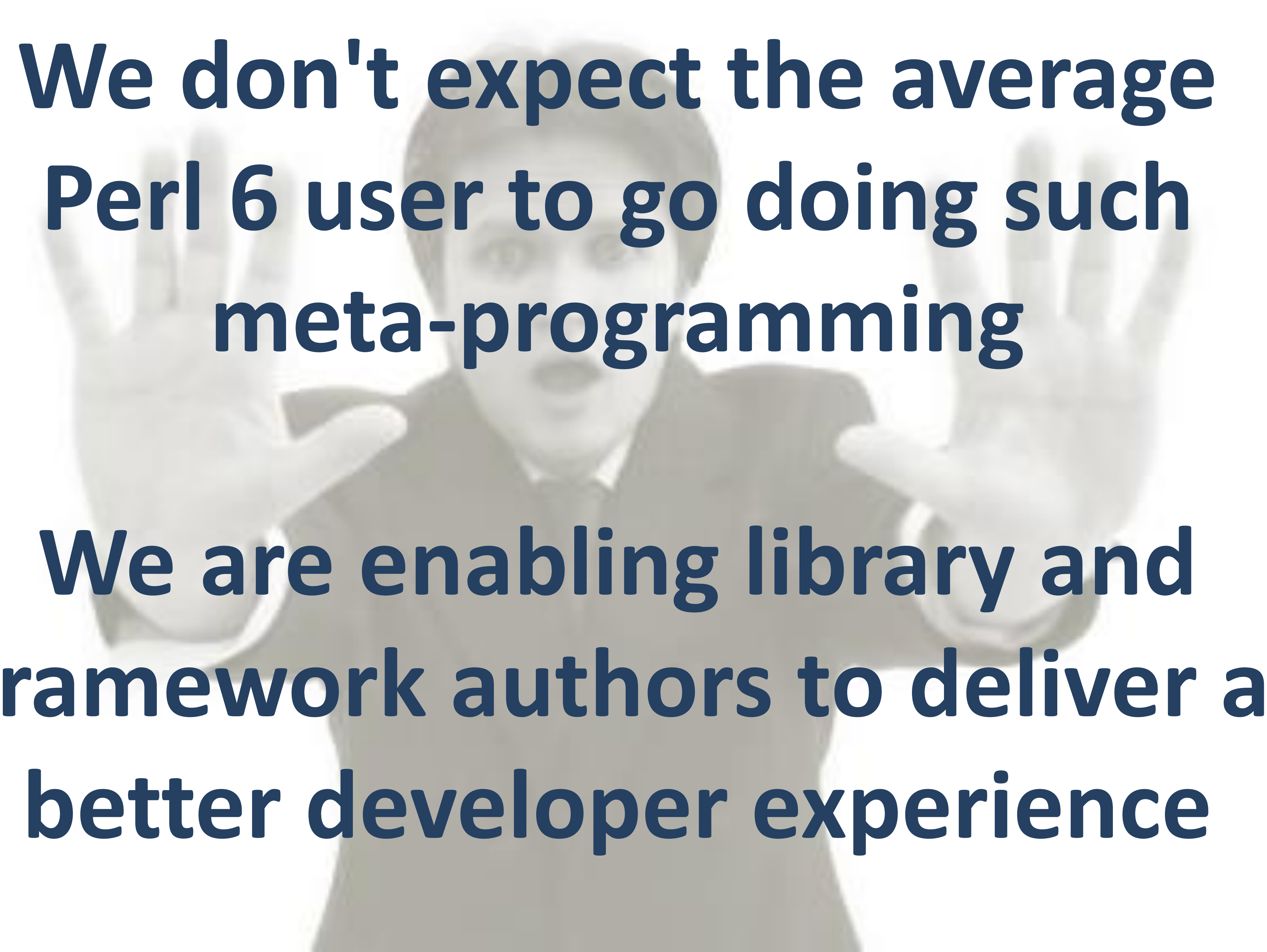
```
my package EXPORTHOW {
    class SUPERSEDE::class is Metamodel::ClassHOW {
        method add_method(Mu $obj, $name, $meth) {
            if self.isa($obj, Controller) &&
                    $meth !~~ UrlTemplate {
                die "$name lacks a URL template";
            }
            callsame;
        }
    }
}
```

