Hi!
I'm Stefan

This is my first talk at FOSDEM, and I'm kinda nervous.

So let's start off with my most favourite hobby: flying.
As in airplanes.
As in piloting.

Here's a picture of me flying a glider.
I know, when you see this picture, your first thought is: „oh, nice pair of legs!“

I'd agree, but what I find even more interesting is the view outside.

Like this autumnish view of my home town which you cannot see because of the fog.
But in the background, you can see the Alps instead.
Which is one reason why I love flying.
It gives you a different perspective, making distances much smaller.

My next favourite occupation is programming.

And when I program be it at work or in my spare time, I do it in Perl.

When we say Perl, we usually mean Perl 5.
The version of Perl that's been with us for some 20 years and makes billions of any currency every year.

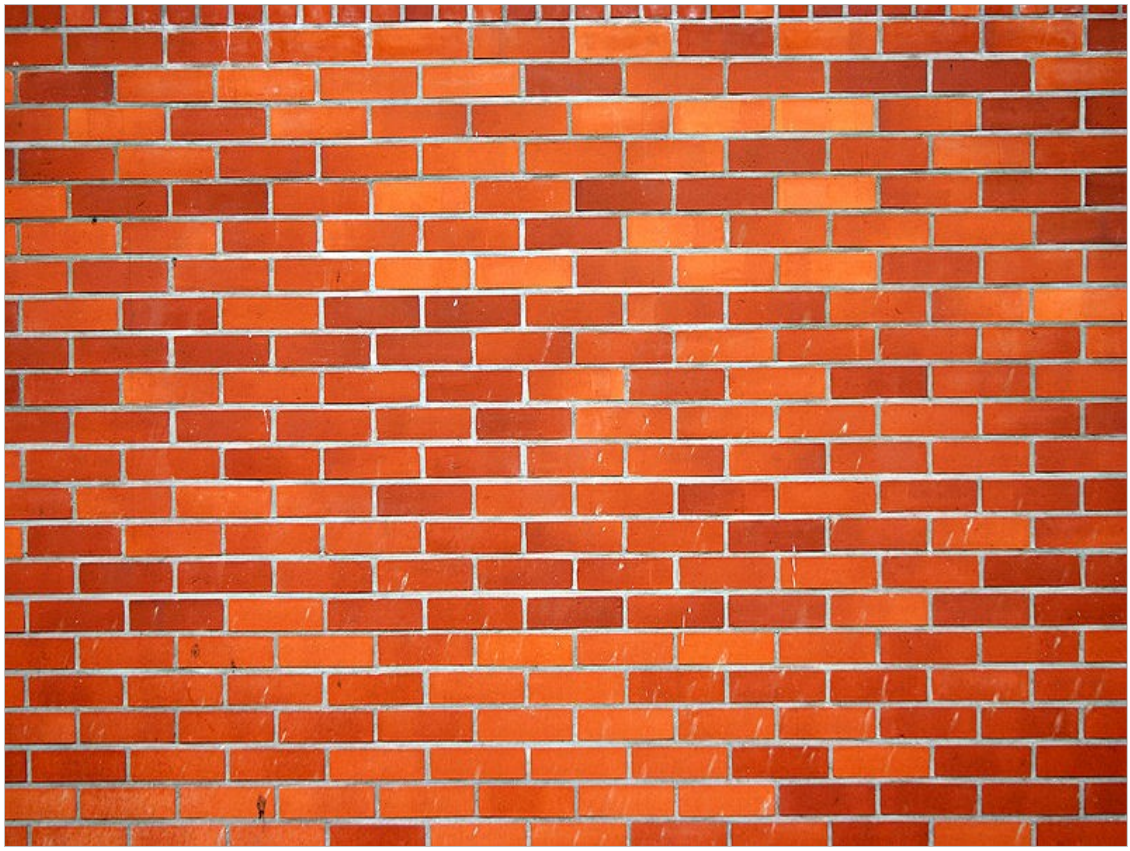
In about 2005, I first heard about a programming language called „Perl 6“.
I first though this was going to be a nice update to the Perl 5 I knew and loved. But all in all, I was quite satisfied with the Perl I knew.

By about 2007, I was hooked. I had seen some presentations and Perl 6 shaped up to be a language that would be easier to get into but also insanely powerful.

