Meta-programming in Perl 6

Jonathan Worthington
jnthn | http://6guts.wordpress.com/

jnthn.WHO

From England, now living in Sweden

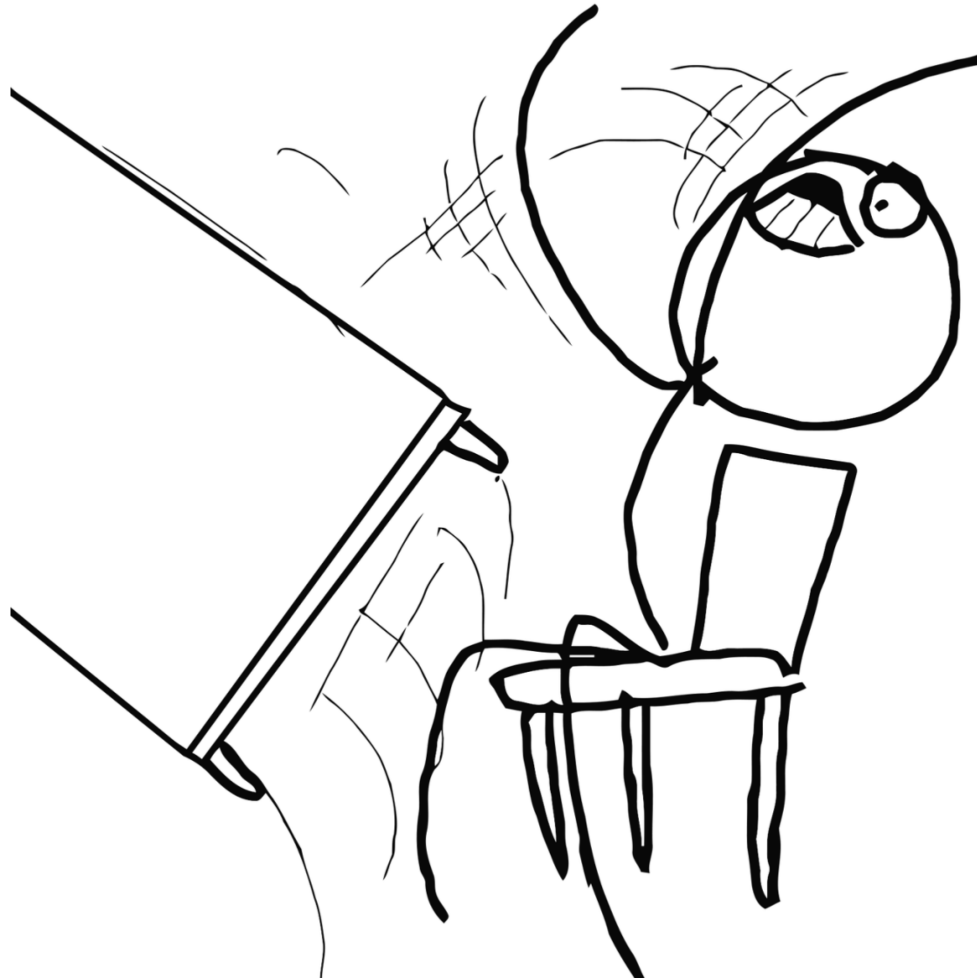
Rakudo Perl 6 core developer

Designer of 6model, the meta-object core that Rakudo builds upon

Beer drinker, serial traveller

Meta-programming is...

Meta-programming is...



Meta-programming is…

Programming

Meta-programming is...

$\text{Programming}^{\text{Programming}}$

Meta-programming is...

Programming the thing you do your programming with

Meta-programming is...

Hacking your language

Meta-circularity

Hacking your language

using your language

Meta-circularity

Using existing language features to...

