Nonhierarchical OOP

Shawn M Moore
@sartak

Best Practical Solutions

"Object"

Before we get into the thick of it let's talk about the _essence_ of object-oriented programming. What is an object?

The best definition I've heard is that an object has "state", "behavior", and "identity". I got this definition from Pascal Costanza, a Lisper, but he got it from this book. Probably a good book but I haven't read it.

"Object"

- state

Before we get into the thick of it let's talk about the _essence_ of object-oriented programming. What is an object?

The best definition I've heard is that an object has "state", "behavior", and "identity". I got this definition from Pascal Costanza, a Lisper, but he got it from this book. Probably a good book but I haven't read it.

"Object"

- state
- behavior

Before we get into the thick of it let's talk about the _essence_ of object-oriented programming. What is an object?

The best definition I've heard is that an object has "state", "behavior", and "identity". I got this definition from Pascal Costanza, a Lisper, but he got it from this book. Probably a good book but I haven't read it.

"Object"

- state
- behavior
- identity

Before we get into the thick of it let's talk about the _essence_ of object-oriented programming. What is an object?

The best definition I've heard is that an object has "state", "behavior", and "identity". I got this definition from Pascal Costanza, a Lisper, but he got it from this book. Probably a good book but I haven't read it.

"Object"

- state
- behavior
- identity

Object Oriented Design with Applications - Grady Booch

Before we get into the thick of it let's talk about the _essence_ of object-oriented programming. What is an object?

The best definition I've heard is that an object has "state", "behavior", and "identity". I got this definition from Pascal Costanza, a Lisper, but he got it from this book. Probably a good book but I haven't read it.

state

- attributes
- member variables
- properties
- values
- slots

Hopefully you recognize at least one of these terms. It's unfortunate that the OOP community as a whole hasn't embraced a common lexicon for this concept. Each language seems to invent its own terminology.

State is information that each object carries. It doesn't even have to be inside the object's data structure, as Perl's inside-out objects demonstrate.

state

- attributes
- member variables
- properties
- values
- slots
- アット物

Ruby rocks, just sayin

behavior

- methods

I don't know of any language that calls the unit of behavior something else. Yay! Mission accomplished.

identity

Identity is a little more nebulous.

Say we have a merchant named Fanty. And we have another merchant named Mingo. They're not the same merchant, so comparison between them returns false.

So identity basically it means that two objects, even if they have the same state and behavior, are distinct. Just because Fanty and Mingo have the same state - that their job is merchant - and the same behavior - which is no behavior - they are still separate objects. They are different instances. They each were created by separate calls to a constructor.

identity

```
var fanty = {
    job: 'merchant',
};
```


Identity is a little more nebulous.

Say we have a merchant named Fanty. And we have another merchant named Mingo. They're not the same merchant, so comparison between them returns false.

So identity basically it means that two objects, even if they have the same state and behavior, are distinct. Just because Fanty and Mingo have the same state - that their job is merchant - and the same behavior - which is no behavior - they are still separate objects. They are different instances. They each were created by separate calls to a constructor.

identity

```
var fanty = {
    job: 'merchant',
};

var mingo = {
    job: 'merchant',
};
```


Identity is a little more nebulous.

Say we have a merchant named Fanty. And we have another merchant named Mingo. They're not the same merchant, so comparison between them returns false.

So identity basically it means that two objects, even if they have the same state and behavior, are distinct. Just because Fanty and Mingo have the same state - that their job is merchant - and the same behavior - which is no behavior - they are still separate objects. They are different instances. They each were created by separate calls to a constructor.

identity

```
var fanty = {
    job: 'merchant',
};

var mingo = {
    job: 'merchant',
};
```

```
fanty == mingo // false
```


Identity is a little more nebulous.

Say we have a merchant named Fanty. And we have another merchant named Mingo. They're not the same merchant, so comparison between them returns false.

So identity basically it means that two objects, even if they have the same state and behavior, are distinct. Just because Fanty and Mingo have the same state – that their job is merchant – and the same behavior – which is no behavior – they are still separate objects. They are different instances. They each were created by separate calls to a constructor.

identity

```
var fanty = {
    job: 'merchant'x
};

var mingo = {
    job: 'merchant'x
};
```

```
fanty == mingo // false
```


Sorry for the syntax error. I do this a LOT because I'm a Perl guy. Just ask Cornelius and Gugod who keep fixing my Javascript.

identity

```
mingo.job = 'investor';

fanty.job // still merchant
```


If objects did not have identity, then changing Mingo's job might change Fanty's as well, and that would become really confusing. Talk about heisenbugs.

"Object"

- state
- behavior
- identity

So this gives us a very broad definition of object-oriented programming. I'd say this is all you *really* need to call a language object-oriented.

"Object"

Unnecessary:

So what did we leave out?

You don't need classes for OOP. For example, Javascript. Though it has classes, they're kind of a pain. Javascript programmers seem happier to forsake class-based OOP for prototype-based OOP.

You also don't need inheritance. I looked into it, the only existent OOP language that lacked inheritance is early versions of Visual Basic. Inheritance is useful for structuring medium and large programs, for sure, but I could easily envision a modern OOP language without inheritance. We'll talk more about what kind of design it might use.

Finally you also don't need polymorphism, which lets an object act in place of another as long as it is sufficiently similar. There are various ways to do this, such as duck typing, interfaces, and roles.

We'll be talking a lot about these today. They're all certainly useful, don't get me wrong, but it is useful to examine what we stand to gain by stretching them or even ignoring them entirely.

"Object"

Unnecessary:

- classes

So what did we leave out?

You don't need classes for OOP. For example, Javascript. Though it has classes, they're kind of a pain. Javascript programmers seem happier to forsake class-based OOP for prototype-based OOP.

You also don't need inheritance. I looked into it, the only existent OOP language that lacked inheritance is early versions of Visual Basic. Inheritance is useful for structuring medium and large programs, for sure, but I could easily envision a modern OOP language without inheritance. We'll talk more about what kind of design it might use.

Finally you also don't need polymorphism, which lets an object act in place of another as long as it is sufficiently similar. There are various ways to do this, such as duck typing, interfaces, and roles.

We'll be talking a lot about these today. They're all certainly useful, don't get me wrong, but it is useful to examine what we stand to gain by stretching them or even ignoring them entirely.

"Object"

Unnecessary:

- classes
- inheritance

So what did we leave out?

You don't need classes for OOP. For example, Javascript. Though it has classes, they're kind of a pain. Javascript programmers seem happier to forsake class-based OOP for prototype-based OOP.

You also don't need inheritance. I looked into it, the only existent OOP language that lacked inheritance is early versions of Visual Basic. Inheritance is useful for structuring medium and large programs, for sure, but I could easily envision a modern OOP language without inheritance. We'll talk more about what kind of design it might use.

Finally you also don't need polymorphism, which lets an object act in place of another as long as it is sufficiently similar. There are various ways to do this, such as duck typing, interfaces, and roles.

We'll be talking a lot about these today. They're all certainly useful, don't get me wrong, but it is useful to examine what we stand to gain by stretching them or even ignoring them entirely.

"Object"

Unnecessary:

- classes
- inheritance
- polymorphism

So what did we leave out?

You don't need classes for OOP. For example, Javascript. Though it has classes, they're kind of a pain. Javascript programmers seem happier to forsake class-based OOP for prototype-based OOP.

You also don't need inheritance. I looked into it, the only existent OOP language that lacked inheritance is early versions of Visual Basic. Inheritance is useful for structuring medium and large programs, for sure, but I could easily envision a modern OOP language without inheritance. We'll talk more about what kind of design it might use.

Finally you also don't need polymorphism, which lets an object act in place of another as long as it is sufficiently similar. There are various ways to do this, such as duck typing, interfaces, and roles.

We'll be talking a lot about these today. They're all certainly useful, don't get me wrong, but it is useful to examine what we stand to gain by stretching them or even ignoring them entirely.

"Object"

Unnecessary:

- classes
- inheritance
- polymorphism

but useful!

So what did we leave out?

You don't need classes for OOP. For example, Javascript. Though it has classes, they're kind of a pain. Javascript programmers seem happier to forsake class-based OOP for prototype-based OOP.

You also don't need inheritance. I looked into it, the only existent OOP language that lacked inheritance is early versions of Visual Basic. Inheritance is useful for structuring medium and large programs, for sure, but I could easily envision a modern OOP language without inheritance. We'll talk more about what kind of design it might use.

Finally you also don't need polymorphism, which lets an object act in place of another as long as it is sufficiently similar. There are various ways to do this, such as duck typing, interfaces, and roles.

We'll be talking a lot about these today. They're all certainly useful, don't get me wrong, but it is useful to examine what we stand to gain by stretching them or even ignoring them entirely.

Inheritance

Inheritance is definitely a good way to enable code reuse and extensibility. As long as your code is well-factored, users will be able to reuse your code and change the parts of it that they need to.

Inheritance

- reuse

Inheritance is definitely a good way to enable code reuse and extensibility. As long as your code is well-factored, users will be able to reuse your code and change the parts of it that they need to.

Inheritance

- reuse
- extensibility

Inheritance is definitely a good way to enable code reuse and extensibility. As long as your code is well-factored, users will be able to reuse your code and change the parts of it that they need to.

Inheritance

```
Devel::StackTrace
```


Devel::StackTrace is a Perl module that provides a nice interface for creating and inspecting stack traces. It gives you an object-oriented API for looking at a stack and each of the frames inside it.

Inheritance

```
Devel::StackTrace
```

```
  Devel::StackTrace::WithLexicals
```


I extended it using the black-magic module PadWalker to produce Devel::StackTrace::WithLexicals. This captures the lexical variables of each stack frame. Handy for writing a debugger or a more advanced error display.

Extending Devel::StackTrace let me reuse its code to capture a stack trace, and whatnot. It also let me override individual pieces, like extending the capturing to include PadWalker's lexical inspection. Finally, polymorphism lets anyone use Devel::StackTrace::WithLexicals in place of Devel::StackTrace and everything will just work, since Devel::StackTrace::WithLexicals fulfills all of Devel::StackTrace's API.

Inheritance

So inheritance is pretty good!

Inheritance

```
Devel::StackTrace
```

```
    Devel::StackTrace::Profiled
```


But it's not useful in all cases. Inheritance has its limits. Say we wanted to create a new subclass of Devel::StackTrace that includes timing information for each stack frame.

Inheritance

```
Devel::StackTrace::WithLexicals
::AndProfile
```

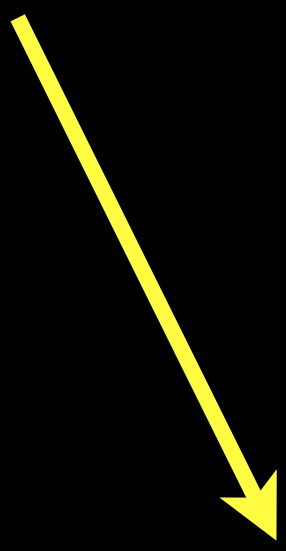

What if we wanted to have lexical variables AND a profile in the stack trace?

Inheritance

Unfortunately we can only inherit from ONE of these classes.

Inheritance

Devel::StackTrace::WithLexicals



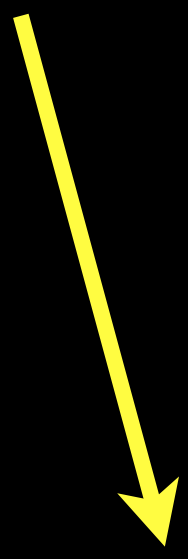
Devel::StackTrace::Profiled

Devel::StackTrace::WithLexicals
::AndProfile

Unfortunately we can only inherit from ONE of these classes.

Inheritance

Devel::StackTrace::WithLexicals



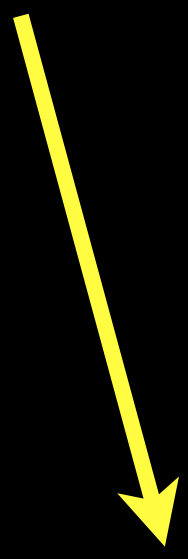
+ Profiled

Devel::StackTrace::WithLexicals
::AndProfile

We could subclass WithLexicals and reimplement Profiled...

Inheritance

Devel::StackTrace::Profiled



+ Lexicals

Devel::StackTrace::WithLexicals
::AndProfile

Or we could do it the other way around, we could subclass Profiled and reimplement WithLexicals...

Inheritance

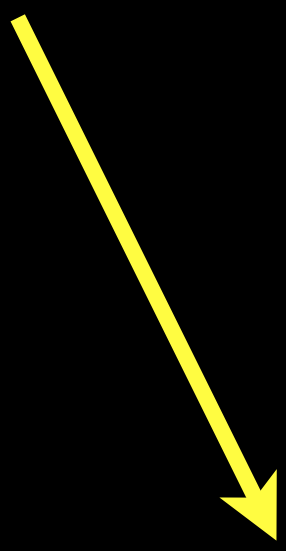
This sucks, we can't reuse the code we've already implemented! Reusing code is one of the two major ideals of OOP. So simple inheritance is not powerful enough for real-world problems that we encounter.

Multiple Inheritance

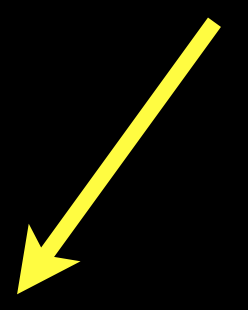
So we can just say that Devel::StackTrace::WithLexicals::AndProfile subclasses both WithLexicals and Profiled. That way we can reuse the code of both modules.

Multiple Inheritance

Devel::StackTrace::WithLexicals



Devel::StackTrace::Profiled



Devel::StackTrace::WithLexicals
::AndProfile

So we can just say that Devel::StackTrace::WithLexicals::AndProfile subclasses both WithLexicals and Profiled. That way we can reuse the code of both modules.

Multiple Inheritance

But this probably won't actually work. WithLexicals and Profiled need to override the same _record_caller_data method to capture information about each stack frame.

Multiple Inheritance

```
Devel::StackTrace::WithLexicals
 after _record_caller_data {
     ...
 }
```


But this probably won't actually work. WithLexicals and Profiled need to override the same _record_caller_data method to capture information about each stack frame.

