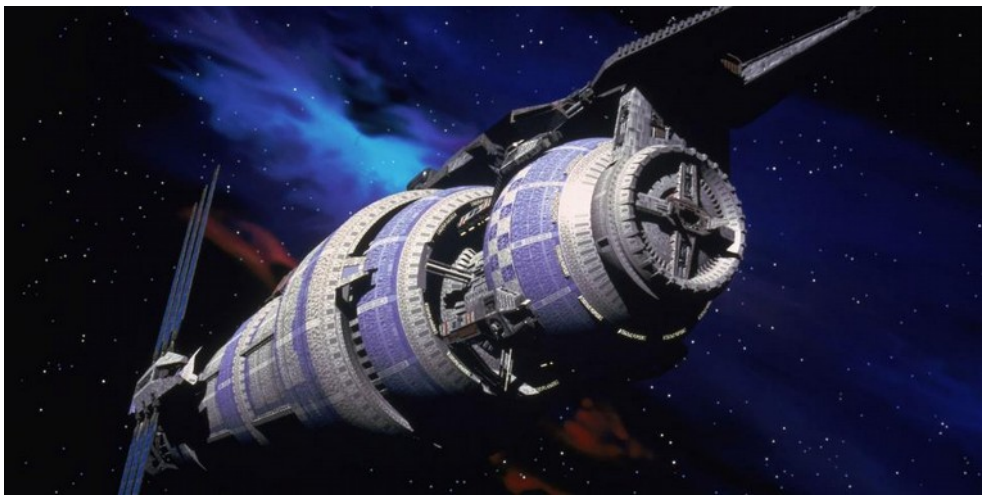
I remember the birth of this millenium.
I am quite surprised though that I do.
Because back then, I did everything I could, so I wouldn't remember.
In other words: I was drunk.
I was shit faced.
I was so drunk, I danced half naked on our village's market place.
Fortunately for me, where I'm from, that is somewhat acceptable behavior for a youth my age back then.
It was 16 years ago after all.

In the same year, when I was trying to get through puberty without too many scars, Perl 6 was announced.
It took me some five years though, till I first heard about it.
In 2005, safely arrived in adulthood and a bit calmer, I wasn't that excited about Perl 6.
I though, ok, this will probably gonna be a nice update to the Perl 5 I know and love.
They would probably remove some of the cruft and add some niceties.
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 both easier to get into and insanely powerful at the same time.
I also realized, that this really was a completely new language.