Introspect them

Tweak them

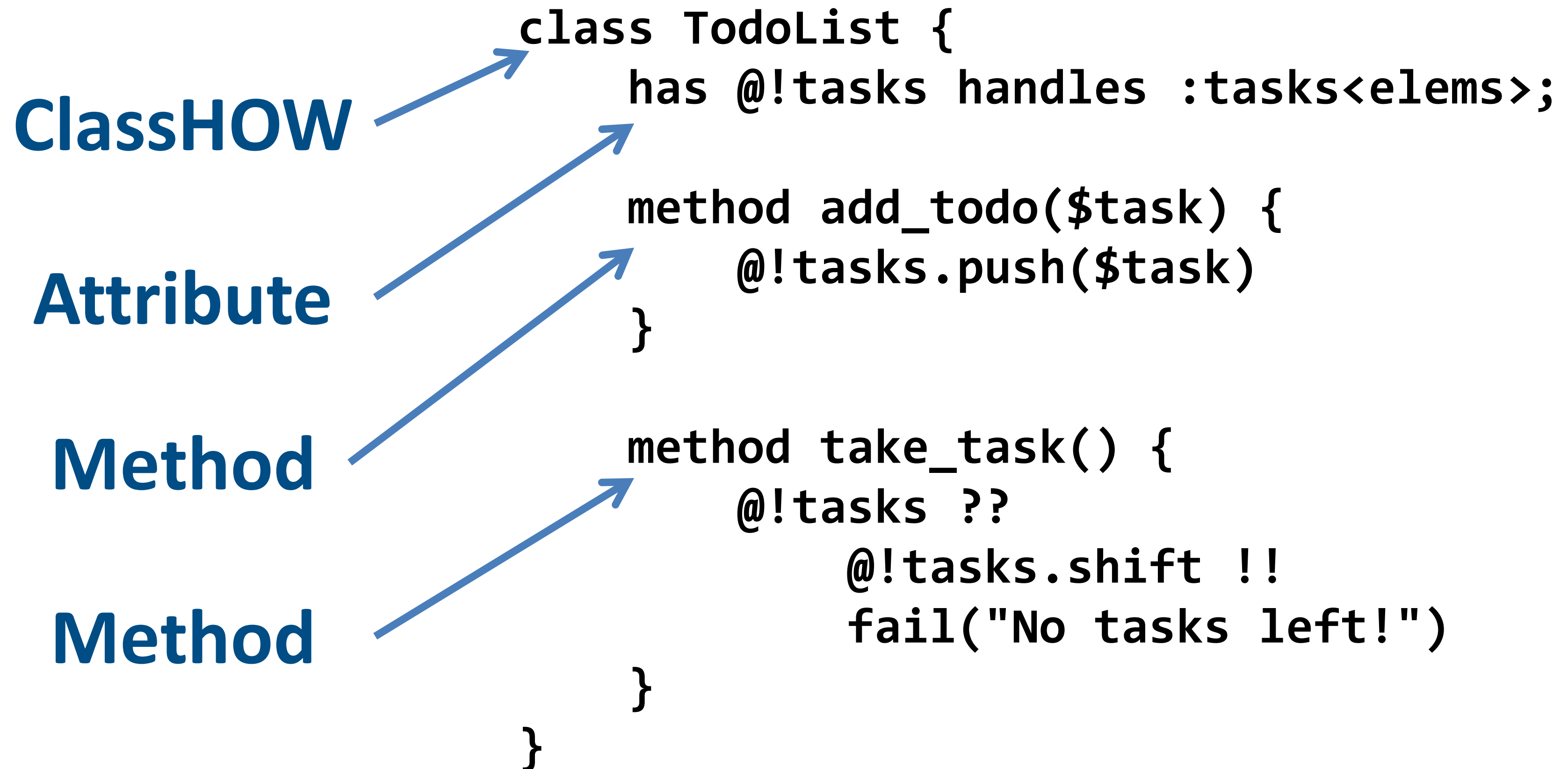
Build entirely new ones

Meta-objects

Declarations in Perl 6 programs usually lead to the creation of meta-objects

A meta-object is simply an object that defines how an element of our language works

Meta-objects



Introspection

Getting information from meta-objects

Can access information about...

Classes and Roles
Methods
Attributes
Signatures and parameters

Class Describer

Given a class, we want to output a list of attributes and methods

```
class TodoList {
    has @!tasks handles :tasks<elems>;
    method add_todo($task) {
        @!tasks.push($task)
    }
    method take_task() {
        @!tasks ??
            @!tasks.shift !!
            fail("No tasks left!")
    }
}
```

```
Type TodoList
  Attributes:
    @!tasks (private)
  Methods:
    add_todo
    take_task
    tasks
```

Class Describer: Outline

```
module Describe;

sub describe(::T) is export {
    ...
}
```

Class Describer: Outline

```
module Describe;

sub describe(::T) is export {
    join "\n", gather {
        ...
    }
}
```

Class Describer: Type name

```
module Describe;

sub describe(::T) is export {
    join "\n", gather {
        take "Type {T.^name}";
        ...
    }
}
```

Class Describer: Attributes

```
module Describe;

sub describe(::T) is export {
    join "\n", gather {
        take "Type {T.^name}";
        take "  Attributes:";
        for T.^attributes(:local) -> $attr {
            take "    $attr.name() ({
                $attr.has_accessor ?? 'public' !! 'private'
            })";
        }
        ...
    }
}
```