In the years after that, I've seen more talks about Perl 6 and read about it and was itching to try it out.

But whenever I thought about trying it out, I hit a wall.

And I don't mean this Wall!



More like a brick wall.
And it was always the same brick wall.
I know, because it still had my marks from previous tries.
The wall I'm talking about is missing modules.
At first it was database access.
Then template modules,
Web frameworks,
IMAP client,
all the things I'm used to as a Perl 5 programmer.

And not to mention the quarter of a million of LOC that I'm sitting on at work that we're maintaining.
We start new projects about twice a decade. And I'm actually trying to reduce that further.

So let me ask a quick question:
Who is sitting with me in the same boat?

Who is tempted to try Perl 6 but is hampered by missing modules or existing code bases?

Leapfrogging the bootstrap

It's not just Perl 6.
It's been 5 years since Python 3 was released.
And it's only now finally picking up steam.
The reason is exactly the same problem we face with Perl 6.
We've been waiting for Perl 6 for more than a decade. I absolutely do not want to wait another 5 years for modules to be ported. I want to leapfrog this bootstrap and go right into having fun.

My goal for today is, that after this talk, you will be able to go out and give Perl 6 a try. That missing modules or existing code bases no longer keep you from experiencing this great language.

How do I want to accomplish that?

use NativeCall;

In one of the talks I mentioned earlier, I learned about NativeCall.

NativeCall is Perl 6's way to access system libraries.

Think Perl 5's XS, but without the horror in your voice. The „ooh XS“.

```
use NativeCall;

sub some_function(int)
    is native('libsomething')
    { * }

some_function(42);
```

Instead, NativeCall is actually beautiful.
Have a look at this.
This is all pure Perl 6 code.
Just like that you can have Perl 6 load a library for you, and access a function this library provides.

When I saw this, it got me thinking.

libperl.so

The Perl 5 interpreter is also provided as a library.

If I can use NativeCall to load perl5 and use it's embedding interface, then in theory I should be able to build a bridge.

And literally, within an hour of having this idea, I finished my very first Perl 6 program. This program used NativeCall to load perl5 and print a “hello world” written in Perl 5.

This Hello World became a module called Inline::Perl5

Inline::Perl5

A couple of days later, I could use DBI, quickly followed by DBIx::Class.

And I was like “wow”, you're really onto something here!

This was a great start, but if I wanted to use Perl 6 in my projects, I'd have to ascend a much larger mountain.

We use the Catalyst web framework for pretty much all our projects.

So to be able to use Perl 6, I have to be able to do this in Catalyst based projects.

That's where we start looking at some code.

Catalyst creates a start script for you to run your application on a Perl web server.

Such a script may look like this:

```
#!/usr/bin/env perl

BEGIN {
    $ENV{CATALYST_SCRIPT_GEN} = 40;
}

use Catalyst::ScriptRunner;
Catalyst::ScriptRunner->run(
  'CiderWebmail',
  'Server'
);

1;
```

This is a fairly straight forward perl script.
It is basically just loading a module and calling a package method that does the real work.
Now how could we port this to Perl 6?

```
#!/usr/bin/env perl6

%*ENV<CATALYST_SCRIPT_GEN> = 40;

use Inline::Perl5;

my $p5 = Inline::Perl5.new;

$p5.run(q:heredoc/PERL5/);
   use lib qw(lib);
   use Catalyst::ScriptRunner;
   Catalyst::ScriptRunner->run(
      'CiderWebmail',
      'Server'
   );
PERL5
```

If you look at the first line, I just replaced perl by perl6.

Setting the environment variable just got translated to different syntax.

Then we load Inline::Perl5.

We create a new $p5 object and call its run method giving it a single string as argument.

```
use Inline::Perl5;

%*ENV<CATALYST_SCRIPT_GEN> = 40;

my $p5 = Inline::Perl5.new;

$p5.use('lib', 'lib');
$p5.use('Catalyst::ScriptRunner');
$p5.invoke(
    'Catalyst::ScriptRunner',
    'run',
    'CiderWebmail',
    'Server'
);
```

Inline::Perl5 has a 'use' method that does exactly the same as use in Perl5.