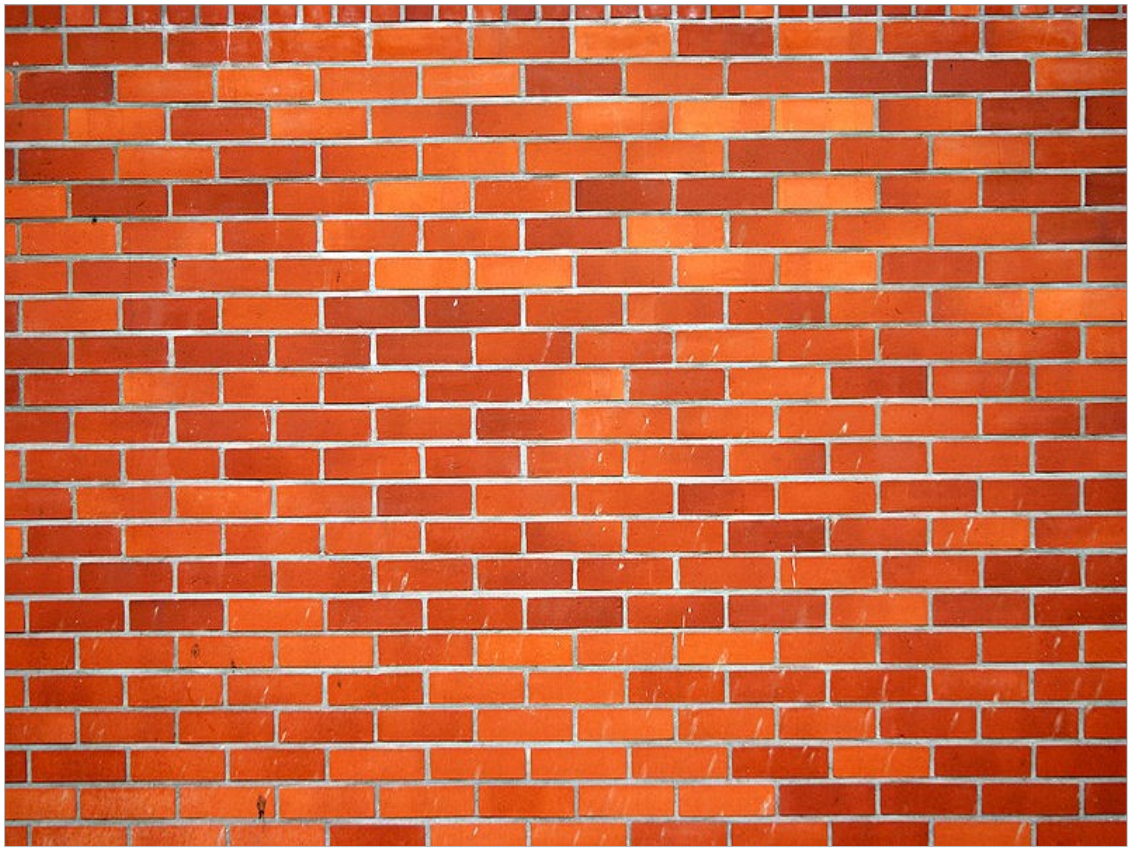


I've seen Damian Conway's keynote where he told us that half his CPAN modules are just trying to bring features to Perl 5 that Perl 6 already has.
Guess what? Shocking as this may come, but most of us, are not actually Damian Conway!
And looking just at at February's new CPAN modules, we find Business-Payment-SwissESR, Pcore-GeoIP or Dancer-SerachApp.
That's loads and loads of special purpose modules that have nothing to do at all with improving the language but just happen to be written in it.

So Damian's message, that half of CPAN is obsolete anyway and the other half would surely be rewritten soonish was maybe a bit too optimistic.
And that's exactly what I've experienced.

In the past couple of years, 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 still 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 having 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.

It's like I can only have either a cool new language, or all the power of CPAN.
There's a rift there and it not only separates our code bases, but our communities also.
You're either a Perl 5 programmer with all your real world connections, or one of the Perl 6 crazies with their high floating visions.
For a time it looked even more dire.
There was some hostility between the two sides.
Thankfully, we since have overcome this, and found some state of peaceful coexistence.
There's even been some exchange of ideas.
But the sides are still quite clearly kept apart.

Building Bridges

One outcome of this exchange of ideas has been sane and powerful (meta-object) models in both Perl 5 and Perl 6.
Just imagine what we could still gain by an even more free flow of ideas.
When instead of being content with living side by side and occasionally talking to each other, we would actually live together.
I for one would love to live in such a future.

Well, we will not get there over night.
But I can show you one important step in this direction today.
This step will remove the most divisive decision you had to make previously: the question whether you want to use the powerful new language features or the massive amounts of pre-existing code.

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.
Or if you consider yourself a 6er, that you will no longer restrict yourself to what's possible with the currently existing module ecosystem.
Because as Perl programmers, we are usually not content with having a cake. Looking at a lot of bellies, we really want to eat it, too :)