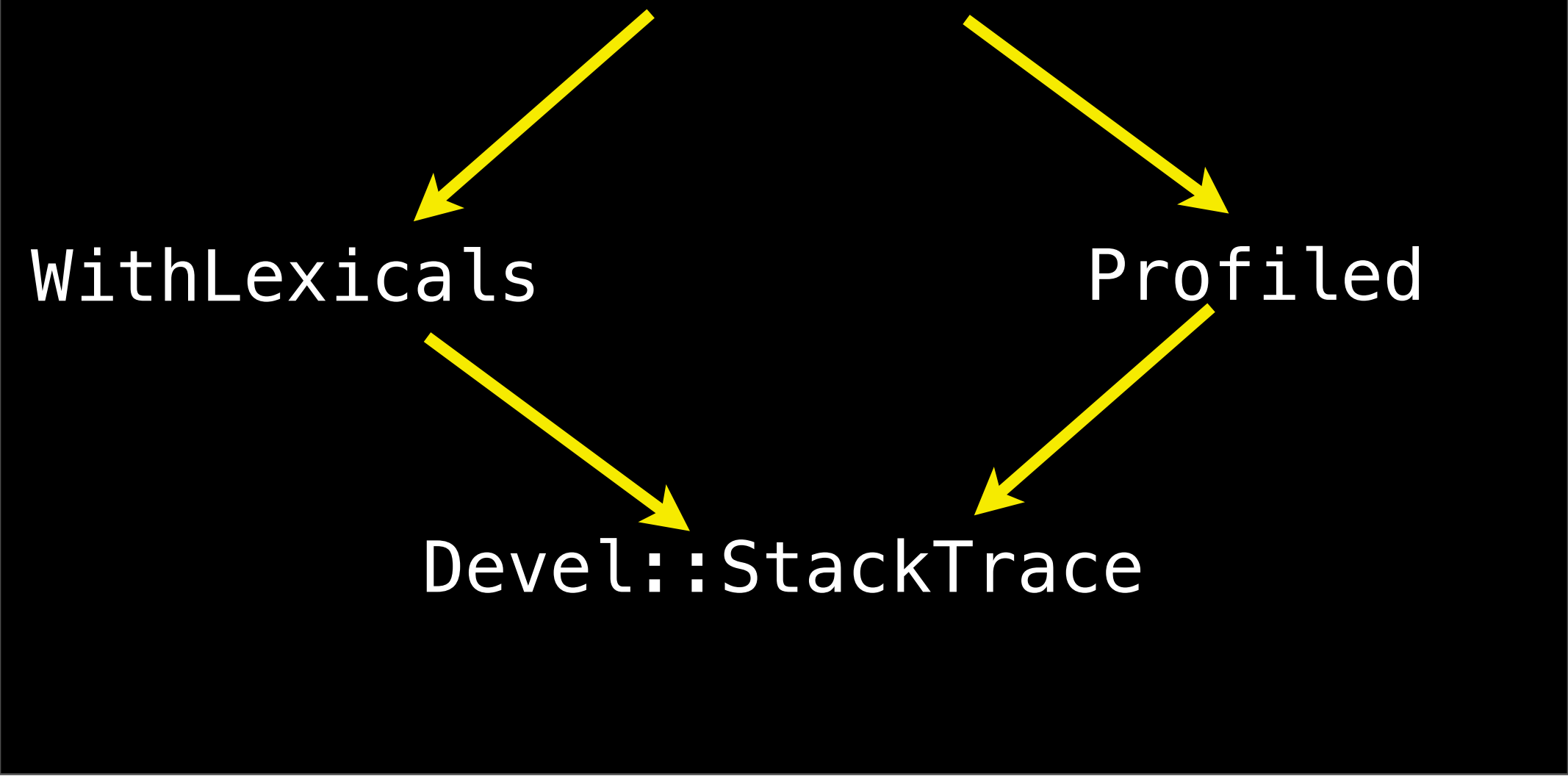
Multiple Inheritance

```
Devel::StackTrace::WithLexicals
after _record_caller_data {
    ...
}
```

```
Devel::StackTrace::Profiled
after _record_caller_data {
    ...
}
```


But this probably won't actually work. WithLexicals and Profiled need to override the same _record_caller_data method to capture information about each stack frame.

Multiple Inheritance

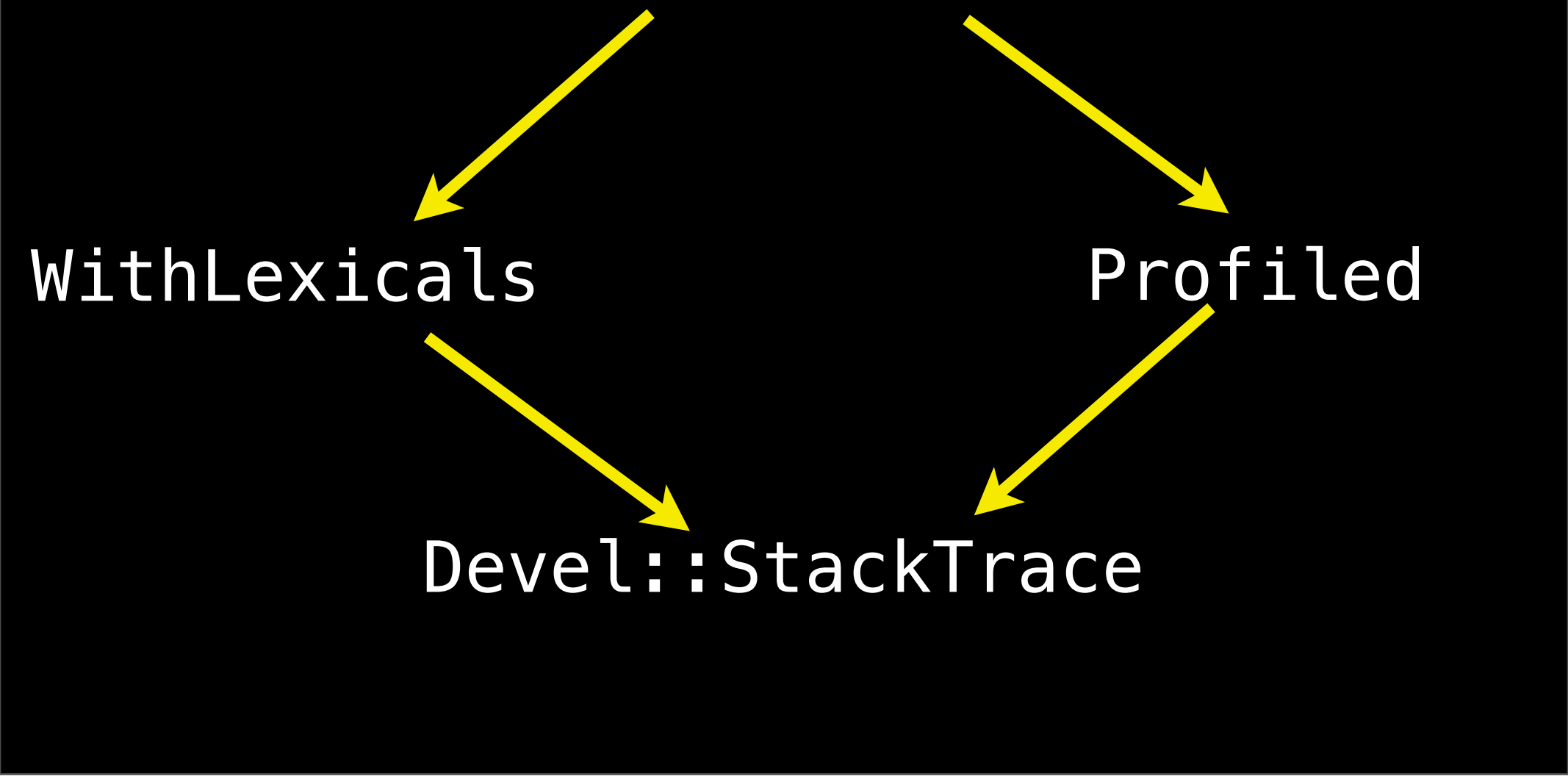


This is the well-known "diamond problem".

When we call _record_caller_data, we have to call both WithLexicals and Profiled's overrides or we lose information.

Multiple Inheritance

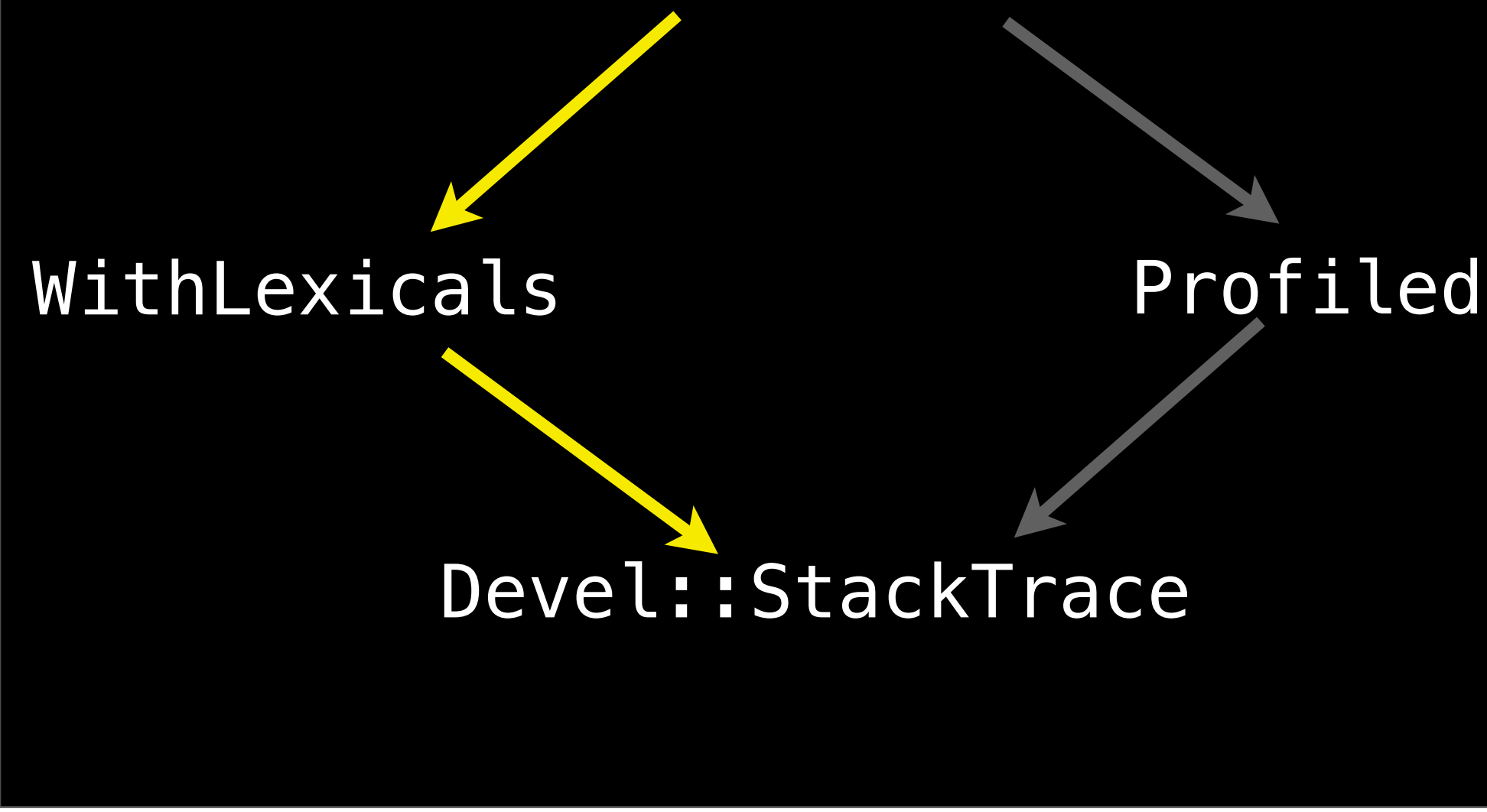
```
$self->_record_caller_data($frame);
```



This is the well-known "diamond problem".

When we call _record_caller_data, we have to call both WithLexicals and Profiled's overrides or we lose information.

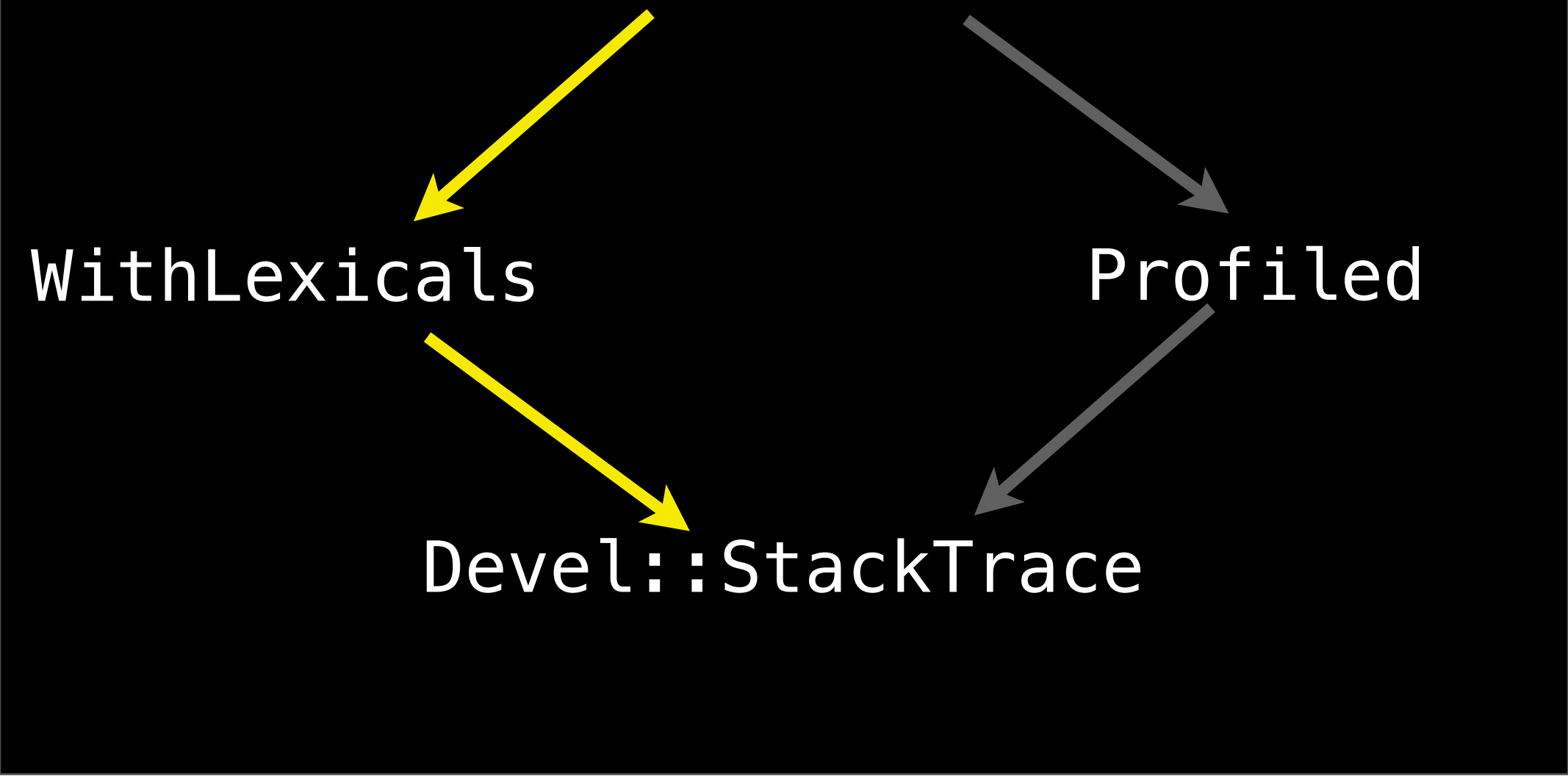
Multiple Inheritance



So typically when you call _record_caller_data, you only call one of the two methods. If we only call WithLexicals, then we won't get timing.