That's basically just sugar, so you don't have to use run just to load a module.

Then there's invoke which lets you call methods on packages or objects.

You just give it your package or object, the name of the method and additional arguments.

```
use Inline::Perl5;
my $p5 = Inline::Perl5.new;

$p5.use('Petal');
my $template = $p5.invoke(
    'Petal',
    'new',
    'bars.xhtml'
);
say $template.process({
    city => 'Linz',
    bars => <Chelsea Walker Bugs>,
});
```

Of course the method may have one or more return values and those get returned by invoke just like you'd expect.
This quick example uses Perl 5's Petal templating module.
We just create a new template object.
Then we feed it some data in the form of a hash containing strings and lists.
The function returns a string which we then print.

With this, one can already work quite nicely. But there's still room for improvement.
Let's have a look at an even simpler example.

```
use Inline::Perl5;
use Digest::MD5:from<Perl5>;

our &md5 := &Digest::MD5::md5;

say md5('foo');
```

Perl 6 is already designed to accomodate interoperation with other languages.

It allows for modules to plug into the 'use' mechanism.

With this, we can get rid of the $p5 object entirely and literally 'use' Perl 5 modules like we're used to.

What does not yet work is automatically importing symbols from these Perl 5 modules. So I do this manually in this example.

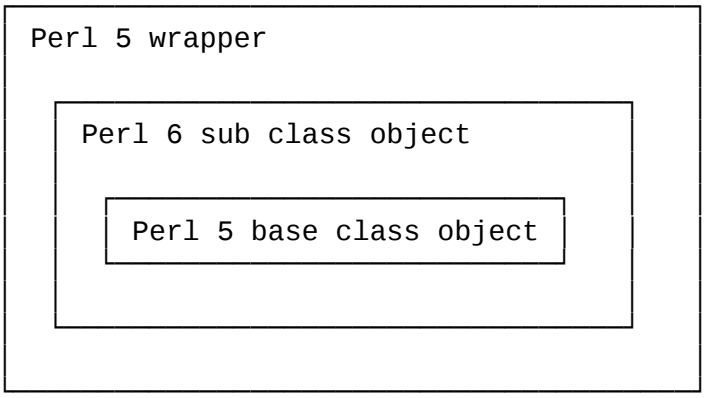
```
use Inline::Perl5;
use XML::XPath:from<Perl5>;

my $xp = XML::XPath.new(
    'xml-xpath.xml'
);
my $nodeset = $xp.find('//baz/@qux');

say $nodeset.get_node(1).getData;
```

When you 'use' Perl 5 modules this way, Inline::Perl5 automatically creates a corresponding Perl6 package for you, including wrappers for all functions and methods.
With this, you can just call a constructor as if the module was a real native Perl 6 class.
The wrappers still just call invoke like we've seen in earlier examples.
Again, this is a complete and working example.

What I've shown so far allows you to use a large number of CPAN modules.
However, some of them expect you to subclass.
In Catalyst you write model, view and controller classes that are subclasses of classes Catalyst provides.
Controllers for example are derived from Catalyst::Controller.

These inheritance based APIs usually expect you to implement some methods.
For Catalyst controllers those methods are the actions that handle web requests.

So how do we subclass a Perl 5 class in Perl 6?



Inline::Perl5 provides a Perl5Parent role.

By consuming this role in your class, you are telling Inline::Perl5 that your class should act like a subclass of the given Perl 5 class.

That means whenever a method on an object of this class is called, it is dispatched to your Perl 6 class.

If the Perl 6 class does not provide the method, the call is dispatched to the underlying Perl 5 object.

This works regardless of the caller being Perl 5 or Perl 6 code.

I call this inheritance by delegation.

Bot::BasicBot example

```
use Inline::Perl5;

my $p5 = Inline::Perl5.new;
$p5.use('Bot::BasicBot');

class P6Bot does
   Inline::Perl5::Perl5Parent[
      'Bot::BasicBot'
   ]
{ ... }
```

This inheritance is needed to use Bot::BasicBot.

If you'd like to implement an IRC bot, you can just subclass Bot::BasicBot.

To do this, we again load Inline::Perl5 and have our P6Bot class consume the Perl5Parent role with 'Bot::BasicBot' as parameter.