So 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 all the horror that comes to mind.
Without the nightmares of C macros.

```
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.
You do not even need a C compiler for this.

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 run a “hello world” written in Perl 5.

As Hello::World would have been kind of an awkward name, I called the resulting module 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 our 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 method returns a string which we then print.

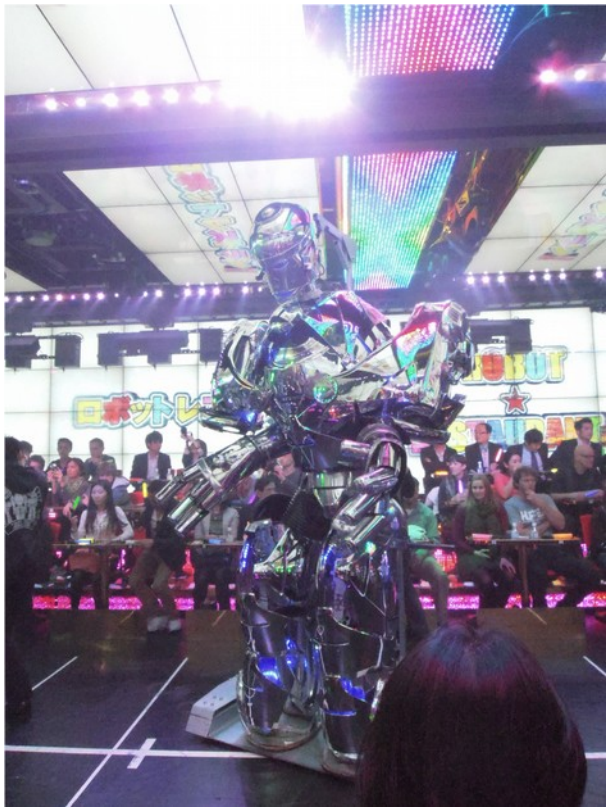
As we can see, it's certainly possible to use Perl 5 modules in Perl 6. But would you really want to program like that? It could certainly look much nicer. It could. It can. And it does :)

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

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.
And even more: you don't even have to load Inline::Perl5. Perl 6 mandates Perl 5 support and Rakudo knows that Inline::Perl5 can provide it. So as long as you have Inline::Perl5 installed, you don't have to worry about that anymore.
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.

```
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;
```

What if such a wrapper is actually wrapping a constructor?
Well, they do what constructors do. They return objects.
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.



Now for some real Science Fiction.
What I've shown so far allows you to use a large number of CPAN modules.
However, some of them expect you to subclass.
Bot::BasicBot or HTML::Parser would be examples.

These inheritance based APIs usually expect you to implement some methods.

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

Bot::BasicBot example

```
use Bot::BasicBot:from<Perl5>;

class P6Bot is Bot::BasicBot {
  ...
}
```

It turns out, you'd do just the same as with any plain ordinary Perl 6 class.

And indeed, that's exactly what happens. Inline::Perl5 creates a shadow class in Perl 6 for the package you load from Perl 5. You can just use this shadow class as base for your own class. The base class will still delegate all calls to methods that you did not override in your subclass to the Perl 5 class.

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 'bot quit';

}
```

When you do override methods, they really are exactly what you would expect to find in any Perl 6 class. We have methods and signatures and can call other methods, probably provided by our base class like „reply“ or „shutdown“ for our IRC bot.

Bot::BasicBot example

```
my $bot = P6Bot.new(
    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.

If you have seen this talk a year ago, I used to mention some gotcha at this point. It's fixed and gone now. This really is everything you have to do.

HTML::Parser example

```
use HTML::Parser:from<Perl5>;

class PrettyPrinter is HTML::Parser {
    has $!level = 0;
    ...
    method indent() {
        return '    ' x $!level;
    }
}
```

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

Again, we subclass the generated wrapper class.

HTML::Parser example

```
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
   );
}
```

And we implement the methods.
I very much like the result. Especially with Perl 6's method signatures.

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

You could also pass it some code references to handle parse events. And by code references, I really mean that. HTML::Parser's XS implementation is rather inflexible and would for example not accept callable objects.

HTML::Parser functional

```
my $level = 0;

sub indent { '    ' x $level }

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

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

sub end($tag, *@args) { $level--; say indent() ~ "</$tag>"; }

HTML::Parser.new(
   api_version => 3,
   start_h => [&start, "tagname, attr, attrseq"],
   text_h  => [&text, "dtext"],
   end_h   => [&end, "tagname"],
).parse_file('html-parser.html');
```

HTML::Parser is not the only module with this issue, so Inline::Perl5 nowadays really gives you that. If you pass a Callable to Perl 5 code, this code will get a real genuine code reference.

Thus, even HTML::Parser will be happy and again I could really just do what its documentation told me to.

HTML::Parser example

```
PrettyPrinter.new
    .parse_file('html-parser.html');
```

Just for completeness, using our PrettyPrinter class is as straight forward, as you can imagine.

With the knowledge we have now, we can use most of CPAN already. So is it time to call it a success and go have some coffee?

I like to pick my goals a bit higher than that.

I'm not just looking for a treasure, I'm looking for the holy grail. I'm going for the big ones. To me, that means Catalyst. That means DBIx::Class.

I want to do this:

DBIx::Class