And trying it out...



```
use Controller;
class Home is Controller {
    method index() is url-template('/') {
        '<h1>HOME PAGE!!!</h1>'
    }
    method about() {
        'Such awesomes!'
    }
}
```

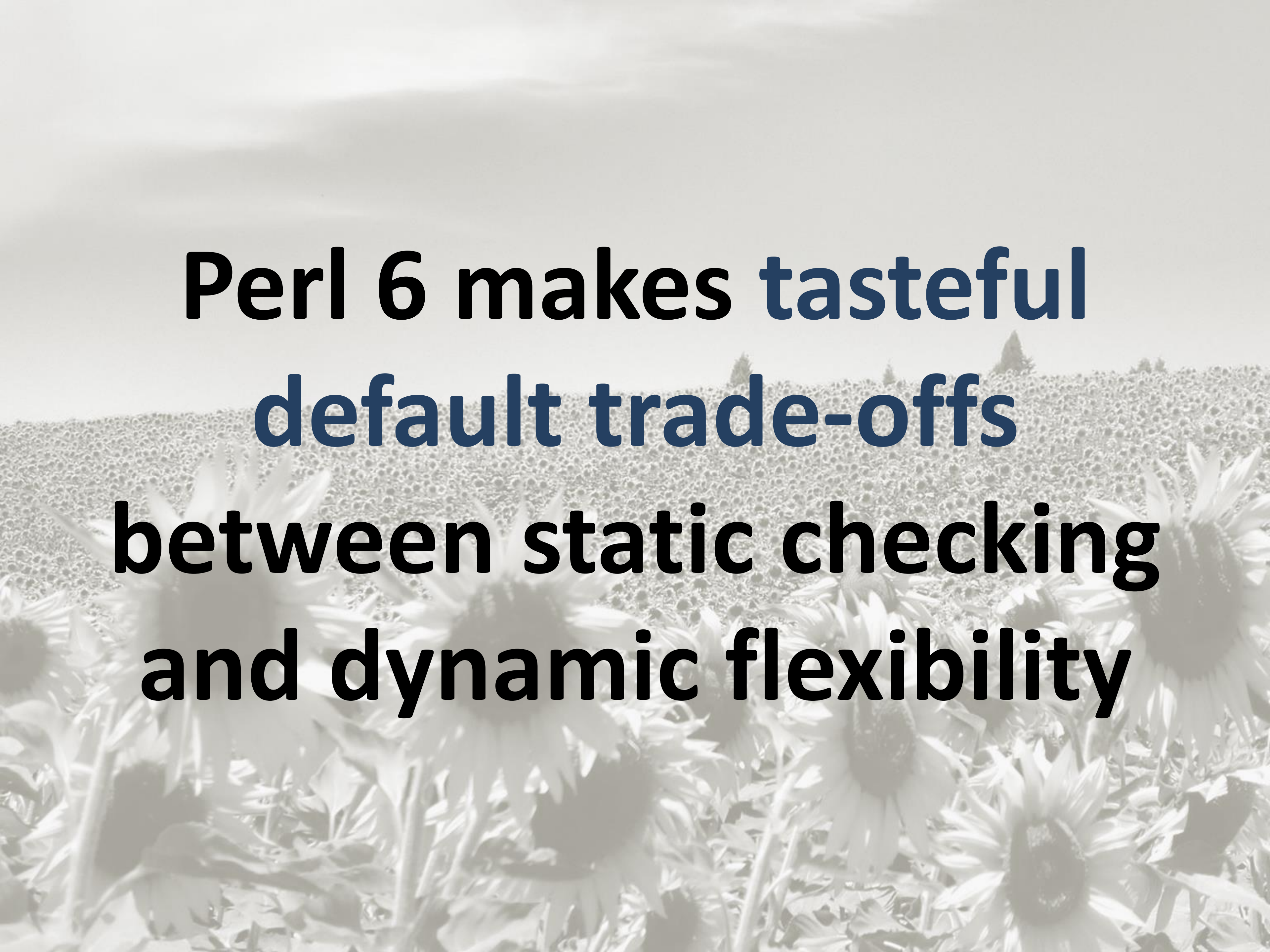
```
===SORRY!===
about lacks a URL template
at y.p6:6
```