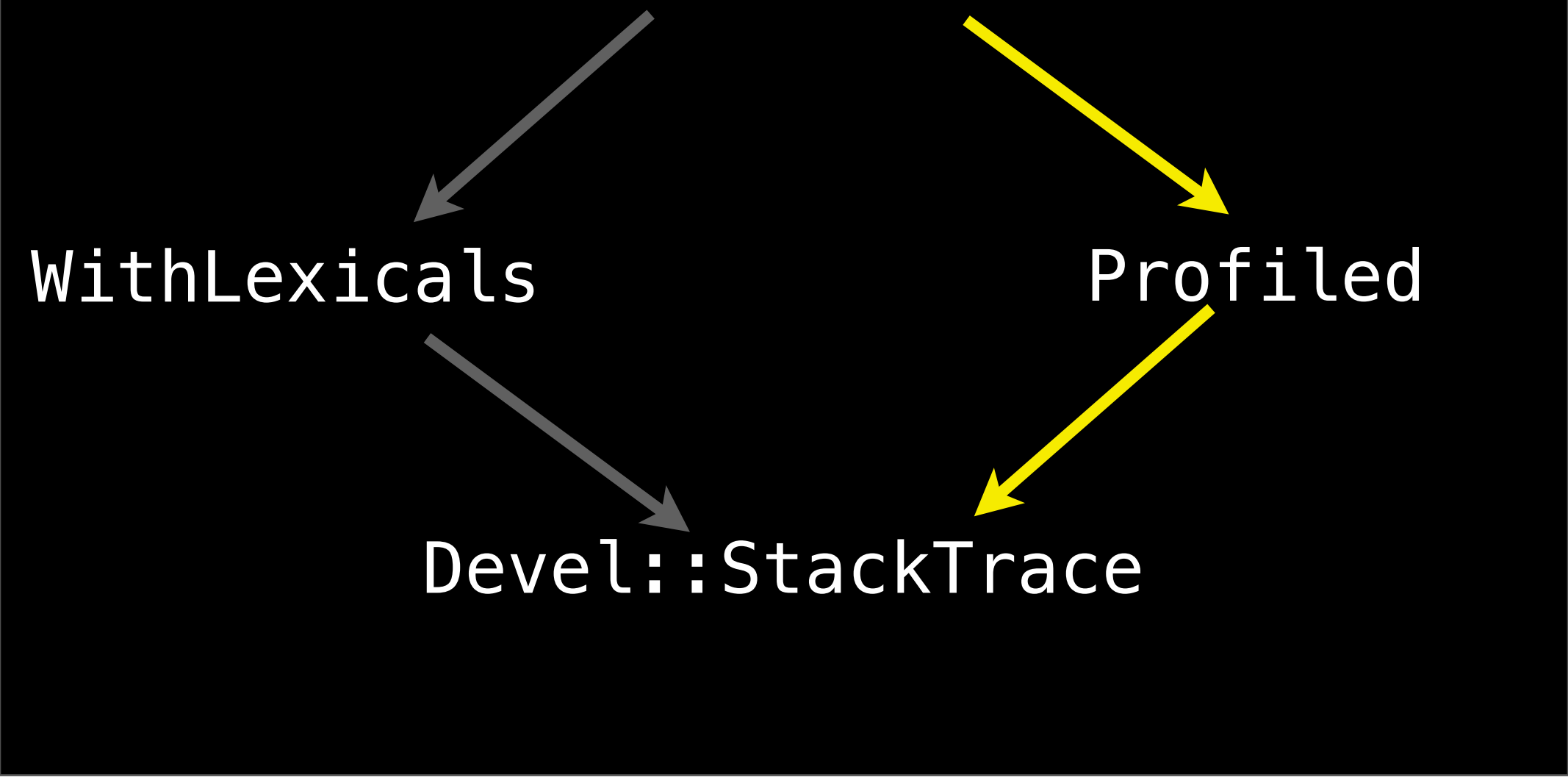
Multiple Inheritance

```
$self->_record_caller_data($frame);
```



So typically when you call _record_caller_data, you only call one of the two methods. If we only call WithLexicals, then we won't get timing.

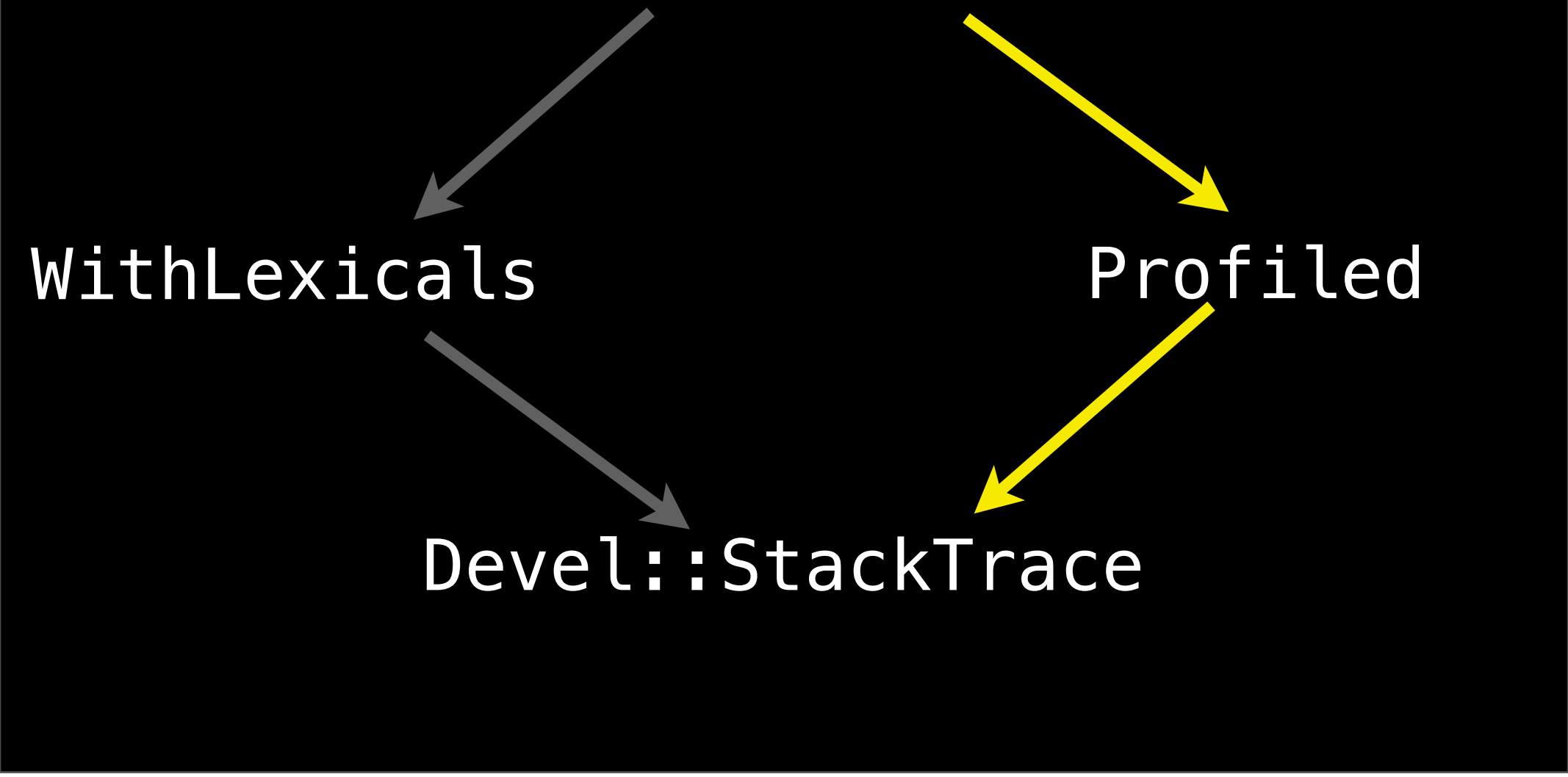
Multiple Inheritance



If we only call Profiled then we won't get the lexical variables.

Multiple Inheritance

```
$self->_record_caller_data($frame);
```



If we only call Profiled then we won't get the lexical variables.

Multiple Inheritance

One way to fix this is to explicitly call each of our superclasses' methods.

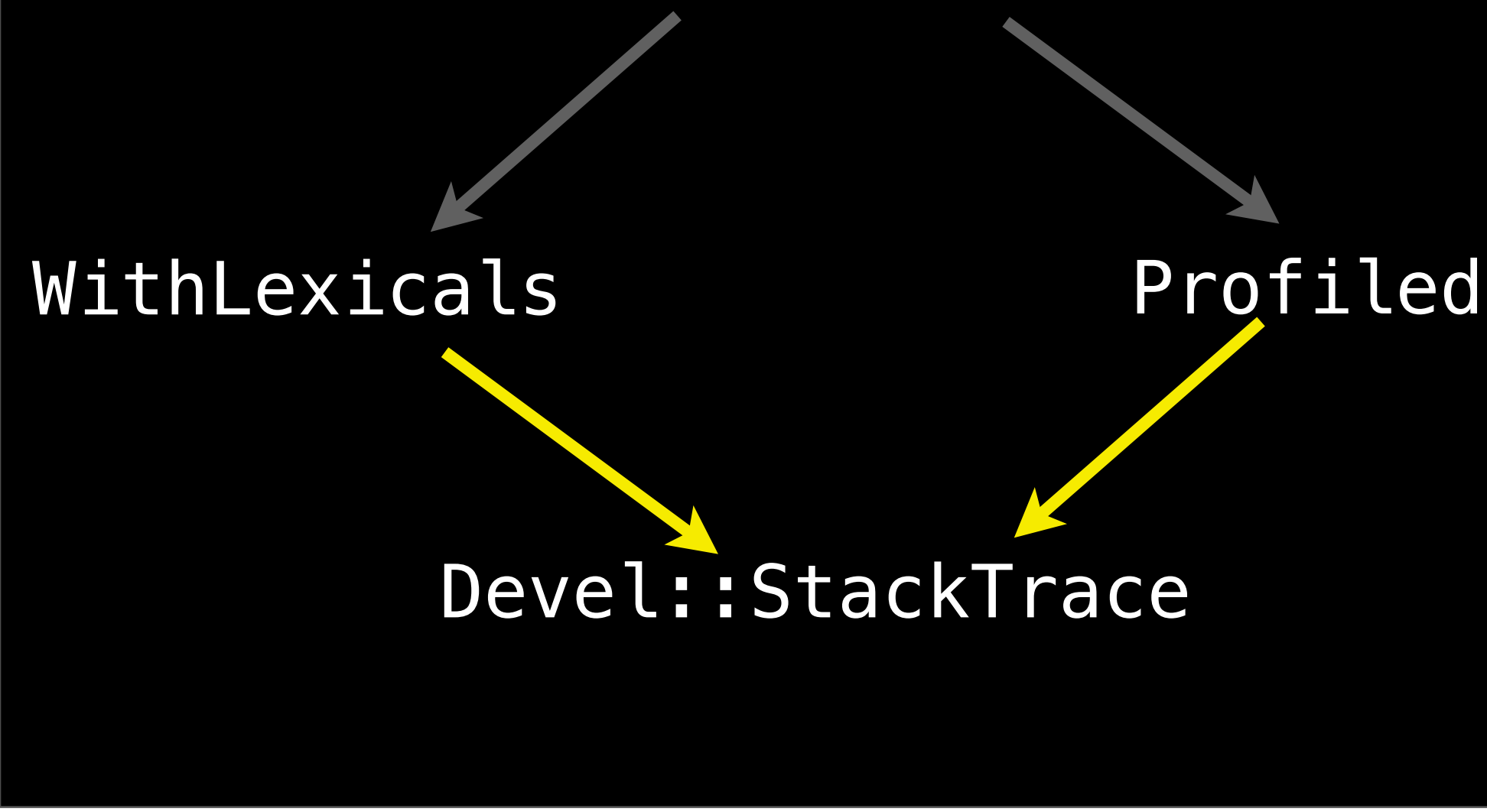
Multiple Inheritance

```
Devel::StackTrace::WithLexicals::AndProfile
```

```
method _record_caller_data {
    $self->WithLexicals::_record_caller_data(@_);
    $self->Profiled::_record_caller_data(@_);
}
```


One way to fix this is to explicitly call each of our superclasses' methods.

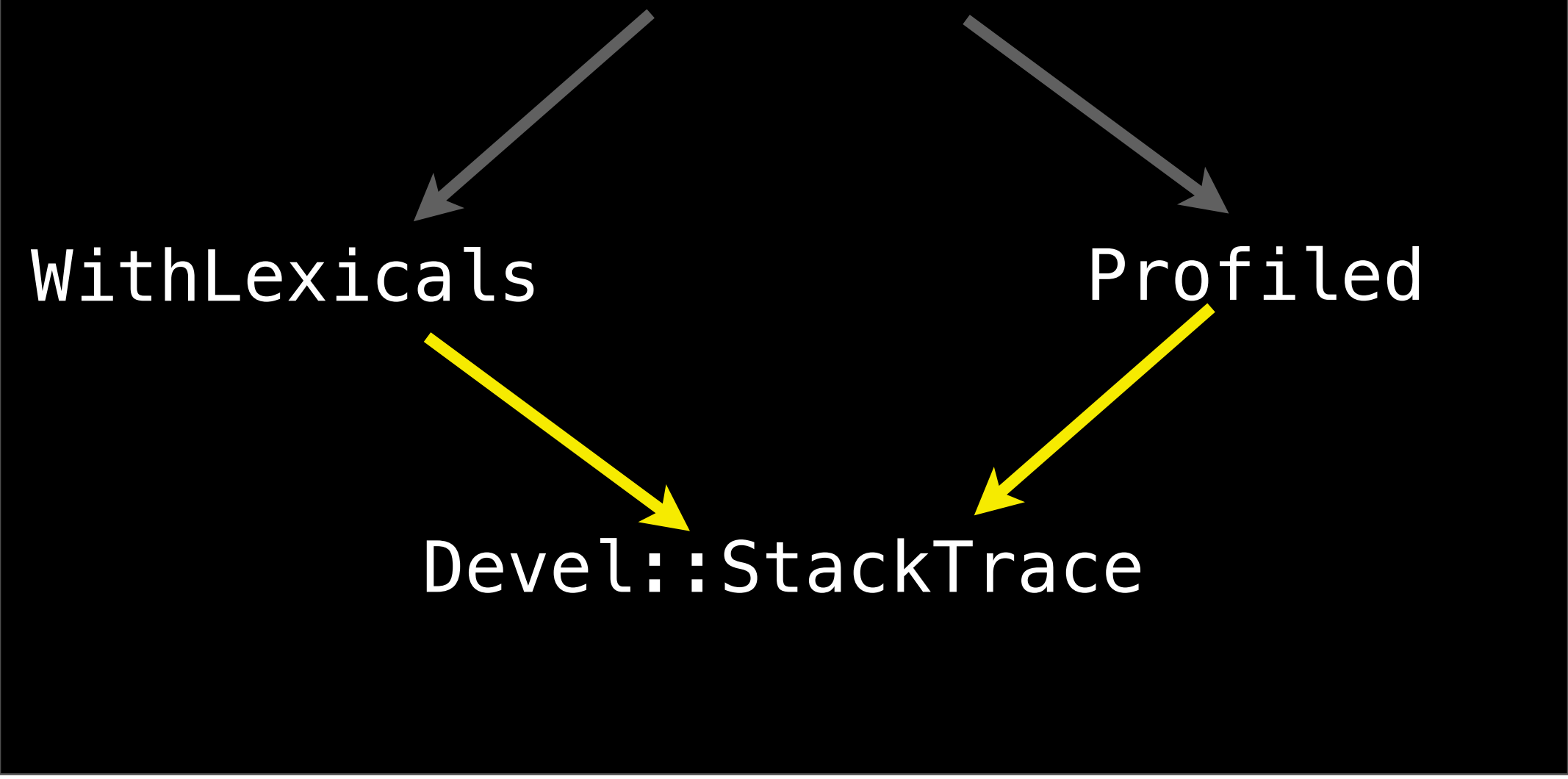
Multiple Inheritance



But this sucks because now the Devel::StackTrace's capturing is called twice! Both of the intermediate classes call it. This could cause all sorts of problems. Even if it had no side effects, it would also be a little slower.