```
use DBIC::Demo:from<Perl5>;
my $schema = DBIC::Demo.connect('dbi:Pg:dbname=nine');

my $nin = $schema.resultset('Artist').create({
   name => 'Nine Inch Nails',
});
$nin.create_related('cds', {
   title => 'The Downward Spiral',
});

my $cd = $schema.resultset('Cd').search_rs.first;

say $cd.full_title;
```

If I am starting to bore you already, I'd consider this a good thing actually. Because I'm showing you the same trick for the third time now. Use from Perl5.

So why am I making this fuzz? Why did I talk about a grail?

Well, while using a DBIx::Class schema is rather straight forward, wouldn't it be really cool, if we could also write one in Perl 6?

Of course, since such schema classes are created by calling a couple of methods on the base class, this is not exactly a challenge either.

But I'm lazy and I hope, you are, too. So I want to have my schema generated by dbic_dumper and then extend it. But dbic_dumper creates Perl 5 code. Surely you cannot just add some Perl 6 code, can you?


CHALLENGE
ACCEPTED

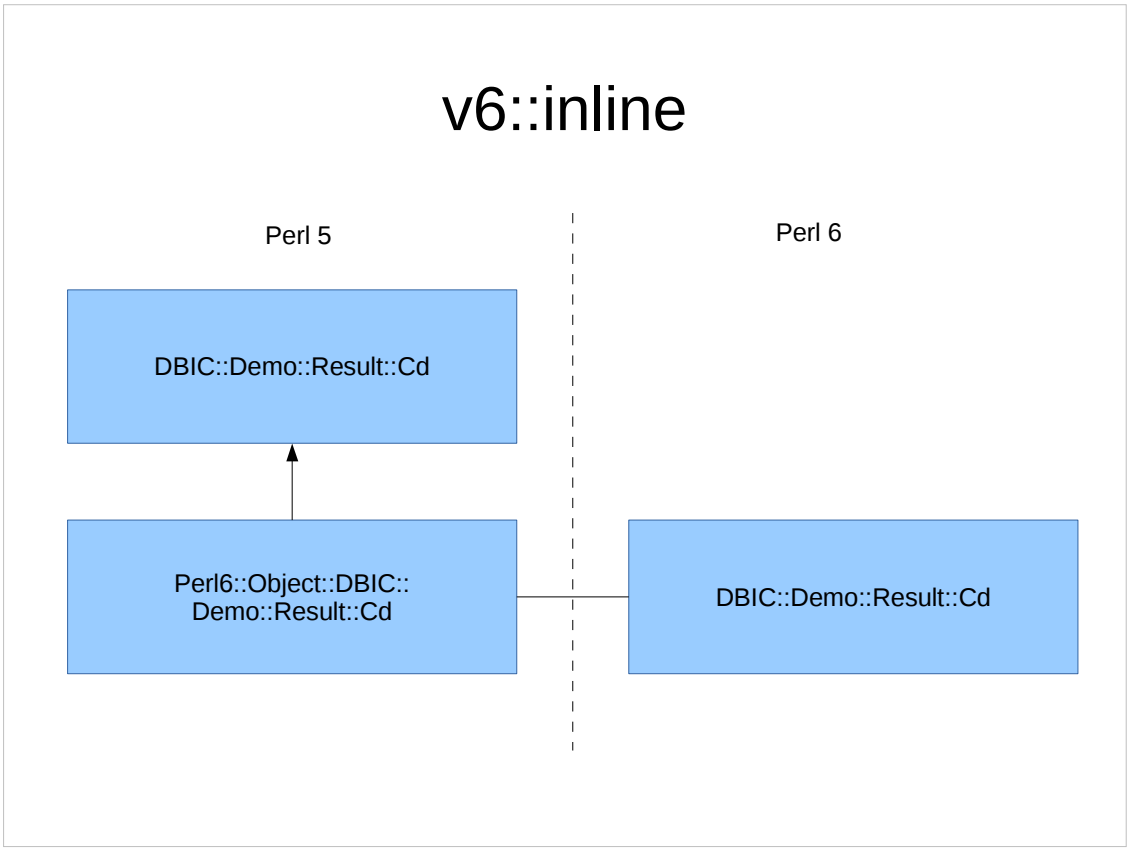
```
use utf8;
package DBIC::Demo::Result::Cd;
...
__PACKAGE__->belongs_to(...);

