Object-oriented programming is about behavior, state, and identity. Behavior as in message passing, method calls. State as in attributes, properties. Identity as in two objects with the same values for their properties are distinct. Every object-oriented programming language provides these core principles. But then there’s the details. Ohh the details.
Every programming language provides a unique take on object-oriented programming. For example, some languages provide inheritance via classes, others via prototype. Some have “final” declarations, some have dynamic dispatch, some generic methods, etc. etc. It’s almost like you could pick any mix of features that make sense together and there’d be a language there. Some exist even where their list of features don’t really make sense together.
We could even treat languages as points on a plane, where position is determined by the set of OOP tradeoffs that language has made. Perl is here. Obviously. And then there’s a bunch of other languages that are more or less different from other languages. By the way, don’t take this chart too literally; it’s merely demonstrative.
Let's zoom in around Perl. All the usual suspects are here. None of these languages provide exactly the same OOP. For example Ruby lets you subclass builtin types like string and array. In Python you don’t have to explicitly “bless” a reference into an object but is otherwise pretty similar to Perl. But they’re all similar in that they’re dynamic, more-or-less-hash-based, weakly typed languages, so that’s why they’re in a quadrant together.
But you might be thinking... hey, there’s more than one way to do it. The way I write OOP Perl is different from the way you write OOP Perl. And Ingy writes Perl like no one else.
Hey, Spiffy may not have worked out, but don’t hate on Ingy!
You might be wondering where Moose is on this chart.
Moose as it exists out of the box is here. But there's something more going on. Moose is flexible and extensible in ways that these other systems are not. In fact, you can extend Moose to be like any of these other systems. And anywhere in between.
So we might render Moose as a **region** instead of a point. Moose can work like any of these other systems…. hang on a sec, this isn’t very Moosey.
That’s better. So, for example, MooseX::InsideOut lets you store properties outside of the object, especially useful when subclassing someone else’s code. And of course you can tweak that to be a point close to, but not exactly, standard inside-out objects. Perhaps you want to store properties by something other than object address, like some kind of application-specific identifier.
This is the crux of the talk: with Moose you can move to any point in this region that makes sense for your application. In other words (shedding the point/region metaphor), with Moose you can select exactly the set of OOP features you want for your application. All without leaving Perl for a different language. All while being able to robustly interoperate with code from CPAN and other developers. That's the true power of Moose. That's what keeps me writing Perl. Not “has”. 
Moose pulls this off by doing something so simple you probably wouldn’t believe it could work. Moose itself is implemented using instances, classes, methods, and attributes. The other systems mentioned, like standard Perl OOP, are implemented with things like hash tables, C functions, structs, pointers, glob assignment. This means you can’t subclass or use any other OOP technique to extend or modify their behavior; each is a static artifact: a singular, area-less point on the OOP chart. You get what you get and you don’t throw a fit.
package Reluctant;
use Moose::Role;

before new_object => sub {
    warn "Another one...? 😞";
    sleep 1;
};

Because Moose is OOP it’s very familiar to extend it. You do it in two parts: first, write the extension. The extension can usually be a role. It’s okay if you haven’t seen roles before, the example will still be clear. In this example I looked at Moose’s documentation or its code to figure out that new_object is the method we wanted to override. Then just use an “before” method modifier to whinge and drag my feet about it. So you get the standard behavior for classes that Moose provides, plus this one tiny incremental change.
Then we tell Moose we want to use this extension in our class. Done. Note that this is robust and modular: only Person and its subclasses will get this behavior, no one else. This is much better than alternatives like monkeypatching! Behind the scenes this is just saying "give me a subclass of the standard Moose class with the Reluctant role". Easy peasy. It's just object-oriented programming. You've subclassed a million classes by now. You got dis!
By the way there are literally millions of Moose extensions on CPAN. Each of these MooseX modules fills in and expands the region of OOP that Moose supports. So a lot of times you can skip that first step and just go to using these existing extensions.
Because it's all OOP, there are no black boxes in Moose. In other words, in Moose, the plumbing (the internals) and the porcelain (the user interface) are one and the same; there's nothing hidden beneath the surface. This is a huge benefit in so many ways because it means there's a unified model, and it's a model you're already familiar with. You can use porcelain to extend any part of Moose because it's all porcelain, and by the way, our porcelain is terrific.
Single parameters to new() must be a HASH ref at /Users/sartak/.perl/perls/perl-5.22.0/lib/site_perl/5.22.0/darwin-2level/Moose/Exception.pm line 37
Moose::Exception::_build_trace('Moose::Exception::SingleParamsToNewMustBeHashRef=HASH(0x7fd49af417a8)') called at reader Moose::Exception::trace (defined at /Users/sartak/.perl/perls/perl-5.22.0/lib/site_perl/5.22.0/darwin-2level/Moose/Exception.pm line 9) line 7
Moose::Exception::trace('Moose::Exception::SingleParamsToNewMustBeHashRef=HASH(0x7fd49af417a8)') called at /Users/sartak/.perl/perls/perl-5.22.0/lib/site_perl/5.22.0/darwin-2level/Moose/Exception.pm line 49
Moose::Exception::BUILD('Moose::Exception::SingleParamsToNewMustBeHashRef=HASH(0x7fd49af417a8)', 'HASH(0x7fd49af42f70)') called at /Users/sartak/.perl/perls/perl-5.22.0/lib/site_perl/5.22.0/darwin-2level/Class/MOP/Method.pm line 126