Class Describer: Methods

```
module Describe;

sub describe(::T) is export {
    join "\n", gather {
        take "Type {T.^name}";
        take "  Attributes:";
        for T.^attributes(:local) -> $attr {
            take "    $attr.name() ({
                $attr.has_accessor ?? 'public' !! 'private'
            })";
        }
        take "  Methods:";
        for T.^methods(:local).sort(*.name) -> $meth {
            take "    $meth.name()";
        }
    }
}
```

Type Construction

During compilation, the compiler makes instances of meta-objects and a series of method calls on them

Since meta-objects are just normal objects, we can also create instances of them

This enables us to dynamically create our own types

Class generation from JSON

We have a JSON file that describes various events that can happen in our system

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

Class generation from JSON

We'd like to build classes out of this, so that we can write code "as normal"...

```
use Events;

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

my $e2 = FlightCancelledEvent.new(
    flight_code => 'AB123',
    passenger_name => 'jnthn');
say $e2.flight_code;
say $e2.passenger_name;
```

Class generation from JSON

First, use JSON::Tiny to parse the JSON

```
module Events;
use JSON::Tiny;

my @events = @(from-json(slurp("events.json")));
for @events -> (:$name, :@values) {
    ...
}
```

Class generation from JSON

For each event, we create a new class...

```
module Events;
use JSON::Tiny;

my @events = @(from-json(slurp("events.json")));
for @events -> (:$name, :@values) {
    my $type := Metamodel::ClassHOW.new_type(:$name);
    ...
}
```

Class generation from JSON

...add attributes for each value...

```
module Events;
use JSON::Tiny;

my @events = @(from-json(slurp("events.json")));
for @events -> (:$name, :@values) {
    my $type := Metamodel::ClassHOW.new_type(:$name);
    for @values -> $attr_name {
        $type.HOW.add_attribute($type, Attribute.new(
            :name('$!' ~ $attr_name), :type(Mu),
            :has_accessor(1), :package($type)
        ));
    }
    ...
}
```

Class generation from JSON

...and compose the class.

```
module Events;
use JSON::Tiny;

my @events = @(from-json(slurp("events.json")));
for @events -> (:$name, :@values) {
    my $type := Metamodel::ClassHOW.new_type(:$name);
    for @values -> $attr_name {
        $type.HOW.add_attribute($type, Attribute.new(
            :name('$!' ~ $attr_name), :type(Mu),
            :has_accessor(1), :package($type)
        ));
    }
    $type.HOW.compose($type);
    ...
}
```

Class generation from JSON

Finally, we export the generated classes

```
module Events;
use JSON::Tiny;
package EXPORT::DEFAULT { }

my @events = @(from-json(slurp("events.json")));
for @events -> (:$name, :@values) {
    my $type := Metamodel::ClassHOW.new_type(:$name);
    for @values -> $attr_name {
        $type.HOW.add_attribute($type, Attribute.new(
            :name('$!' ~ $attr_name), :type(Mu),
            :has_accessor(1), :package($type)
        ));
    }
    $type.HOW.compose($type);
    EXPORT::DEFAULT.WHO{$name} := $type;
}
```

Just like the real thing

From the point of view of the user of the module, the classes are just as real as any written out in code

Same compile time analysis

(So you'll know about typos at compile time)

Just as efficient

(Because the compiler builds them this way too)

But it's slow!

One concern is that parsing JSON and building up the meta-objects takes time, so using the module will be costly

Rakudo supports pre-compilation of modules, but that still won't help at the moment, since we do all of the work in the mainline of the module

BEGIN to the rescue!