# Created by DBIx::Class::Schema::Loader v0.07043 @ 2015-09-01
13:18:52
# DO NOT MODIFY THIS OR ANYTHING ABOVE!
md5sum:7kPADFeDHuCnkZ2VDAhaZg

use v6::inline constructors =>
    [qw(new inflate_result)];

method full_title() {
   "{$.artist.name} - {$.title}"
}
```

The top half of this slide is just some plain normal DBIx::Class schema code, generated by dbic_dumper including the comments and checksums.

Instead of just adding methods in Perl 5 however, we use the special package called “v6::inline”. By using this package, we declare that the rest of the file is in fact written in Perl 6.

And indeed, here we find the “full_title” method, we've seen earlier. This method uses the generated accessors, DBIC provides.

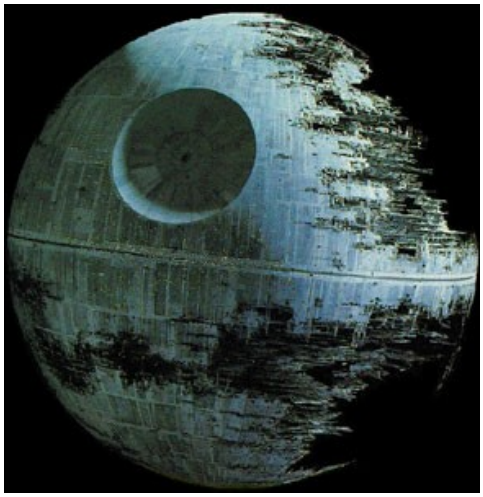
Now the bit to explain here is the rest of the v6::inline line.



What happens is, that Inline::Perl5 automatically creates a subclass for us in Perl 5. This subclass is used as the base for a Perl 6 class of the same name as the package we are in. This is the class where the Perl 6 methods will be added.

Now when we create an object, we actually want this to be a Perl 6 object, of the Perl 6 DBIC::Demo::Result::Cd class. And to achieve that, we have to wrap the constructors and redirect them to the Perl 6 class.

Now since constructors can have arbitrary names and Inline::Perl5 cannot guess them, we simply have to tell it, what they are called. In the case of a DBIx::Class ResultSource, this would be “new” and “inflate_result”.



Are you convinced yet, that we can use the awesome power of the fully armed and operational battle station that is CPAN in Perl 6?

```
use Inline::Perl5;

my @p5 = Inline::Perl5.new xx 10;

my @promises = @p5.kv.map: -> $i, $p5 {
   start {
       $p5.run("
           use 5.10.0;
           sleep $i;
           say 'hello world: $i';
           $i
       ");
   }
};

say "awaiting";
say [+] await @promises;
```

Well, I'm still not done yet!

If you are like me, at some point you will ask the question, how Perl 6' threading features work with Perl 5.

The answer is: somewhat.

You can create multiple independent Perl 5 interpreters and use them in different threads.

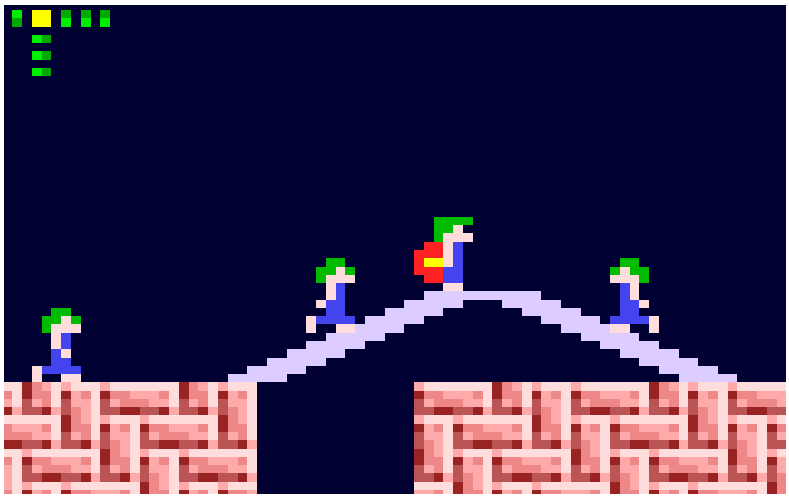
Those interpreters will not see each other and will start out in a blank state. You can however plug them together on the Perl 6 side and push values around.

In fact, replace Perl 6 here by yet another Perl 5 interpreter and you have exactly what ithreads does.

As far as stability goes, it's better than I actually expected but not yet something I'd really recommend. This is really bleeding edge.

It is fun to play around with in any case!

Inline::Perl6



So far we've only tried accessing Perl 5 code from Perl 6.

For me that's just not enough I want to bring both languages together, and that means both directions.

In other words, we need a bridge back.

This bridge is called Inline::Perl6.

I told you earlier, that Perl 5 is embeddable so I could use it as basis for Inline::Perl5. Well Rakudo, or to be more precise MoarVM isn't. But I did it anyway.

I just hope that we can get a real embedding interface one of these days, so I can get rid of a lot of copied code.

Inline::Perl6