But sometimes this power backfires and you get a plumbing problem where you were expecting a porcelain problem. Like the real world, this is a crappy situation to be in. Sorry. 😔
A method for breathing fresh air in a room filled with toxic smoke

The recent rash of fires in high-rise hotels and deaths occasioned thereby has given rise to the need for a breathing device and method for supplying a hotel guest and/or fireman with fresh air until he can be rescued. The device and method of this invention provide for the insertion of a breathing tube through the water trap of a toilet to expose an open end thereof to fresh air from a vent pipe connected to a sewer line of the toilet, to enable the user to breathe fresh air through the tube.
And here’s where we get into the title of the talk. What’s this lifting business about. While yes, it is wonderful that there’s an extension that adds multi-methods to Moose. But Moose extensions really don’t need to be “about” object-oriented programming. There are no limits here. Your Moose extensions can be domain-specific, or “about” your application.
Imagine your application as a pink pill sitting at the top of a stack including Moose, Perl, and the OS. What I’m proposing is that you…
…lift Moose more into your application, so that Moose not something you merely sit on, it becomes an active participant in how your application is architected. Don’t reinvent half of Moose poorly.
The canonical example might be web forms. Your classes have attributes and perhaps you’re writing a web form to populate those attributes. Well, you can start by looking at the core Moose options for attributes.
Web Forms

has email => (  
  is => 'rw',
  required => 1,
);

Requiredness is easy enough. Just inspect the attribute’s “required” property and if so add the little asterisk to indicate that it’s required.
You might be wondering how to figure out if an attribute is required. Remember, Moose is implemented with object-oriented programming, so of course you use method calls to retrieve information. First, get the metaclass (which you might be somewhat familiar with doing thanks to __PACKAGE__->meta->make_immutable). This is the object that represents the class we’re defining. Then, get_attribute, which pulls out the object representing the email attribute we’re defining. Finally, call the is_required accessor which will return true or false.
Web Forms

```php
has email => (  
  is => 'rw',  
  required => 1,  
);
```

And you can hook that into your form validation code to display an error when the field is empty during submit.
Here we use a type constraint. The type constraint is “Email” which would hook into our form validation, set the input type so the browser can do its own validation, and on mobile UIs it displays a keyboard specific to email entry with an @ sign etc. And of course when this value is submitted to the server it will be type-checked again.
Moose has a documentation attribute that lets you associate a string with that attribute. It isn’t used by the core codebase but we can make use of it to display help text on the form.
That’s not bad but I see a couple problems with it. For one, the label “email”, lower case, strikes me as kind of sloppy. Second, a placeholder value, the greyed out text inside the form field, would make this look just a little nicer.
has email => (  
is          => 'rw',  
    placeholder => 'Email',
);

Found unknown argument 'placeholder' in the has declaration for 'email'

Unfortunately Moose does not ship with a “placeholder” option. Which is perfectly reasonable: placeholder is an HTML thing, and Moose is not specifically designed for HTML. But Moose is extensible: let's add one! We're enhancing Moose for our application.
As you remember, first we define an extension. Here is a perfectly complete extension. This role adds an attribute to the class that consumes it. No more no less.
Second, we tell Moose that we want to use the “Field” extension in the “email” attribute. This lets us use the “placeholder” attribute that we created in the Field role. That’s all! Of course the “placeholder” can use the full power of Moose: it has a type constraint so that passing something other than a string would throw a compile-time error. We could make placeholder required if that made sense for our app.
We access that placeholder value the exact same way as required, except we of course use the placeholder accessor instead. It's all OOP!
Second, we tell Moose that we want to use the “Field” extension in the “email” attribute. This lets us use the “placeholder” attribute that we created in the Field role. That’s all!
has label => (
    is => 'ro',
    isa => 'Str',
);
Then we provide a label for our email field, and use it in place of the attribute’s name. Much better!
Here's our full attribute. Not too bad, right? It's all declarative which is nice. Consider how you might implement this without Moose.
While I think traits => ['Field'] is probably fine as is, it’d be even nicer to clean it up. Luckily, thanks to the vibrant MooseX ecosystem, Moose has built up many tools to make this sort of thing automatic.
You can create a package, similar to something like MooseX::Singleton, that automatically applies the Field role to all the attributes in a class, as well as roles to any other object in the Moose ecosystem like your class or your methods. So it's about as sweet as you could ask for!
has email => (
    is => 'rw',
    required => 1,
    isa => 'Email',
    documentation =>
        "We won't send unsolicited email to you.",
    placeholder => 'Email',
    label => 'Email address',
    required_for_mobile => 0,
);
Of course recall that we’re not limited to just adding new options to attributes. We can extend Moose’s concept of class, attribute, role, method, instance, etc. Wrap methods. Add attributes. Mix and match all the different extension mechanisms and package them up into a single “use” statement. Don’t be limited by Moose’s default implementation, or even by MooseX. There’s so much more you can do! Explore the region. Push its edges outward!
Thank you for your attention!

Slides

twitter.com/sartak