Bot::BasicBot example

```
method said(%statement) {
   self.reply(
      $%statement,
      "Hullo {%statement<who>}!"
   ) if %statement<body> eq 'Hi bot!';

   self.shutdown('leaving...')
      if %statement<body> eq 'quit';
}
```

Then we are free to implement methods like 'said' that gets called whenever somone says something on the channel.

Here is where we implement the bot's behaviour. We answer politely when we are greeted and we shut down when asked to.

Bot::BasicBot example

```
my $bot = P6Bot.new(
    perl5    => $p5,
    server   => "irc.freenode.org",
    port     => "6667",
    channels => ["#perl6"],
    nick     => "p6basicbot",
    name     => "Yet Another Bot",
);

$bot.run();
```

Starting up the bot is a matter of copy & paste from the documentation, modulo some syntax changes.

The main difference is that we have an additional parameter called „perl5“ in which we pass the Perl 5 interpreter along.

If this was pure advertising I'd stop here. But there's a small gotcha I discovered when preparing this example.

All Perl 6 objects automatically inherit from the „Mu“ class which provides a method called „say“.

Bot::BasicBot also has a method with the same name and this is used by „reply“.

As Perl5Parent delegates all unknown methods to the underlying Perl 5 class, this does not happen for „say“.

Bot::BasicBot gotcha

```
method say(*@args) {
    $.parent.perl5.invoke(
        'Bot::BasicBot',
        $.parent.ptr,
        'say',
        self,
        |@args
    );
}
```

We have to be a bit more explicit here and implement this delegation ourselves.

The „say“ method you see here contains copy&pasted delegation code.

Here we find our old friend, the „invoke“ method.

HTML::Parser example

```
use Inline::Perl5;
my $p5 = Inline::Perl5.new;
$p5.use('HTML::Parser');

class PrettyPrinter does
Inline::Perl5::Perl5Parent[
   'HTML::Parser'
]
{ ... }
```

HTML::Parser looks rather similiar to what I've just shown. In fact, this is again some copy&paste with changed names.

Again, we subclass using Perl5Parent.

If you've used HTML::Parser before, you might throw in that HTML::Parser does not require you to subclass.

You could also pass it some code referencs to handle parse events. Inline::Perl5 would indeed allow you to do that. But it supports this by creating Perl 5 objects that overload the call operator.

Unfortunately HTML::Parser's XS implementation is rather inflexible and accepts only real, genuine code references.

HTML::Parser example

```
method indent() {
   return '    ' x $!level;
}

method start($tag, %attrs, @attrs, $full) {
   say self.indent
      ~ '<'
      ~ $tag
      ~ @attrs.map({ qq/ $_="{%attrs{$_}}"/ }).join
      ~ '>';
   $!level++;
}

method text($content, *@args) {
   say $content.indent(*).trim.indent($!level * 4);
}
```

So instead, we subclass and implement the methods.
If you ask me, this results in cleaner code anyway.
Especially with Perl 6's method signatures.

HTML::Parser example

```
PrettyPrinter.new(
    perl5 => $p5,
).parse_file('html-parser.html');
```

Using our PrettyPrinter class is as straight forward, as you can imagine.

Again, I would love to be able to stop here, but there's a tiny issue, you should be aware of.

HTML::Parser gotcha

```
method end($tag, *@args) {
    $!level--;
    say self.indent ~ "</$tag>";
}

method sink() { self } # magic
```

It turns out that „end“ is some magic method in Perl 6!

It's provided by class List.

You might wonder how List is connected to any of this.

Well, lists of one element and scalars are intentionally confused in Perl 6 as moritz explained to me.

```
class XStats::Controller::Root
   does Inline::Perl5::Perl5Parent[
      'Xstats::Controller::Root'
   ];

method index($c) {
   $c.stash({
       template  => 'index.zpt',
       uri_graph =>
          $c.uri_for('graph'),
   });
}
```

Equiped with the knowledge of the previous examples, one may be led to believe that implementing a Catalyst controller is a matter of using Perl5Parent again.

Alas, it's not that simple.

Catalyst components are Moose classes.

Catalyst uses Class::MOP's introspection capabilities to find the actions a controller supports and how they should be mapped to URLs.

The latter is done by querying the attributes of the action methods.