Multiple Inheritance

```
$self->_record_caller_data($frame);
```



But this sucks because now the Devel::StackTrace's capturing is called twice! Both of the intermediate classes call it. This could cause all sorts of problems. Even if it had no side effects, it would also be a little slower.

Multiple Inheritance

Furthermore this is a maintenance sink. It breaks encapsulation. Just sucks.

Multiple Inheritance

```
Devel::StackTrace::WithLexicals::AndProfile
```

```
method _record_caller_data {
    $self->WithLexicals::_record_caller_data(@_);
    $self->Profiled::_record_caller_data(@_);
}
```


Furthermore this is a maintenance sink. It breaks encapsulation. Just sucks.

Multiple Inheritance

Multiple inheritance is a pretty naive attempt at making OOP more flexible. It's just no good.

Multiple Inheritance

Multiple inheritance is a pretty naive attempt at making OOP more flexible. It's just no good.

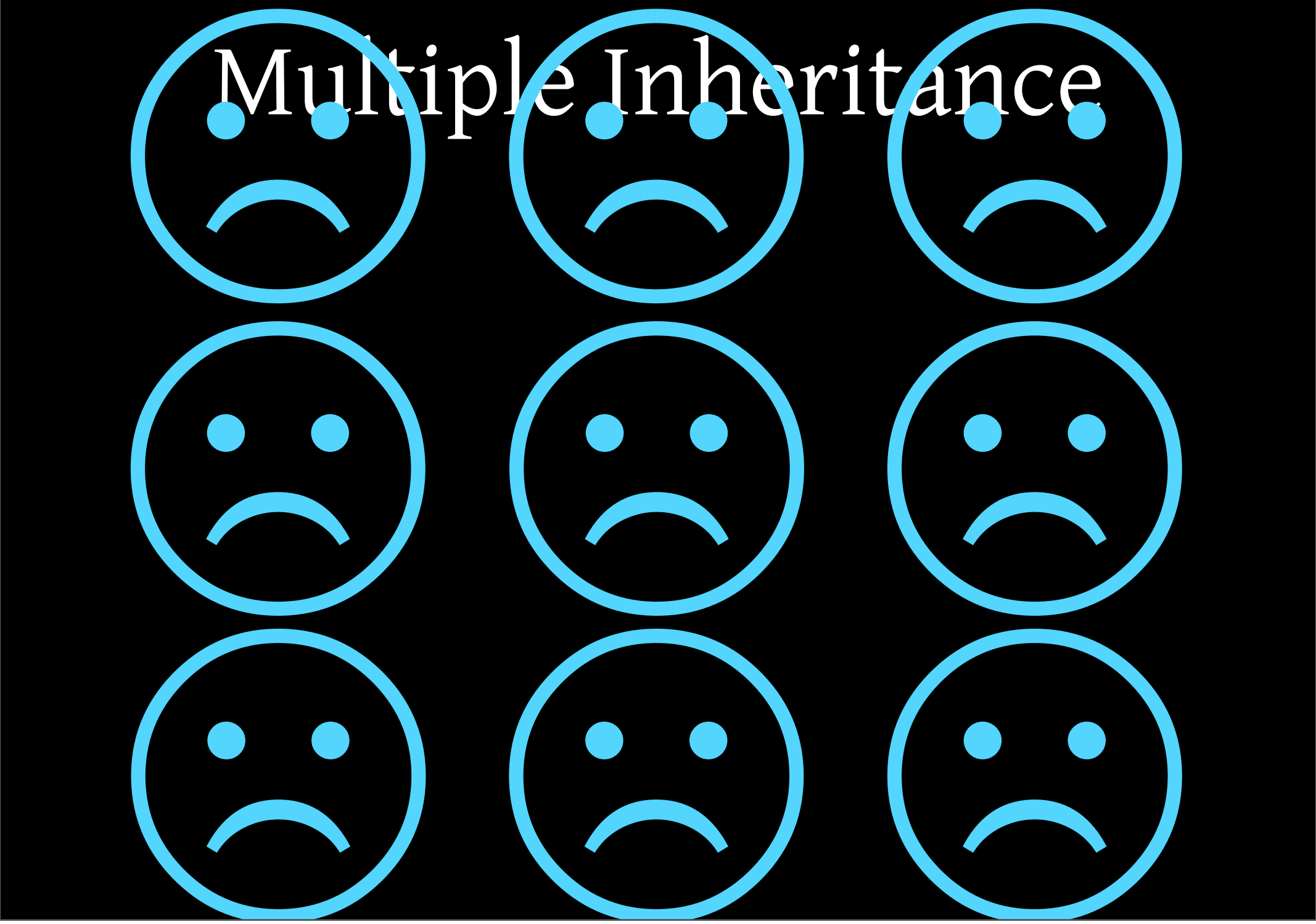
Multiple Inheritance

Multiple inheritance is a pretty naive attempt at making OOP more flexible. It's just no good.

Multiple Inheritance

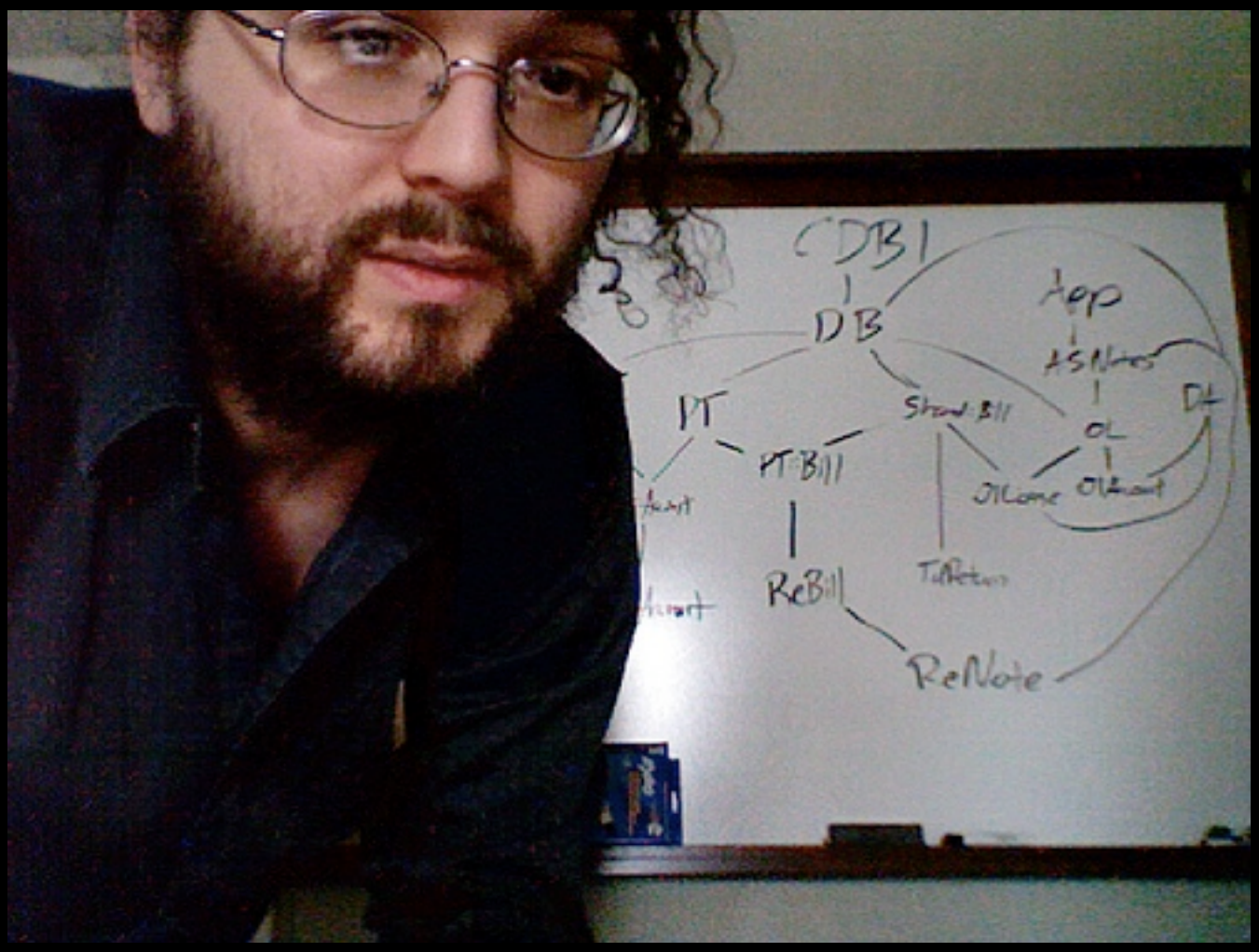
Multiple inheritance is a pretty naive attempt at making OOP more flexible. It's just no good.

Multiple Inheritance



Multiple inheritance is a pretty naive attempt at making OOP more flexible. It's just no good.

Multiple Inheritance



http://www.flickr.com/photos/xwrn/525198268/

Look at how happy Schwern is that he's using multiple inheritance.

MULTIPLE INHERITANCE MUST DIE

Multiple Inheritance

- Ruby++
- Java++
- C#++

Thankfully a lot of languages are eschewing multiple inheritance. So we're making progress.

Multiple Inheritance

- Ruby++
- Java++
- C#++
- PHP++

Thankfully a lot of languages are eschewing multiple inheritance. So we're making progress.

Multiple Inheritance

- Ruby++
- Java++
- C#++
- PHP++
- Perl--
- Python--
- CLOS--

Thankfully a lot of languages are eschewing multiple inheritance. So we're making progress.

Multiple Inheritance

- Ruby++
- Java++
- C#++
- PHP++
- Perl--
- Python--
- CLOS--
- C++--

Thankfully a lot of languages are eschewing multiple inheritance. So we're making progress.

Other Solutions

- Mixins
- Traits/Roles

There are two primary solutions for the diamond problem. There are mixins, which are prominently featured in Ruby, but also exist in Python and many other languages. And there are traits/roles, which are new, but have been implemented in Smalltalk, Perl, Scala, and Fortress.

Mixins

In short, a mixin is a module. Mixins are not classes, so you are not allowed to instantiate them. And its superclass is parameterized, which is the real magic of mixins. Mixins still make use of inheritance, but each instance of a mixin has its own superclass. This can replace some designs where you would need to resort to multiple inheritance.

Mixins

- Module

In short, a mixin is a module. Mixins are not classes, so you are not allowed to instantiate them. And its superclass is parameterized, which is the real magic of mixins. Mixins still make use of inheritance, but each instance of a mixin has its own superclass. This can replace some designs where you would need to resort to multiple inheritance.

Mixins

- Module
- No instantiation (no "new")

In short, a mixin is a module. Mixins are not classes, so you are not allowed to instantiate them. And its superclass is parameterized, which is the real magic of mixins. Mixins still make use of inheritance, but each instance of a mixin has its own superclass. This can replace some designs where you would need to resort to multiple inheritance.

Mixins

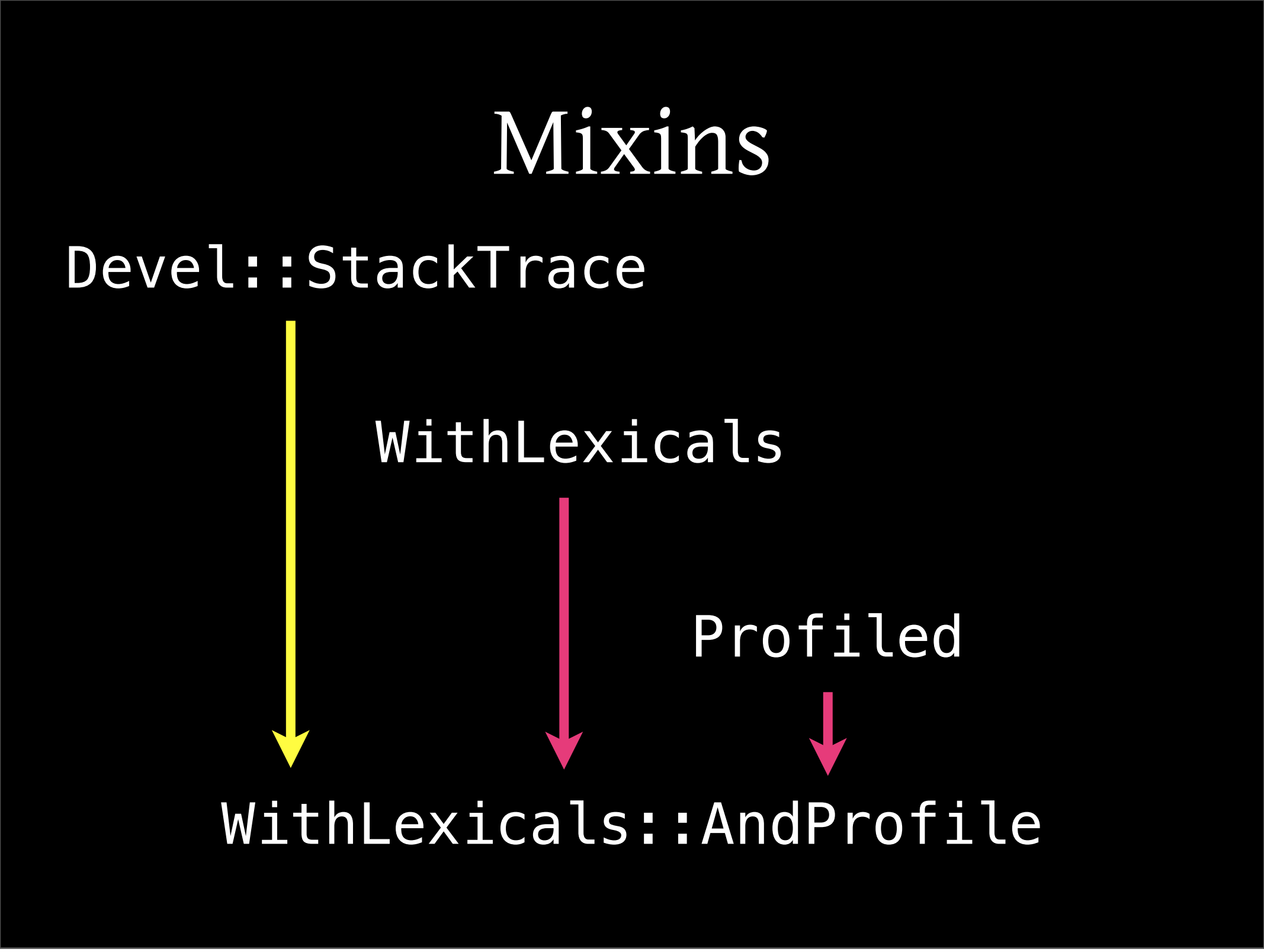
- Module
- No instantiation (no "new")
- Parameterized superclass

In short, a mixin is a module. Mixins are not classes, so you are not allowed to instantiate them. And its superclass is parameterized, which is the real magic of mixins. Mixins still make use of inheritance, but each instance of a mixin has its own superclass. This can replace some designs where you would need to resort to multiple inheritance.