```
use common::sense;
use Inline::Perl6;
BEGIN { Inline::Perl6::initialize };

v6::run('say <Hello from Perl6>');
v6::call('say', 'Hello from Perl6');

v6::run('
    class GLOBAL::MyStr {
        has $.value;
        method say() { $.value.say };
    }');
v6::invoke(
    'MyStr',
    'new',
    v6::named value => 'Hello from Perl6'
)->say;
```

On this slide, you can see 3 ways to have Perl 6 say “Hello World” for you. You always start by loading Inline::Perl6 and initializing.

Then we use one of the functions that are all prefixed by v6::. run is for evaluating some code. There's a call again, to call any named function. And there's invoke to call a method.

There are a couple of questions this code may raise. First, if you look in the middle, my MyStr class has a GLOBAL:: added to its name. That's because in Perl 6 really everything, including classes is lexically scoped by default. Without the GLOBAL, the class would only be known inside the EVAL.

Then when invoking the “new” method, I need to pass named arguments. Perl 5 doesn't really know the concept. So I need to mark them as v6::named. Otherwise Perl 6 would complain about unexpected positional arguments.

This v6::named is also the reason why we need to initialize Inline::Perl6 in a BEGIN block. Otherwise we'd need to add parentheses around v6::named's arguments and that would no longer look as nice, wouldn't it?

Inline::Perl6

```
use common::sense;
use Inline::Perl6;
BEGIN { Inline::Perl6::initialize; }
use DBIC::Demo;

my $schema = DBIC::Demo->connect('dbi:Pg:dbname=nine');

my $nin = $schema->resultset('Artist')->create({
   name => 'Nine Inch Nails',
});

$nin->create_related('cds', {
   title => 'The Downward Spiral',
});

my $cd = $schema->resultset('Cd')->search_rs->first;