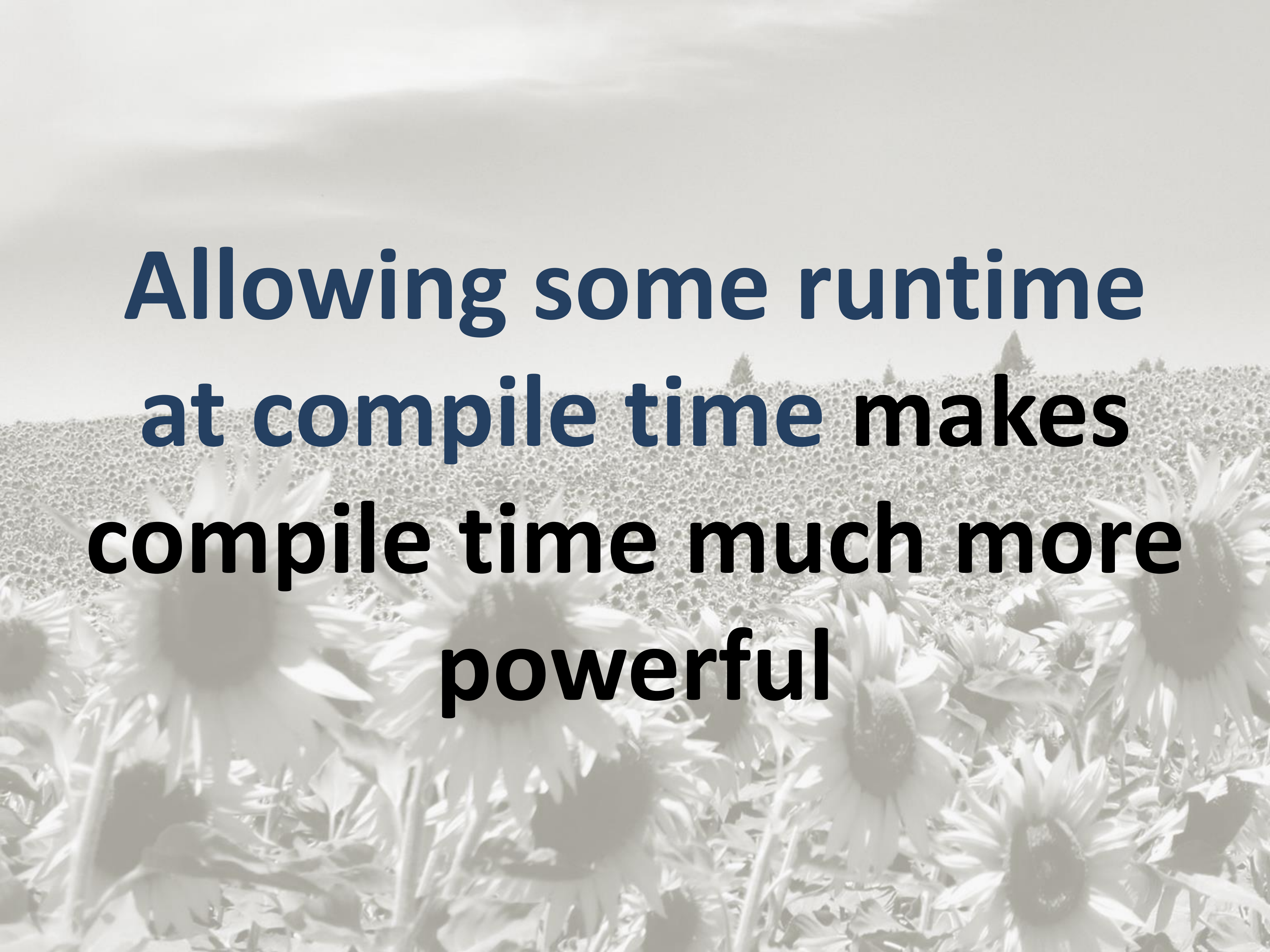
We don't expect the average Perl 6 user to go doing such meta-programming

We are enabling library and framework authors to deliver a better developer experience