Mixins

If WithLexicals and Profiled are mixins, then our subclass can use both of them. Because a mixin lets you choose its superclass, there is no diamond. It's all just single inheritance so everything works.

Mixins



If WithLexicals and Profiled are mixins, then our subclass can use both of them. Because a mixin lets you choose its superclass, there is no diamond. It's all just single inheritance so everything works.

Mixins

The _record_caller_data overrides of both Profiled and WithLexicals are called, and the original Devel::StackTrace method is called only once. This is exactly what we want.

Mixins

```
$self->_record_caller_data($frame);
```


The _record_caller_data overrides of both Profiled and WithLexicals are called, and the original Devel::StackTrace method is called only once. This is exactly what we want.

Mixins

```
$self->_record_caller_data($frame);
```

```
Profiled
```


The _record_caller_data overrides of both Profiled and WithLexicals are called, and the original Devel::StackTrace method is called only once. This is exactly what we want.

Mixins

```
$self->_record_caller_data($frame);
```

Profiled

WithLexicals

The _record_caller_data overrides of both Profiled and WithLexicals are called, and the original Devel::StackTrace method is called only once. This is exactly what we want.

Mixins

```
$self->_record_caller_data($frame);
```

Profiled

WithLexicals

Devel::StackTrace

The _record_caller_data overrides of both Profiled and WithLexicals are called, and the original Devel::StackTrace method is called only once. This is exactly what we want.

Mixins

Mixins are a great improvement over multiple inheritance.

Mixins

Mixins reuse the inheritance mechanism. On one hand, this is nice because inheritance has been around for a while, and people understand it. But mixins use inheritance in an unusual way, so there are some gotchas in the details.

Mixins

- uses inheritance

Mixins reuse the inheritance mechanism. On one hand, this is nice because inheritance has been around for a while, and people understand it. But mixins use inheritance in an unusual way, so there are some gotchas in the details.

Mixins

- uses inheritance
- 

Mixins reuse the inheritance mechanism. On one hand, this is nice because inheritance has been around for a while, and people understand it. But mixins use inheritance in an unusual way, so there are some gotchas in the details.

Mixins

- uses inheritance
- 
- 

Mixins reuse the inheritance mechanism. On one hand, this is nice because inheritance has been around for a while, and people understand it. But mixins use inheritance in an unusual way, so there are some gotchas in the details.

Mixins

Let's say we define a mixin for recording lexicals for Devel::StackTrace. We wrap the method that collects data for each frame, which calls a helper method for actually extracting the lexical variables from the stack frame.

Mixins

```
mixin WithLexicals
```


Let's say we define a mixin for recording lexicals for Devel::StackTrace. We wrap the method that collects data for each frame, which calls a helper method for actually extracting the lexical variables from the stack frame.

Mixins

```
mixin WithLexicals

after _record_caller_data {
    $self->_record_extra_data
}
```


Let's say we define a mixin for recording lexicals for Devel::StackTrace. We wrap the method that collects data for each frame, which calls a helper method for actually extracting the lexical variables from the stack frame.

Mixins

```
mixin WithLexicals

after _record_caller_data {
    $self->_record_extra_data
}

method _record_extra_data {
    # extract lexicals
}
```


Let's say we define a mixin for recording lexicals for Devel::StackTrace. We wrap the method that collects data for each frame, which calls a helper method for actually extracting the lexical variables from the stack frame.

Mixins

```
mixin Profiled

after _record_caller_data {
    $self->_record_extra_data
}

method _record_extra_data {
    # profiling
}
```


And we do the exact same with Profiled.

Mixins

```
$self->_record_extra_data($frame);
```


When the Profiled mixin's wrapper is called, it calls _record_extra_data. The Profiled mixin's _record_extra_data method is called. But not WithLexical's.

Mixins

```
$self->_record_extra_data($frame);
```

```
Profiled
```


When the Profiled mixin's wrapper is called, it calls _record_extra_data. The Profiled mixin's _record_extra_data method is called. But not WithLexical's.

Mixins

```
$self->_record_extra_data($frame);
```

Profiled

WithLexicals

When the Profiled mixin's wrapper is called, it calls _record_extra_data. The Profiled mixin's _record_extra_data method is called. But not WithLexical's.

Mixins

```
$self->_record_extra_data($frame);
```


When the WithLexical's mixin's wrapper is called, it again calls _record_extra_data. But there's only a single namespace for methods! The Profiled mixin's _record_extra_data method is called again. But not WithLexical's.

Mixins

```
$self->_record_extra_data($frame);
```

```
Profiled
```


When the WithLexical's mixin's wrapper is called, it again calls _record_extra_data. But there's only a single namespace for methods! The Profiled mixin's _record_extra_data method is called again. But not WithLexical's.

Mixins

```
$self->_record_extra_data($frame);
```

Profiled

WithLexicals

When the WithLexical's mixin's wrapper is called, it again calls _record_extra_data. But there's only a single namespace for methods! The Profiled mixin's _record_extra_data method is called again. But not WithLexical's.

Mixins

So mixins have a problem similar to diamond inheritance.

Mixins

Since mixins are better than MI they get a smaller sad face

Roles

Roles, which are also called traits, are similar to mixins. A role is a module, so it is a collection of methods and other stuff.

You can't instantiate roles, they have no "new" method.

And the way roles are different from mixins are that they use a new kind of "composition" instead of inheritance.

Roles

- Module

Roles, which are also called traits, are similar to mixins. A role is a module, so it is a collection of methods and other stuff.

You can't instantiate roles, they have no "new" method.

And the way roles are different from mixins are that they use a new kind of "composition" instead of inheritance.

Roles

- Module
- No instantiation (no "new")

Roles, which are also called traits, are similar to mixins. A role is a module, so it is a collection of methods and other stuff.

You can't instantiate roles, they have no "new" method.

And the way roles are different from mixins are that they use a new kind of "composition" instead of inheritance.

Roles

- Module
- No instantiation (no "new")
- Composition not inheritance

Roles, which are also called traits, are similar to mixins. A role is a module, so it is a collection of methods and other stuff.

You can't instantiate roles, they have no "new" method.

And the way roles are different from mixins are that they use a new kind of "composition" instead of inheritance.

Roles

- Originally in Smalltalk
- in 2002

Many of you have probably never heard of roles. Roles (or traits) were designed in 2002 using Smalltalk. 2002 is a pretty long time ago.

It's ancient history as far as the internet is concerned.

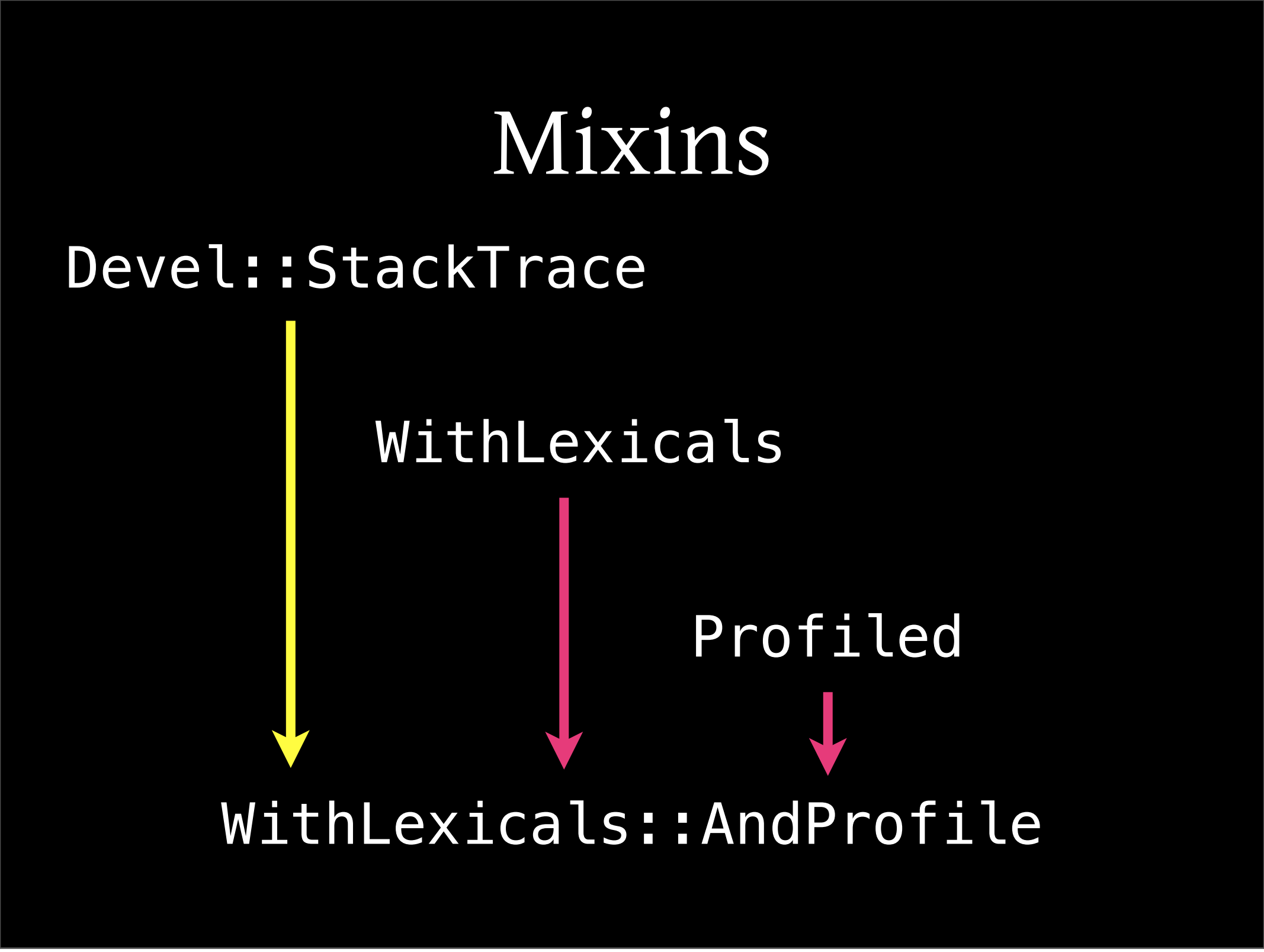
Lisp

- if-then-else
- recursive functions
- garbage collection
- first-class functions
- closures
- dynamic typing
- interactive programming

But 2002 is still cutting edge in terms of language design.

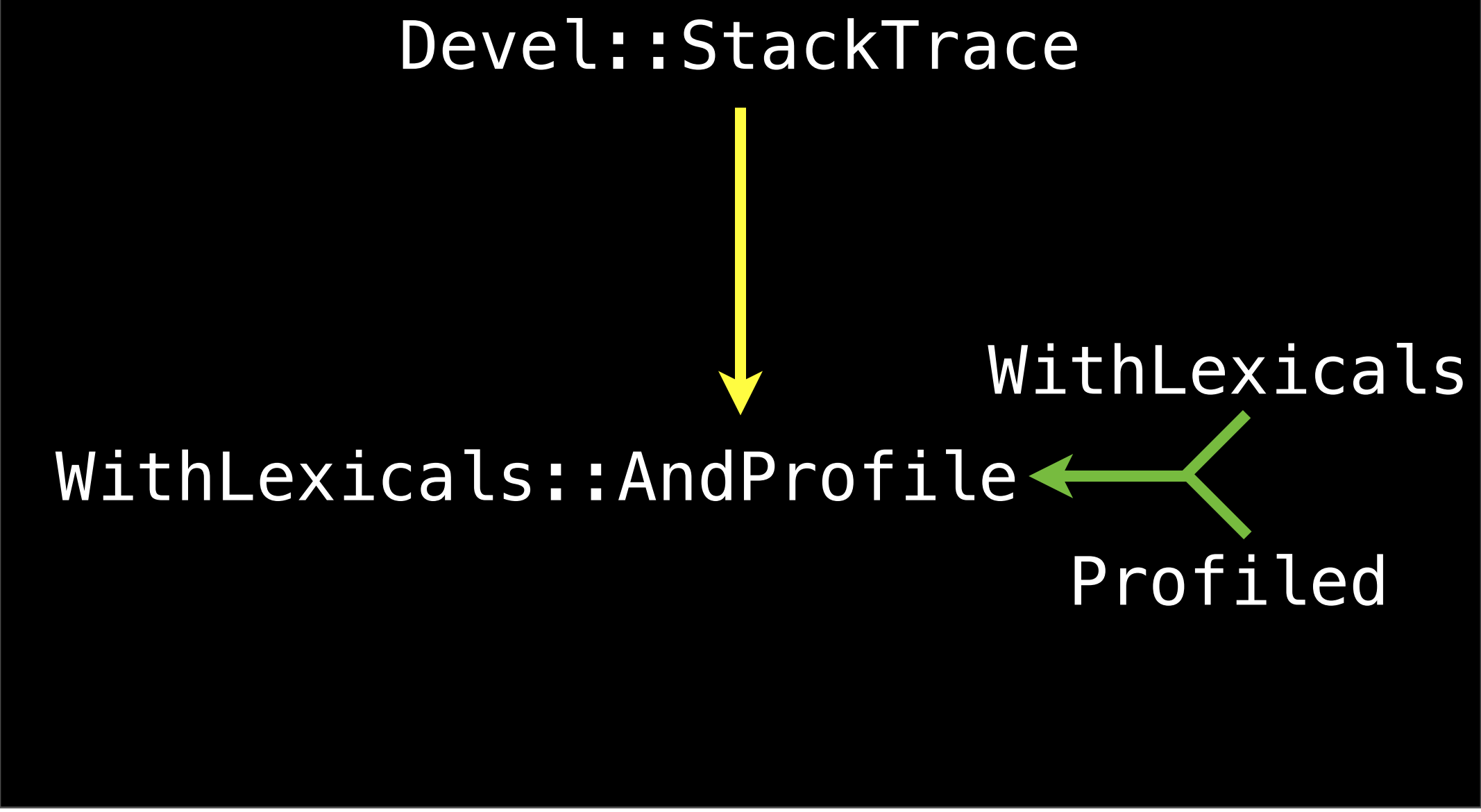
Lisp stole all the good ideas in the 60s and 70s. These are all things we take for granted nowadays, but most of these took a long time to get into the mainstream. Only in the last decade or two have these ideas took hold.