Perl 6 does not have subroutine attributes and Class::MOP does only deal with classes it created itself.

To be able to use a Perl 6 object as Catalyst controller, I wrote a replacement for Class::MOP that I could use as meta class for my controllers.

```
class Perl6::MOP;

use CatalystX::Perl6::Component::Perl5Attributes;

has $.parent;

class P6Method {
   has $.method;

   method can($name) {
       return self.^can($name);
   }

   method name {
       return $.method.gist;
   }

   method attributes {
       return $.method.attributes;
   }

   method body {
       return $.method;
   }
}

method name {
   return $.parent.name;
}

method get_nearest_methods_with_attributes {
   my $class = ::($.parent.name);

   return
       |$class.^methods\
           .grep({$_.^does(::Perl5Attributes)})\
           .map({P6Method.new(method => $_)}),
       |$.parent.get_nearest_methods_with_attributes;
}

method can($name) {
   return self.^can($name);
}

method linearized_isa {
   return $.parent.linearized_isa;
}
```

Luckily Perl 6 already supports meta objects and introspection so this ended up being surprisingly little code.

If you can't read it, don't worry.

I really only have to map the Class::MOP API that Catalyst uses to the Perl 6 introspection API ending up with a couple of one liner methods.

```
package CatalystX::Perl6::Component;

use Moose::Role;

sub COMPONENT {
   my ($class, $app, $args) = @_;
   my $self = $class->new($app, $args);
   my $p6 = $Perl6::ObjectCreator->create($class, $self);
   bless $self, "Perl6::Object::$class"; # Explodes if we bless $p6 here!
   return $p6;
}

sub init_metaclass {
   my ($class) = @_;

   my $perl6_class = "Perl6::Object::$class";
   {
       no strict 'refs';
       @{ $perl6_class . "::ISA" } = qw(Perl6::Object);
   }

   Class::MOP::store_metaclass_by_name(
       $perl6_class,
       $Perl6::ObjectCreator->create('Perl6::MOP', $class->meta)
   );
}

1;
```

I then wrote a Moose Role that hooks up the Perl6::MOP and tricks Catalyst into using our Perl 6 subclass for the controller.

With introspection being taken care of, we can have a look at the other question:

How do we manage information about how to map URLs to actions?

While Perl 6 does not have subroutine attributes it has traits.

Traits allow you to easily attach meta data to all kinds of objects including methods.

```
our role Perl5Attributes[Routine $r] {
   has $.attributes = [];
}

multi trait_mod:<is>(
   Routine $declarand!,
   :@p5attrs!
) is export {
   $declarand
      does Perl5Attributes[$declarand]
      unless
         $declarand.does(Perl5Attributes);
   $declarand.attributes.push(@p5attrs);
}
```

These traits just apply a role to the method and stores the meta data in a new attribute that can be queried by Perl6::MOP.

This is pretty much everything you need to write Catalyst components in Perl 6.

However there's one annoying thing left.

Catalyst automatically loads modules from the Model, View and Controller namespaces.

Obviously these are Perl 5 modules which would then have to load the corresponding Perl 6 modules containing the real code.

```
package XStats::Controller::Root;
use Moose;
use namespace::autoclean;

BEGIN {
   extends 'Catalyst::Controller';
   with 'CatalystX::Perl6::Component';
}

__PACKAGE__->config(namespace => '');
__PACKAGE__->init_metaclass;
__PACKAGE__->meta->make_immutable;
```

So you'd end up having duplicate class trees with Perl 5 shims like this one and Perl 6 classes containing the meat.

Luckily, there's a solution for that, too.

```
use v6-inline;

use CatalystX::Perl6::Component::Perl5Attributes;

also does Inline::Perl5::Perl5Parent[
   'Xstats::Controller::Root'
];
also does CatalystX::Perl6::Component;

method index($c) is Path is Args[0] {
   $c.stash({
       template  => 'index.zpt',
       uri_graph => $c.uri_for('graph'),
   });
}

method end($c) is ActionClass["RenderView"] {
}
```

- Inline::Perl5 automatically creates a package called v6-inline.
- By just saying "use v6-inline;" you can hand over the rest of the file to Perl 6 for processing.
- This allows you to have the Perl 6 code in the same file als the Perl 5 shim.
- My most favourite feature of this is, that as you can see if you look closely, there is no static 1; at the end of the module.
- Or at least, it's not visible. Because v6-inline is implemented as a source filter. And it replaces all your shiny Perl 6 code with a static 1;

Live Demo!