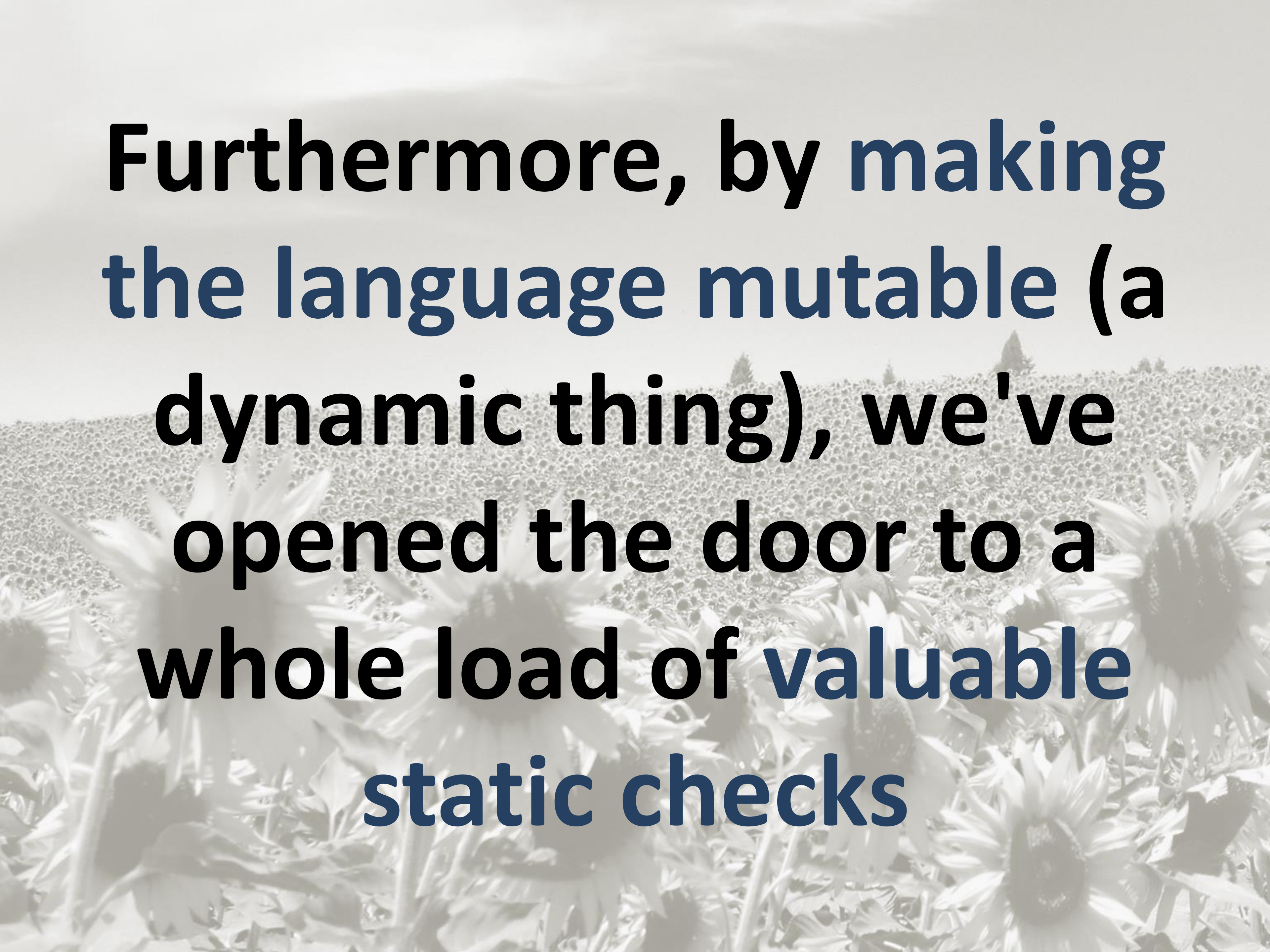
In summary…



Perl 6 makes **tasteful default trade-offs** between static checking and dynamic flexibility



Allowing some runtime at compile time makes compile time much more powerful



Furthermore, by making the language mutable (a dynamic thing), we've opened the door to a whole load of valuable static checks

Questions?