Mixins



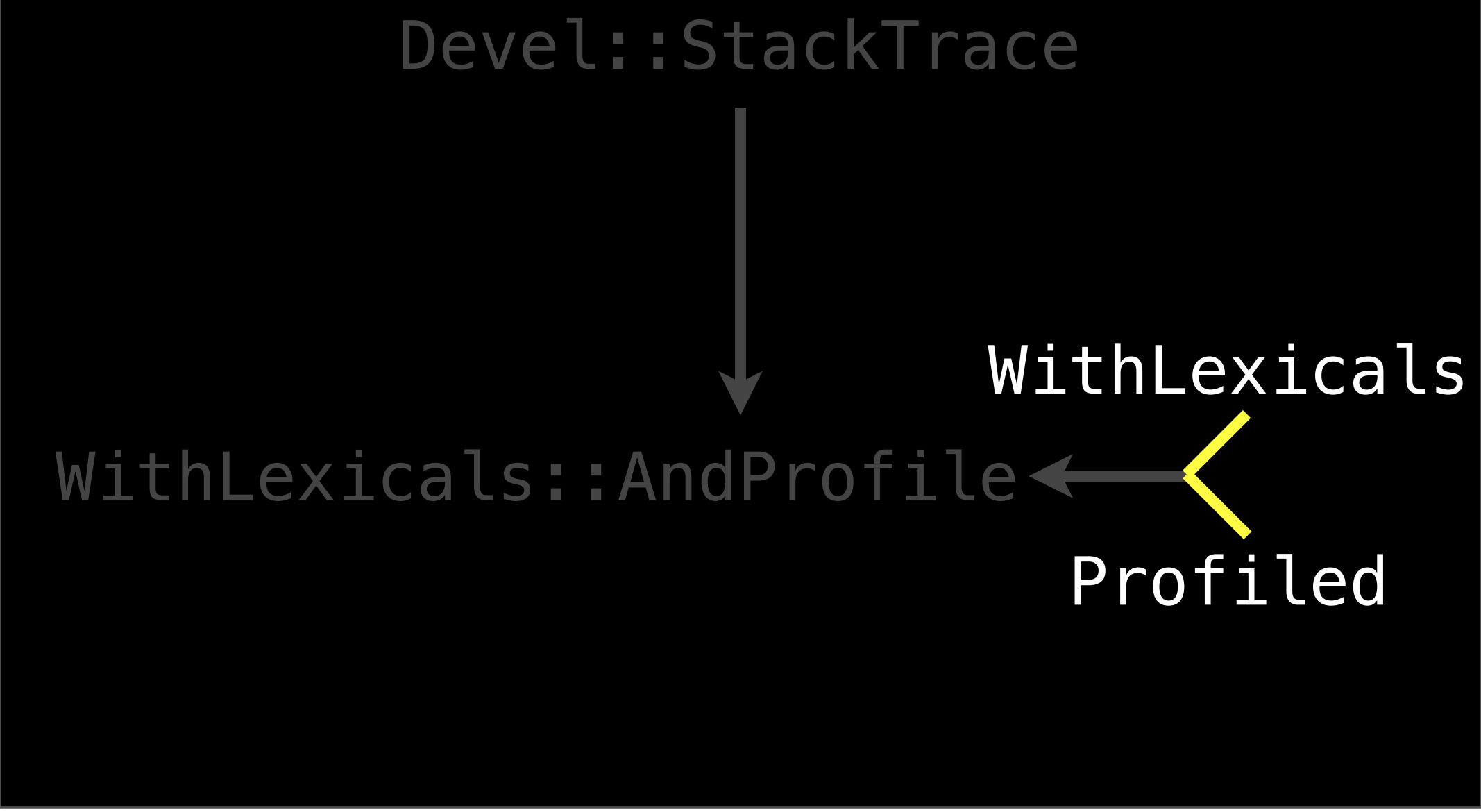
So this is what mixins look like. They use inheritance which is vertical.

Roles



Roles are horizontal instead. When a class consumes a role, it does not inherit from it, but we say the class "consumes" the role. It's kind of like copying and pasting the code from the role into the class. We call that "flattening" since it's not using inheritance, it puts the code directly into the class.

Roles



So this part is interesting. When a class consumes multiple roles, they are first combined into a single role. Again this is like copying and pasting, but with some protection.

Roles

So we have a role. It has a method modifier on the frame capture method. Which calls its own _record_extra_data method.

Roles

```
role WithLexicals
```


So we have a role. It has a method modifier on the frame capture method. Which calls its own _record_extra_data method.

Roles

```
role WithLexicals

after _record_caller_data {
    $self->_record_extra_data
}
```


So we have a role. It has a method modifier on the frame capture method. Which calls its own _record_extra_data method.

Roles

```
role WithLexicals

after _record_caller_data {
    $self->_record_extra_data
}

method _record_extra_data {
    # extract lexicals
}
```


So we have a role. It has a method modifier on the frame capture method. Which calls its own _record_extra_data method.

Roles

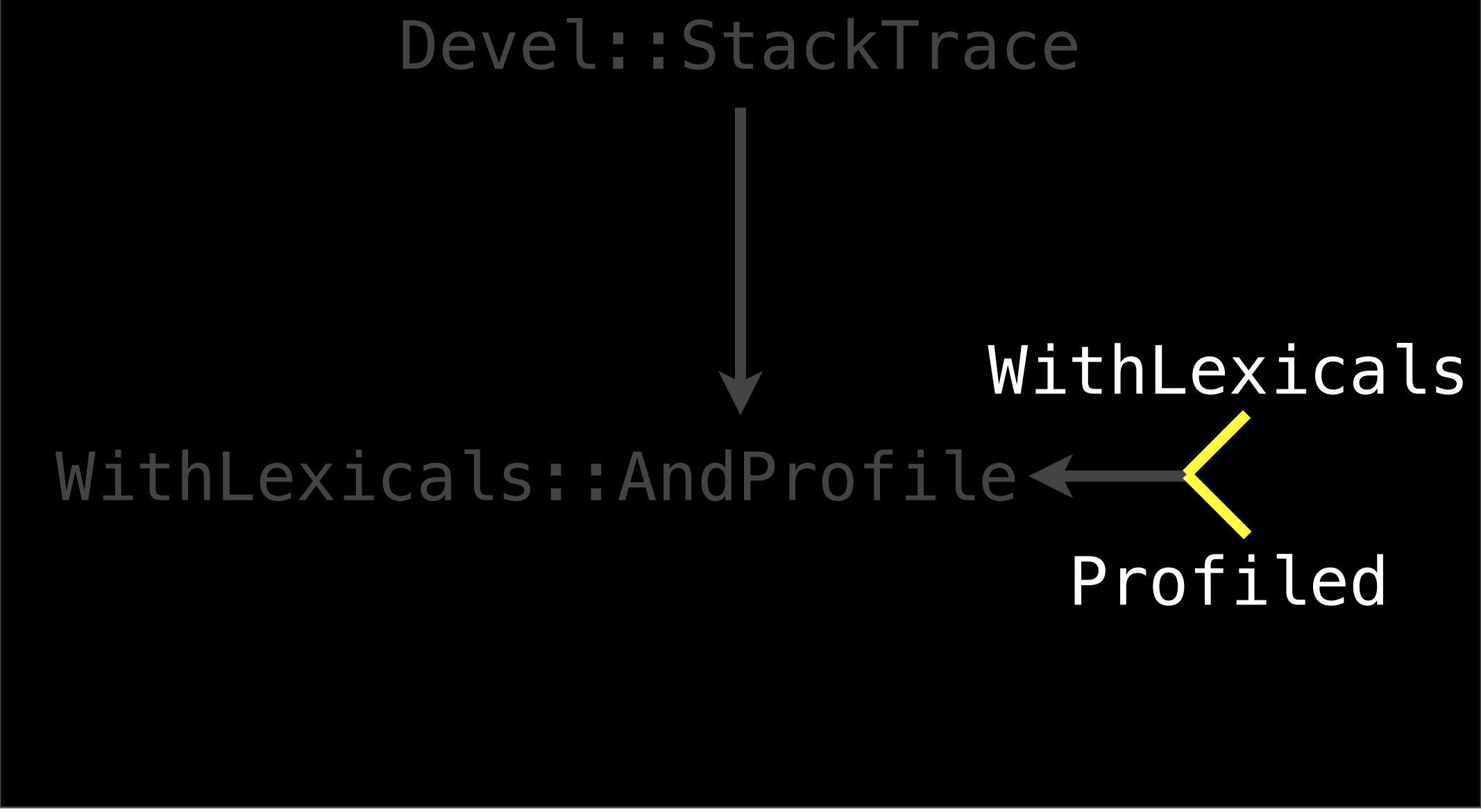
```
role Profiled

after _record_caller_data {
    $self->_record_extra_data
}

method _record_extra_data {
    # profiling
}
```

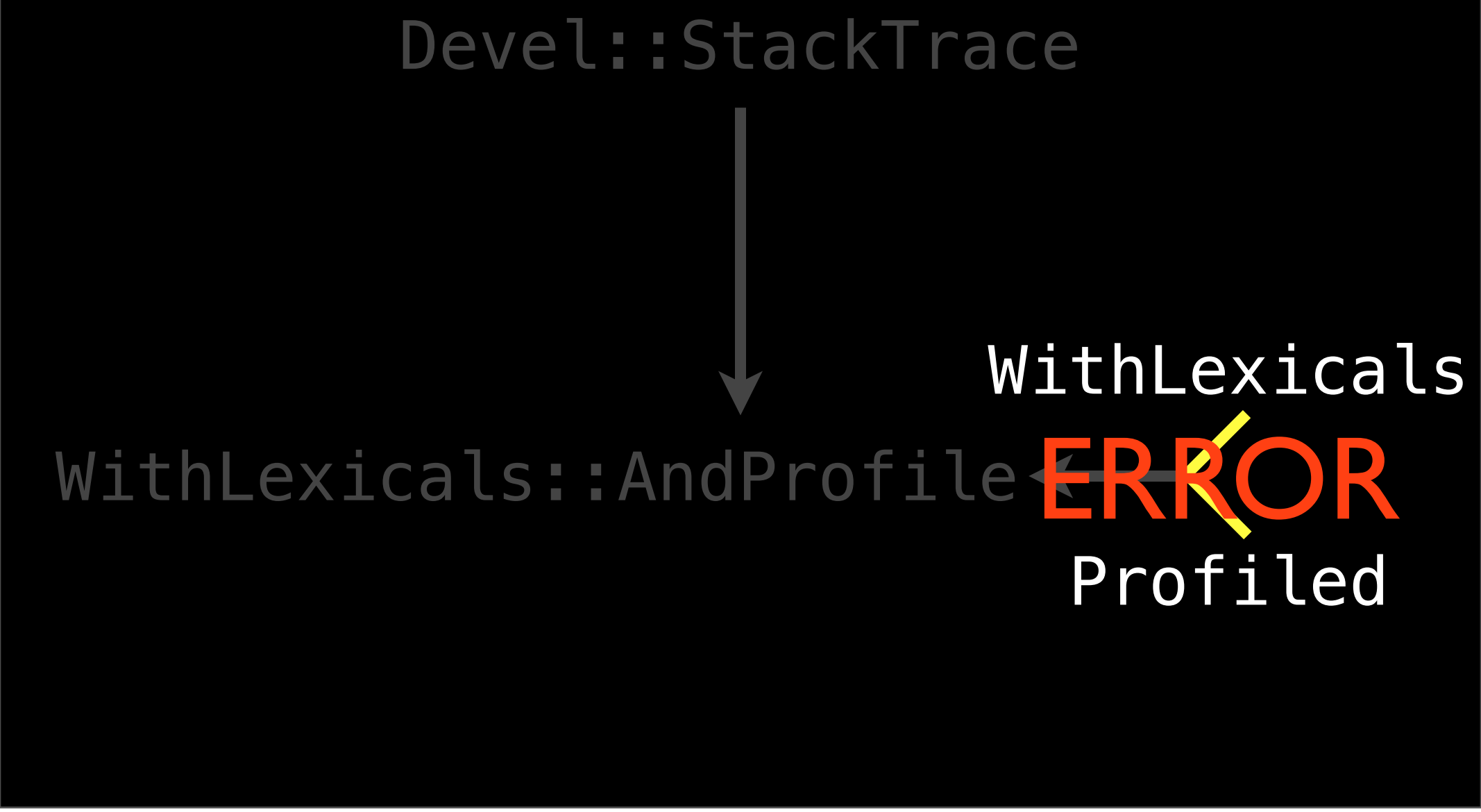

And again we have a Profiled role, same deal. And we have a method name collision on _record_extra_data.

Roles



When we combine these two roles, there is an ERROR.

Roles



When we combine these two roles, there is an ERROR.

Roles

```
Due to a method name conflict in roles
  'Profiled' and 'WithLexicals', the
  method '_record_extra_data' must be
      implemented or excluded by
      'WithLexicals::AndProfile'
```


The role system generates a method name conflict. Both of the two roles defined a method with the same name, so roles tell you about that problem. At compile time.

Mixins

```
$self->_record_extra_data($frame);
```

Profiled

WithLexicals

Remember, this is what happens in a mixin. The compile-time error roles give you is so much nicer than a bug at runtime which may not even make itself obvious. You might never notice that WithLexicals's method is not being called until after you ship to production.

Roles

```
$self->_record_caller_data($frame);
```

Profiled

WithLexicals

Devel::StackTrace

The original diamond problem is also not present for roles because of the flattening.

Polymorphism

- subclass polymorphism
- duck typing
- Go interfaces
- Java interfaces
- roles

Like I said earlier, polymorphism lets an object act in place of another as long as it is sufficiently similar. I'll talk about all these methods to enable polymorphism.

Subclass Polymorphism

- Liskov substitutition principle

Subclass polymorphism is familiar to all object-oriented programmers. The formal name for it is the Liskov substitution principle. It means that any code that operates on a particular class will handle any subclasses of that class.

Subclass Polymorphism

```
my $trace = Devel::StackTrace->new;

print trace_to_html($trace);
```


Say we create a Devel::StackTrace object and pass it to a trace_to_html function which renders it nicely. trace_to_html knows how to handle Devel::StackTrace so everything is fine.

Subclass Polymorphism

```
my $trace =
  Devel::StackTrace::Profiled->new;

print trace_to_html($trace);
```


Now we're passing a subclass of Devel::StackTrace but it should still just work. Devel::StackTrace::Profiled fulfills Devel::StackTrace's API completely, it just provides more methods for inspecting the timings of each frame.

Subclass Polymorphism

```
my $trace = Devel::Traceback->new;

print trace_to_html($trace);
```


Say someone else implements a completely different module Devel::Traceback module. It does not use Devel::StackTrace, perhaps because it's written in XS, or the code is cleaner, whatever the reason.

Subclass Polymorphism

```
my $trace = Devel::Traceback->new;

print trace_to_html($trace);
```

Devel::StackTrace

This won't work though because trace_to_html only accepts Devel::StackTrace objects, not Devel::Traceback objects. And for whatever reason – and there are potentially very many – we don't want Devel::Traceback to inherit from Devel::StackTrace.

Subclass Polymorphism

Inheritance for polymorphism is a *lot* to demand of an API. Turns out, again, that inheritance is a lose.

A subclass must play nicely with the superclass's state. So they share the object's data structure. This means that, for example, you can't use the same attribute names as any of your superclasses.

Your subclass inherits all of the methods that its superclasses has, even if it doesn't want them.

Your subclass must provide the same API or it will break in confusing ways.