Limitations

- Source Filters
- Devel::*
- Perl6::*
- (introspection)

I've shown you a lot of things that do work with Inline::Perl5.

An obvious question is: what does not work?

Without making any claim of completeness, there are some obvious candidates:

* Any kind of source filter would be highly surprised by finding Perl 6 code.
* Low level modules in the Devel:: namespace.
* Probably most modules in the Perl6 namespace which are by their very nature obsolete.
* As you've seen, interaction with Class::MOP does not work out of the box, but can at least in some cases be implemented easily.

```
{
   use v5;
   if( $^O eq 'MSWin32' ) {
      say $0;
   }

   {
      use 6.0.0;
      multi sub foo(*     ) {
         ...
      }
   }
}
# $*DISTRO is a Perl 6 "special variable"
say "Back to Perl 6 on a $*DISTRO.name box.";
```

If you don't actually need XS support, there is an alternative to Inline::Perl5.

It's called v5 and in short is an implementation of Perl 5, written in Perl 6.

It parses Perl 5 code using a Perl 6 grammar and compiles it to an AST just like it's done with Perl 6 code.

This means that it can benefit from JIT compilation and threading support thereby in theory allowing it to surpass Perl 5's performance.

So if you can get away with it, v5 should be your first place to go.

However, like I said, it does not support XS and never will.

It also does not cope too well with anything based on Devel::Declare or influencing the parsing in any way.

Then, there's another alternative. Maybe, you want to use a Python module instead of Perl 5?

```
use v6;
use Inline::Python;

my $py = Inline::Python.new();

$py.run(q:heredoc/PYTHON/);
    import PyQt4
    import PyQt4.QtCore
    import PyQt4.QtGui
    PYTHON
```

There's actually some Python module that I'd love to use in Perl.

It's the bindings for the Qt C++ library.

If you're a KDE user, you're probably quite familiar with it. It is used to render all the user interfaces.

There have been several attempts to provide Perl bindings, but they usually and up being incomplete and unmaintained.

Python bindings on the other hand are maintained by the Qt developers themselves.

Inline::Python to the rescue!

If you've stayed with me so far, this should look quite natural to you by now.

Very straight forward Python code execution.

```
class MessageBox
  does Inline::Python::PythonParent[
    'PyQt4.QtGui',
    'QWidget'
  ]
  { ... }
```

For inheritance, we have a PythonParent role which by the way also allows for multiple inheritance.

```
method setup() {
    self.resize(250, 150);
    self.setWindowTitle(
        'message box'
    );
    self.show();
    self.center();
}
```

This setup method is a straight conversion of an example from the PyQt4 tutorial.

```
method closeEvent($event) {
    my $reply = $py.invoke(
        'PyQt4.QtGui',
        'QMessageBox',
        'question',
        $.parent,
        'Message',
        'Are you sure to quit?',
        YES,
        DEFAULT,
    );
    $reply == YES
        ?? $event.accept
        !! $event.ignore;
}
```

In the handler for the window close event, I open a dialog box nagging you with the obligatory „Are you sure“ question.

```
my $app = $py.call(
  'PyQt4.QtGui',
  'QApplication',
  ['pyqt4.t'],
);

my $message_box = MessageBox.new;
$message_box.setup;
$app.exec_;
```

Running the application is again a straight forward port directly from the tutorial.

If you wonder why the exec method has a trailing underscore, that's simply because „exec“ is a keyword in Python. And unlike the Perl 6 developers, Python's designers focused more on making the implementation of the language parser simpler instead of focusing on the user.

TRY IT!

http://github.com/niner/Inline-Perl5
http://niner.name/talks

Now that we're at the end, let me re-visit my goal. I've shown you ways to use Perl 5 and even Python modules in Perl 6 code. I've shown you a technique for a piecemeal upgrade of an existing codebase.

There should be nothing in your way if you want to have fun with Perl 6. And I sincerely hope, you'll have as much fun, as I had with all this so far.

Thank you!