say $cd->full_title;
```

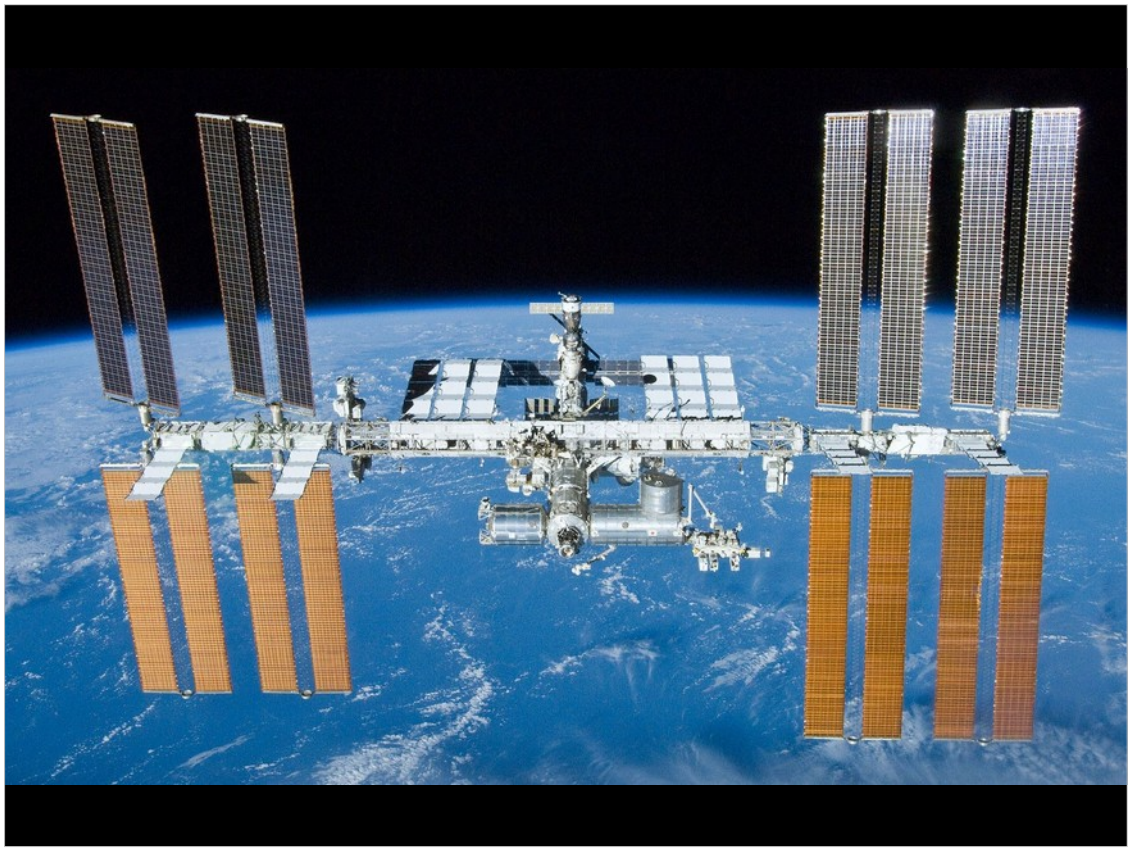
You may still remember this little script. It's the DBIx::Class example I showed earlier, where I wrote part of a resultset class in Perl 6. With the help of Inline::Perl6, we can still use this schema, including the Perl 6 extension.

In fact, I'm quite confident in telling you, that every feature of Inline::Perl5 I've shown you today, still works from Inline::Perl6. And why is that?

It's really just because Inline::Perl6 doesn't do all that much. It really just loads MoarVM, Rakudo and Inline::Perl5. It tells Inline::Perl5 to use the already running Perl 5 interpreter and it tricks MoarVM into thinking that it's just doing callbacks. That's why we have all the power still available.

And yes, you really can have a Perl 6 module that's using a Perl 5 module and use that then from Perl 5.

After 15 years of growing apart, our code bases can finally grow together again.



The Perl community speaks in many languages. Be it Perl 5 or regular expressions, or C or Perl 6 or Formats or Grammars.

We speak in many languages, but it's always the same voice. It is the voice, you can hear wandering around during coffee breaks at conferences like this.

It is the tiny, quiet voice in the back of your head that says “We are one, we are the Perl community”. Because whatever the arguments, whatever the opinions, whatever our preferences are. As long as it gets the job done and as long as we have fun, it just does not matter which part of our toolbox we use.

As long as we recognize this singular truth, that we are one.

Thank you.

Thank You!

http://niner.name/talks/
http://github.com/niner/

Bonus - List context

```
$schema.resultset('Cd').search.first;

$schema.resultset('Cd').search_rs.first;
```

You probably didn't notice, but the DBIC example was not 100 % plain DBIC code.

There is an important difference between Perl 5 and Perl 6. While Perl 6 adds tons of features, it also removed some. For example a constant source of bugs and surprises: the distinction between list and scalar context. The concept just doesn't exist there. Unfortunately, DBIx::Class makes use of context a lot.

Fortunately, it also provides a way to work around that.

So instead of using search, use search_rs. Same goes for the generated accessors.

Bonus - Scalar References

```
my $cd =
$schema.resultset('Cd').search_rs(
    {
        title => \('= artist.name'),
    },
    {
        join => 'artist',
    },
);
```

Another feature that was removed is references. We usually don't need them anymore, since we can pass around arrays and hashes just as they are.

And again, DBIx::Class does make use of references. Scalar references to be precise. It uses those to mark pieces of literal SQL that you can use instead of values.

In Perl 5 you can create a reference to any variable using the backslash operator. In Perl 6, it creates a Capture instead. A Capture “captures” what you're passing to a function. That could be positional and/or named arguments. This has nothing to do at all with references in Perl 5. Indeed, there is no matching concept in Perl 5. It just looks rather similar. So Inline::Perl5 misuses Captures to make up for the missing reference support. For you, that means, that you have to add some extra parentheses.