Inheritance itself is not very conducive for encapsulation. Subclasses are affected by a refactoring of a superclass even if the superclass's public API stays the same. If a superclass adds a method then you must ensure that none of the subclasses already have that method, or if they do that it makes sense to override it.

Subclass Polymorphism

- state

Inheritance for polymorphism is a *lot* to demand of an API. Turns out, again, that inheritance is a lose.

A subclass must play nicely with the superclass's state. So they share the object's data structure. This means that, for example, you can't use the same attribute names as any of your superclasses.

Your subclass inherits all of the methods that its superclasses has, even if it doesn't want them.

Your subclass must provide the same API or it will break in confusing ways.

Inheritance itself is not very conducive for encapsulation. Subclasses are affected by a refactoring of a superclass even if the superclass's public API stays the same. If a superclass adds a method then you must ensure that none of the subclasses already have that method, or if they do that it makes sense to override it.

Subclass Polymorphism

- state
- methods

Inheritance for polymorphism is a *lot* to demand of an API. Turns out, again, that inheritance is a lose.

A subclass must play nicely with the superclass's state. So they share the object's data structure. This means that, for example, you can't use the same attribute names as any of your superclasses.

Your subclass inherits all of the methods that its superclasses has, even if it doesn't want them.

Your subclass must provide the same API or it will break in confusing ways.

Inheritance itself is not very conducive for encapsulation. Subclasses are affected by a refactoring of a superclass even if the superclass's public API stays the same. If a superclass adds a method then you must ensure that none of the subclasses already have that method, or if they do that it makes sense to override it.

Subclass Polymorphism

- state
- methods
- same API

Inheritance for polymorphism is a *lot* to demand of an API. Turns out, again, that inheritance is a lose.

A subclass must play nicely with the superclass's state. So they share the object's data structure. This means that, for example, you can't use the same attribute names as any of your superclasses.

Your subclass inherits all of the methods that its superclasses has, even if it doesn't want them.

Your subclass must provide the same API or it will break in confusing ways.

Inheritance itself is not very conducive for encapsulation. Subclasses are affected by a refactoring of a superclass even if the superclass's public API stays the same. If a superclass adds a method then you must ensure that none of the subclasses already have that method, or if they do that it makes sense to override it.

Subclass Polymorphism

- state
- methods
- same API
- breaks encapsulation

Inheritance for polymorphism is a *lot* to demand of an API. Turns out, again, that inheritance is a lose.

A subclass must play nicely with the superclass's state. So they share the object's data structure. This means that, for example, you can't use the same attribute names as any of your superclasses.

Your subclass inherits all of the methods that its superclasses has, even if it doesn't want them.

Your subclass must provide the same API or it will break in confusing ways.

Inheritance itself is not very conducive for encapsulation. Subclasses are affected by a refactoring of a superclass even if the superclass's public API stays the same. If a superclass adds a method then you must ensure that none of the subclasses already have that method, or if they do that it makes sense to override it.

Subclass Polymorphism

- state
- methods
- same API
- breaks encapsulation
- increases coupling

Inheritance for polymorphism is a *lot* to demand of an API. Turns out, again, that inheritance is a lose.

A subclass must play nicely with the superclass's state. So they share the object's data structure. This means that, for example, you can't use the same attribute names as any of your superclasses.

Your subclass inherits all of the methods that its superclasses has, even if it doesn't want them.

Your subclass must provide the same API or it will break in confusing ways.

Inheritance itself is not very conducive for encapsulation. Subclasses are affected by a refactoring of a superclass even if the superclass's public API stays the same. If a superclass adds a method then you must ensure that none of the subclasses already have that method, or if they do that it makes sense to override it.

Duck Typing

"when I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck."

-- James Whitcomb Riley

Duck typing is pretty popular in modern dynamic languages like Python, Ruby, and Perl. This kind of quote, which is apparently a common quote in the American South.

Duck Typing

when I see an object that has a `walk` method, a `swim` method, and a `quack` method, I call that object a duck.

Here it is recontextualized to describe duck typing.

Duck Typing

```
if $object->can('walk')
&& $object->can('swim')
&& $object->can('quack') {
    feed($object, 'bread');
}
```


So what this code does is it checks that the object has walk, swim and duck methods before treating it like a duck. It doesn't have to have yellow feathers or an orange bill, but it's close enough to a duck. $object doesn't have to inherit from the Duck class.

Duck Typing

```
sub trace_to_html {
   unless
       $trace->can('frame_function')
   && $trace->can('frame_file')
   && $trace->can('frame_line') {
       die "Not stack tracey enough"
   }

       ...
}
```


In our example of stack traces, we want to define a common enough API that all stack trace classes can implement. As long as your stack trace defines the three methods we need, then we can use it to generate HTML.

Duck Typing

```
my $trace = Devel::StackTrace->new;

print trace_to_html($trace);
```

So this works.

Duck Typing

```
my $trace =
  Devel::StackTrace::Profiled->new;

print trace_to_html($trace);
```

Using a subclass works too of course.

Duck Typing

```
my $trace = Devel::Traceback->new;

print trace_to_html($trace);
```

And using an entirely different module works. All that matters is that they provide the same three methods.

Duck Typing

So the problems with duck typing are that it only checks method names. It can't check whether the method takes certain kinds of arguments, or what its return value will be. Duck typing can't check that the method will do anything useful or expected. I'll talk more about this later.

Also duck typing is very ad hoc. It's not actually a language-level feature. The only language-level feature needed is being able to ask an object whether it has a certain method. Everything else is built on top in a non-introspectable way.

It's also kind of hard to maintain. If you want to add a new method to the duck type list you have to update every place you perform that duck type.

Duck Typing

- Method names

So the problems with duck typing are that it only checks method names. It can't check whether the method takes certain kinds of arguments, or what its return value will be. Duck typing can't check that the method will do anything useful or expected. I'll talk more about this later.

Also duck typing is very ad hoc. It's not actually a language-level feature. The only language-level feature needed is being able to ask an object whether it has a certain method. Everything else is built on top in a non-introspectable way.

It's also kind of hard to maintain. If you want to add a new method to the duck type list you have to update every place you perform that duck type.

Duck Typing

- Method names
- Very ad-hoc

So the problems with duck typing are that it only checks method names. It can't check whether the method takes certain kinds of arguments, or what its return value will be. Duck typing can't check that the method will do anything useful or expected. I'll talk more about this later.

Also duck typing is very ad hoc. It's not actually a language-level feature. The only language-level feature needed is being able to ask an object whether it has a certain method. Everything else is built on top in a non-introspectable way.

It's also kind of hard to maintain. If you want to add a new method to the duck type list you have to update every place you perform that duck type.

Duck Typing

- Method names
- Very ad-hoc
- Maintenance burden

So the problems with duck typing are that it only checks method names. It can't check whether the method takes certain kinds of arguments, or what its return value will be. Duck typing can't check that the method will do anything useful or expected. I'll talk more about this later.

Also duck typing is very ad hoc. It's not actually a language-level feature. The only language-level feature needed is being able to ask an object whether it has a certain method. Everything else is built on top in a non-introspectable way.

It's also kind of hard to maintain. If you want to add a new method to the duck type list you have to update every place you perform that duck type.

Duck Typing

```
sub trace_to_html {
   unless
      $trace->can('frame_function')
   && $trace->can('frame_file')
   && $trace->can('frame_line') {
      die "Not stack tracey enough"
   }

      . . .
}
```


So here we want to add a new method.

Duck Typing

```
sub trace_to_html {
  unless
      $trace->can('frame_function')
  && $trace->can('frame_file')
  && $trace->can('frame_line')
  && $trace->can('frame_count') {
      die "Not stack tracey enough"
  }
  ...
}
```


Every time we check whether something is a stack trace, we have to add this additional check.

Duck Typing

```
sub trace_to_html {
  unless
      $trace->can('frame_function')
  && $trace->can('frame_file')
  && $trace->can('frame_line')
  && $trace->can('frame_count') {
      die "Not stack tracey enough"
  }
  ...
}
```


Also this is not really introspectable for something like Eclipse. It's just a random check in the middle of a method. An IDE wouldn't want to inspect this because it could easily get too complicated to determine except at runtime. So we want more structure than duck typing provides.

Go Interfaces

Google Go's interfaces are like duck typing but better. An interface is a named collection of method signatures, so it's an actual entity in your programs instead of the design pattern of duck typing. But it's very similar to duck typing. There are two key differences.

One is that interfaces have names. A duck type is just a list of method names.

Also, interfaces can also check arguments and return values. Duck types generally check only that methods of the given names exist.

Go Interfaces

- *named*

Google Go's interfaces are like duck typing but better. An interface is a named collection of method signatures, so it's an actual entity in your programs instead of the design pattern of duck typing. But it's very similar to duck typing. There are two key differences.

One is that interfaces have names. A duck type is just a list of method names.

Also, interfaces can also check arguments and return values. Duck types generally check only that methods of the given names exist.

Go Interfaces

- *named*
- method signature

Google Go's interfaces are like duck typing but better. An interface is a named collection of method signatures, so it's an actual entity in your programs instead of the design pattern of duck typing. But it's very similar to duck typing. There are two key differences.

One is that interfaces have names. A duck type is just a list of method names.

Also, interfaces can also check arguments and return values. Duck types generally check only that methods of the given names exist.

Go Interfaces

```
if !$trace->does('StackTrace') {
    die
}
```


Because interfaces are named, you can check a single thing - does the object implement the interface - rather than checking every single method. Now when we want to add that frame_count method to this duck type, we only have to do it in a single place, which is in the StackTrace interface. I like it when I don't have to repeat myself.

Go Interfaces

Another thing interfaces do is check the entire method signature. Not only the method's name, but we can enforce the types of the arguments and return values. This is more than I've seen from most duck typing. So now we can confirm that, say, frame_count returns an integer. The more of the API we can check, the better the type check is because there will be fewer false positives. Which is kind of the whole point of type checking right?

Go Interfaces

- Name

Another thing interfaces do is check the entire method signature. Not only the method's name, but we can enforce the types of the arguments and return values. This is more than I've seen from most duck typing. So now we can confirm that, say, frame_count returns an integer. The more of the API we can check, the better the type check is because there will be fewer false positives. Which is kind of the whole point of type checking right?

Go Interfaces

- Name
- Return value

Another thing interfaces do is check the entire method signature. Not only the method's name, but we can enforce the types of the arguments and return values. This is more than I've seen from most duck typing. So now we can confirm that, say, frame_count returns an integer. The more of the API we can check, the better the type check is because there will be fewer false positives. Which is kind of the whole point of type checking right?

Go Interfaces

- Name
- Return value
- Arguments

Another thing interfaces do is check the entire method signature. Not only the method's name, but we can enforce the types of the arguments and return values. This is more than I've seen from most duck typing. So now we can confirm that, say, frame_count returns an integer. The more of the API we can check, the better the type check is because there will be fewer false positives. Which is kind of the whole point of type checking right?

Go Interfaces

A surprising thing about Go interfaces is that a class that fulfills an interface automatically "does" that interface. The class does not have to declare that it does that interface.

This is nice because, like duck typing, it improves decoupling between the interface and the class. This is handy when the class is written without knowledge of the interface, but someone else wants to use the class with functions that require that interface.

But it's also a source of problems because it means interfaces cannot declare semantics. This is kind of a subtle point but I will try to explain it.

Go Interfaces

- not declared -- automatic

A surprising thing about Go interfaces is that a class that fulfills an interface automatically "does" that interface. The class does not have to declare that it does that interface.

This is nice because, like duck typing, it improves decoupling between the interface and the class. This is handy when the class is written without knowledge of the interface, but someone else wants to use the class with functions that require that interface.

But it's also a source of problems because it means interfaces cannot declare semantics. This is kind of a subtle point but I will try to explain it.

Go Interfaces

- not declared -- automatic
- decoupling

A surprising thing about Go interfaces is that a class that fulfills an interface automatically "does" that interface. The class does not have to declare that it does that interface.

This is nice because, like duck typing, it improves decoupling between the interface and the class. This is handy when the class is written without knowledge of the interface, but someone else wants to use the class with functions that require that interface.

But it's also a source of problems because it means interfaces cannot declare semantics. This is kind of a subtle point but I will try to explain it.

Go Interfaces

- not declared -- automatic
- decoupling
- declaring semantics

A surprising thing about Go interfaces is that a class that fulfills an interface automatically "does" that interface. The class does not have to declare that it does that interface.

This is nice because, like duck typing, it improves decoupling between the interface and the class. This is handy when the class is written without knowledge of the interface, but someone else wants to use the class with functions that require that interface.

But it's also a source of problems because it means interfaces cannot declare semantics. This is kind of a subtle point but I will try to explain it.

Go Interfaces

```
interface NonblockingReader {
    string read();
}
```


Say we want an interface for classes that implement a nonblocking read. Something that reads data but does not wait for it if there is no data yet, returning the empty string or null or whatever. So classes which fulfill this interface must have a method named "read" which takes no arguments and returns a string.

Go Interfaces

```
class Filesystem {
    string read(string file) { ... }
}
```


We have a filesystem class. It has a read method, but it takes an argument so it does not fulfill the NonblockingReader interface. A simple duck type for the "read" method would not have caught this, so we're already ahead of the game.

Go Interfaces

~~NonblockingReader~~

```
class Filesystem {
    string read(string file) { ... }
}
```


We have a filesystem class. It has a read method, but it takes an argument so it does not fulfill the NonblockingReader interface. A simple duck type for the "read" method would not have caught this, so we're already ahead of the game.

Go Interfaces

```
class NonblockingSocket {
    string read() { ... }
}
```


We have a nonblocking socket class. It fulfills the interface entirely. It has a method named read that takes no arguments and returns a string. Great! So this class does the NonblockingReader interface and it didn't even need to declare that.

Go Interfaces

NonblockingReader

```
class NonblockingSocket {
    string read() { ... }
}
```


We have a nonblocking socket class. It fulfills the interface entirely. It has a method named read that takes no arguments and returns a string. Great! So this class does the NonblockingReader interface and it didn't even need to declare that.

Go Interfaces

```
class BlockingSocket {
    string read() { ... }
}
```


Now we have a *blocking* socket class. It too fulfills the interface. It has a read method that takes no arguments and returns a string. But it will block so it does not *really* fulfill the interface.

But according to the system, it DOES do the interface, because it has that read method! Oh no!

Go Interfaces

NonblockingReader

```
class BlockingSocket {
    string read() { ... }
}
```


Now we have a *blocking* socket class. It too fulfills the interface. It has a read method that takes no arguments and returns a string. But it will block so it does not *really* fulfill the interface.

But according to the system, it DOES do the interface, because it has that read method! Oh no!

Go Interfaces

So because interfaces are implicit, they can be a little dangerous. They may lie about whether an object fulfills an interface or not.

But they're also cool! It's an interesting new idea. It may show up in other languages if it turns out to work particularly well in Go. And from what I hear it does work well.

It's like duck typing's younger but more refined brother.

Go Interfaces

- dangerous

So because interfaces are implicit, they can be a little dangerous. They may lie about whether an object fulfills an interface or not.

But they're also cool! It's an interesting new idea. It may show up in other languages if it turns out to work particularly well in Go. And from what I hear it does work well.

It's like duck typing's younger but more refined brother.

Go Interfaces

- dangerous
- but cool!

So because interfaces are implicit, they can be a little dangerous. They may lie about whether an object fulfills an interface or not.

But they're also cool! It's an interesting new idea. It may show up in other languages if it turns out to work particularly well in Go. And from what I hear it does work well.

It's like duck typing's younger but more refined brother.

Go Interfaces

- dangerous
- but cool!
- duck typing codified

So because interfaces are implicit, they can be a little dangerous. They may lie about whether an object fulfills an interface or not.

But they're also cool! It's an interesting new idea. It may show up in other languages if it turns out to work particularly well in Go. And from what I hear it does work well.

It's like duck typing's younger but more refined brother.

Java Interfaces

Java interfaces are like Go interfaces. They're much older though.

Each interface has a name.

Each interface checks the full method signature of each method, not just the method's name.

And each interface must be explicitly declared. This is better because it avoids the problem of declaring semantics, the blocking reader versus nonblocking reader.

This is worse because it requires that the class author knows every single interface that the class implements, even interfaces that are written after the class is published!

Java Interfaces

- named

Java interfaces are like Go interfaces. They're much older though.

Each interface has a name.

Each interface checks the full method signature of each method, not just the method's name.

And each interface must be explicitly declared. This is better because it avoids the problem of declaring semantics, the blocking reader versus nonblocking reader.

This is worse because it requires that the class author knows every single interface that the class implements, even interfaces that are written after the class is published!

Java Interfaces

- named
- method signature

Java interfaces are like Go interfaces. They're much older though.

Each interface has a name.

Each interface checks the full method signature of each method, not just the method's name.

And each interface must be explicitly declared. This is better because it avoids the problem of declaring semantics, the blocking reader versus nonblocking reader.

This is worse because it requires that the class author knows every single interface that the class implements, even interfaces that are written after the class is published!

Java Interfaces

- named
- method signature
- explicitly declared

Java interfaces are like Go interfaces. They're much older though.

Each interface has a name.

Each interface checks the full method signature of each method, not just the method's name.

And each interface must be explicitly declared. This is better because it avoids the problem of declaring semantics, the blocking reader versus nonblocking reader.

This is worse because it requires that the class author knows every single interface that the class implements, even interfaces that are written after the class is published!

Java Interfaces

- named
- method signature
- explicitly declared
- *omniscience a plus*

Java interfaces are like Go interfaces. They're much older though.

Each interface has a name.

Each interface checks the full method signature of each method, not just the method's name.

And each interface must be explicitly declared. This is better because it avoids the problem of declaring semantics, the blocking reader versus nonblocking reader.

This is worse because it requires that the class author knows every single interface that the class implements, even interfaces that are written after the class is published!

Java Interfaces

```
class BlockingSocket
implements NonblockingReader {
    string read() { ... }
}
```


Now this is obviously a lie. If the programmer lies then nothing is safe. But you could certainly yell at him when you discover this during a code review.

Roles

Roles are like interfaces too.

Each role has a name, so you can ask whether a class performs a role rather than a list of method names.

By default roles can only require methods by name.

But if you use the MooseX::Role::Parameterized extension, you can require anything that you can write code for. For example that the class provides some method with two "w"s in the name.

The big advantage to roles is that they can provide a default implementation for methods. That's one huge problem with Java interfaces. Everyone who wants to fulfill a Java interface has to actually write the code in their class. There's no code reuse with interfaces. But with roles, where you get that default implementation, it's all about reuse.

Roles

- named

Roles are like interfaces too.

Each role has a name, so you can ask whether a class performs a role rather than a list of method names.

By default roles can only require methods by name.

But if you use the MooseX::Role::Parameterized extension, you can require anything that you can write code for. For example that the class provides some method with two "w"s in the name.

The big advantage to roles is that they can provide a default implementation for methods. That's one huge problem with Java interfaces. Everyone who wants to fulfill a Java interface has to actually write the code in their class. There's no code reuse with interfaces. But with roles, where you get that default implementation, it's all about reuse.

Roles

- named
- method names

Roles are like interfaces too.

Each role has a name, so you can ask whether a class performs a role rather than a list of method names.

By default roles can only require methods by name.

But if you use the MooseX::Role::Parameterized extension, you can require anything that you can write code for. For example that the class provides some method with two "w"s in the name.

The big advantage to roles is that they can provide a default implementation for methods. That's one huge problem with Java interfaces. Everyone who wants to fulfill a Java interface has to actually write the code in their class. There's no code reuse with interfaces. But with roles, where you get that default implementation, it's all about reuse.

Roles

- named
- method names
- extension for method signatures and more

Roles are like interfaces too.

Each role has a name, so you can ask whether a class performs a role rather than a list of method names.

By default roles can only require methods by name.

But if you use the MooseX::Role::Parameterized extension, you can require anything that you can write code for. For example that the class provides some method with two "w"s in the name.

The big advantage to roles is that they can provide a default implementation for methods. That's one huge problem with Java interfaces. Everyone who wants to fulfill a Java interface has to actually write the code in their class. There's no code reuse with interfaces. But with roles, where you get that default implementation, it's all about reuse.

Roles

- named
- method names
- extension for method signatures and more

ww

Roles are like interfaces too.

Each role has a name, so you can ask whether a class performs a role rather than a list of method names.

By default roles can only require methods by name.

But if you use the MooseX::Role::Parameterized extension, you can require anything that you can write code for. For example that the class provides some method with two "w"s in the name.

The big advantage to roles is that they can provide a default implementation for methods. That's one huge problem with Java interfaces. Everyone who wants to fulfill a Java interface has to actually write the code in their class. There's no code reuse with interfaces. But with roles, where you get that default implementation, it's all about reuse.

Roles

- named
- method names
- extension for method signatures and more
- default implementation

ww

Roles are like interfaces too.

Each role has a name, so you can ask whether a class performs a role rather than a list of method names.

By default roles can only require methods by name.

But if you use the MooseX::Role::Parameterized extension, you can require anything that you can write code for. For example that the class provides some method with two "w"s in the name.

The big advantage to roles is that they can provide a default implementation for methods. That's one huge problem with Java interfaces. Everyone who wants to fulfill a Java interface has to actually write the code in their class. There's no code reuse with interfaces. But with roles, where you get that default implementation, it's all about reuse.

Roles

- named
- method names
- extension for method signatures and more
- default implementation
- explicitly declared

ww

Roles are like interfaces too.

Each role has a name, so you can ask whether a class performs a role rather than a list of method names.

By default roles can only require methods by name.

But if you use the MooseX::Role::Parameterized extension, you can require anything that you can write code for. For example that the class provides some method with two "w"s in the name.

The big advantage to roles is that they can provide a default implementation for methods. That's one huge problem with Java interfaces. Everyone who wants to fulfill a Java interface has to actually write the code in their class. There's no code reuse with interfaces. But with roles, where you get that default implementation, it's all about reuse.

Roles

- named
- method names
- extension for method signatures and more
- default implementation
- explicitly declared
- composable

ww

Roles are like interfaces too.

Each role has a name, so you can ask whether a class performs a role rather than a list of method names.

By default roles can only require methods by name.

But if you use the MooseX::Role::Parameterized extension, you can require anything that you can write code for. For example that the class provides some method with two "w"s in the name.

The big advantage to roles is that they can provide a default implementation for methods. That's one huge problem with Java interfaces. Everyone who wants to fulfill a Java interface has to actually write the code in their class. There's no code reuse with interfaces. But with roles, where you get that default implementation, it's all about reuse.

Roles

```
role WithLexicals

after _record_caller_data {
    $self->_record_extra_data
}

method _record_extra_data {
    # extract lexicals
}
```


Providing default implementation is not actually new. We saw this earlier when I was showing off role conflict detection. This role is adding a new method - with a full implementation - to the classes that consume it.

Roles

```
role WithLexicals

after _record_caller_data {
    $self->_record_extra_data
}

method _record_extra_data {
    # extract lexicals
}
```


You can also see that roles can override methods from the consuming class, which may or may not be inherited.

Roles

```
if !$trace->does('StackTrace') {
    die
}
```


And of course because roles are named you can ask an object if it fulfills a particular role. So roles can definitely be used like interfaces to make polymorphism work well. And they can also be used like mixins to enable code reuse.

Roles

Or you could do both at the same time. Like here we are declaring an Equality role. It requires that each consumer have a "compare" method. And we give each consumer an "equals" method. It calls the compare method that we required.

If a particular consumer wants to optimize this equals method to not require the compare, then that class can define its own equals method with its own logic. It will still perform the Equality role.

Roles

```
role Equality
```


Or you could do both at the same time. Like here we are declaring an Equality role. It requires that each consumer have a "compare" method. And we give each consumer an "equals" method. It calls the compare method that we required.

If a particular consumer wants to optimize this equals method to not require the compare, then that class can define its own equals method with its own logic. It will still perform the Equality role.

Roles

```
role Equality

requires 'compare';
```


Or you could do both at the same time. Like here we are declaring an Equality role. It requires that each consumer have a "compare" method. And we give each consumer an "equals" method. It calls the compare method that we required.

If a particular consumer wants to optimize this equals method to not require the compare, then that class can define its own equals method with its own logic. It will still perform the Equality role.

Roles

```
role Equality

requires 'compare';

sub equals {
    $self->compare($other) == 0
}
```


Or you could do both at the same time. Like here we are declaring an Equality role. It requires that each consumer have a "compare" method. And we give each consumer an "equals" method. It calls the compare method that we required.

If a particular consumer wants to optimize this equals method to not require the compare, then that class can define its own equals method with its own logic. It will still perform the Equality role.

Learn More

There are a lot of role features I didn't talk about. None of them are really present in other systems.

Aliasing methods lets you rename methods when you consume a role.

Excluding methods lets you consume only some methods from a role.

Conflict resolution in roles. That's where I started with two roles providing two methods with the same name. Turns out you can resolve such conflicts quite nicely using aliasing and excluding. You don't need to rewrite one of the roles.

You can apply a role directly to an object, not just a class. That way only a single object does that role, not the entire class. Maybe that makes it pass more type checks, or maybe that gives it extra methods. Up to you. I think you can also apply mixins directly to an object.

There's parameterized roles which let you configure a role differently based on options that the

Learn More

- aliasing methods

There are a lot of role features I didn't talk about. None of them are really present in other systems.

Aliasing methods lets you rename methods when you consume a role.

Excluding methods lets you consume only some methods from a role.

Conflict resolution in roles. That's where I started with two roles providing two methods with the same name. Turns out you can resolve such conflicts quite nicely using aliasing and excluding. You don't need to rewrite one of the roles.

You can apply a role directly to an object, not just a class. That way only a single object does that role, not the entire class. Maybe that makes it pass more type checks, or maybe that gives it extra methods. Up to you. I think you can also apply mixins directly to an object.

There's parameterized roles which let you configure a role differently based on options that the

Learn More

- aliasing methods
- excluding methods

There are a lot of role features I didn't talk about. None of them are really present in other systems.

Aliasing methods lets you rename methods when you consume a role.

Excluding methods lets you consume only some methods from a role.

Conflict resolution in roles. That's where I started with two roles providing two methods with the same name. Turns out you can resolve such conflicts quite nicely using aliasing and excluding. You don't need to rewrite one of the roles.

You can apply a role directly to an object, not just a class. That way only a single object does that role, not the entire class. Maybe that makes it pass more type checks, or maybe that gives it extra methods. Up to you. I think you can also apply mixins directly to an object.

There's parameterized roles which let you configure a role differently based on options that the

Learn More

- aliasing methods
- excluding methods
- resolving conflicts

There are a lot of role features I didn't talk about. None of them are really present in other systems.

Aliasing methods lets you rename methods when you consume a role.

Excluding methods lets you consume only some methods from a role.

Conflict resolution in roles. That's where I started with two roles providing two methods with the same name. Turns out you can resolve such conflicts quite nicely using aliasing and excluding. You don't need to rewrite one of the roles.

You can apply a role directly to an object, not just a class. That way only a single object does that role, not the entire class. Maybe that makes it pass more type checks, or maybe that gives it extra methods. Up to you. I think you can also apply mixins directly to an object.

There's parameterized roles which let you configure a role differently based on options that the

Learn More

- aliasing methods
- excluding methods
- resolving conflicts
- applying to an object

There are a lot of role features I didn't talk about. None of them are really present in other systems.

Aliasing methods lets you rename methods when you consume a role.

Excluding methods lets you consume only some methods from a role.

Conflict resolution in roles. That's where I started with two roles providing two methods with the same name. Turns out you can resolve such conflicts quite nicely using aliasing and excluding. You don't need to rewrite one of the roles.

You can apply a role directly to an object, not just a class. That way only a single object does that role, not the entire class. Maybe that makes it pass more type checks, or maybe that gives it extra methods. Up to you. I think you can also apply mixins directly to an object.

There's parameterized roles which let you configure a role differently based on options that the

Learn More

- aliasing methods
- excluding methods
- resolving conflicts
- applying to an object
- parameterized roles

There are a lot of role features I didn't talk about. None of them are really present in other systems.

Aliasing methods lets you rename methods when you consume a role.

Excluding methods lets you consume only some methods from a role.

Conflict resolution in roles. That's where I started with two roles providing two methods with the same name. Turns out you can resolve such conflicts quite nicely using aliasing and excluding. You don't need to rewrite one of the roles.

You can apply a role directly to an object, not just a class. That way only a single object does that role, not the entire class. Maybe that makes it pass more type checks, or maybe that gives it extra methods. Up to you. I think you can also apply mixins directly to an object.

There's parameterized roles which let you configure a role differently based on options that the

Learn More

- aliasing methods
- excluding methods
- resolving conflicts
- applying to an object
- parameterized roles
- new design patterns

There are a lot of role features I didn't talk about. None of them are really present in other systems.

Aliasing methods lets you rename methods when you consume a role.

Excluding methods lets you consume only some methods from a role.

Conflict resolution in roles. That's where I started with two roles providing two methods with the same name. Turns out you can resolve such conflicts quite nicely using aliasing and excluding. You don't need to rewrite one of the roles.

You can apply a role directly to an object, not just a class. That way only a single object does that role, not the entire class. Maybe that makes it pass more type checks, or maybe that gives it extra methods. Up to you. I think you can also apply mixins directly to an object.

There's parameterized roles which let you configure a role differently based on options that the

Learn More

http://scg.unibe.ch/research/traits

http://tinyurl.com/traits-papers

http://search.cpan.org/perldoc?Moose::Manual::Roles

http://tinyurl.com/roles-manual

http://sartak.org/talks/osdc.tw-2010/nonhierarchical-oop/

http://tinyurl.com/nh-oop-talk

If you want to learn more about the origins of roles, their initial implementation and design using Smalltalk, and their formal specification, check out the first link. It's worth noting that roles depart from these original traits in that they are allowed to have attributes. That's kind of a long story, but it turns out it's a really useful feature to have, even if it makes some things harder.

If you want to learn more about the current best implementation of roles, which is in Perl's Moose object system, check out the second link.

The second link is more for human being users of roles, whereas the first link is for super hackers who are building roles into a language.

The third link is of course these slides.

Thank you!

Thank you!

- Thank you

Thank you!

- Thank you
- 謝謝

Thank you!

- Thank you
- 謝謝
- 有り難うございました