We can move all of our generation code into a BEGIN block...

```
module Events;
use JSON::Tiny;

package EXPORT::DEFAULT { }

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

BEGIN to the rescue!

All objects constructed and reachable once CHECK time is over will be serialized if the module is pre-compiled

This includes any meta-objects that we construct at BEGIN time

Thus, we need only do the JSON parse once when we pre-compile the module ☺

Hacking the language

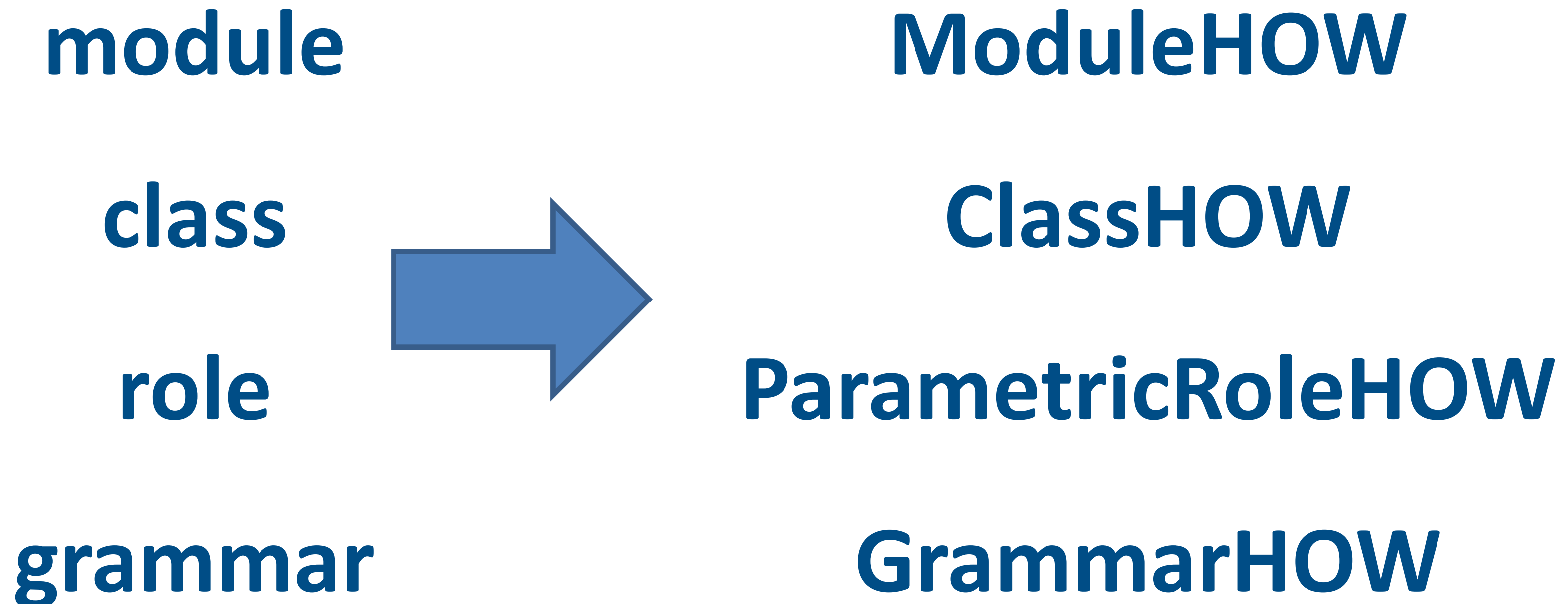
So far, we've used introspection to look at standard Perl 6 classes, or built them

We can also tweak the standard definition of these various meta-objects

This means we can change the way OO works, or extend it to support new features

Declarators and meta-objects

In Perl 5, we have the package keyword. In Perl 6, we have various kinds of package, with corresponding meta-object types...



Grammar::Tracer

The Grammar::Tracer module supplies a customized GrammarHOW that prints a trace of the grammar as it parses

```
TOP
|   country
|   |   name
|   |   * MATCH "Norway"
|   |   destination
|   |   |   name
|   |   |   * MATCH "Oslo"
|   |   |   num
|   |   |   * MATCH "59.914289"
|   |   |   num
|   |   |   * MATCH "10.738739"
|   |   |   integer
|   |   |   * MATCH "2"
|   |   * MATCH "\tOslo : 59.914289,10.738739 : 2\r\n"
|   |   destination
|   |   |   name
|   |   |   * MATCH "Bergen"
```

Inside Grammar::Tracer

First, we declare a class that inherits from GrammarHOW; we also derive from Mu

```
my class TracedGrammarHOW is Metamodel::GrammarHOW is Mu {
    ...
}
```

We enter it in EXPORTHOW, under a key corresponding to the package declarator

```
my module EXPORTHOW { }
EXPORTHOW.WHO.<grammar> = TracedGrammarHOW;
```

Inside Grammar::Tracer

We wish to intercept method calls on the grammar, so we override find_method

```
method find_method($obj, $name) {
    ...
}
```

Inside Grammar::Tracer

We defer to the normal method dispatcher to find the rule to call

```
method find_method($obj, $name) {
    my $meth := callsame;
    ...
}
```

Inside Grammar::Tracer

We skip over any guts-related methods, so they won't appear in the trace

```
method find_method($obj, $name) {
    my $meth := callsame;
    substr($name, 0, 1) eq '!'
    || $name eq any(<parse CREATE Bool defined MATCH>) ??
        $meth !!
        -> $c, |$args {
            ...
        }
}
```

Inside Grammar::Tracer

If we want to trace the method, we return a closure that will output the rule name...

```
method find_method($obj, $name) {
    my $meth := callsame;
    substr($name, 0, 1) eq '!'
    || $name eq any(<parse CREATE Bool defined MATCH>) ??
        $meth !!
        -> $c, |$args {
            say ('|  ' x $indent) ~ BOLD() ~ $name ~ RESET();
            ...
        }
}
```

Inside Grammar::Tracer

...then call it and capture the result, while tracking indentation...

```
method find_method($obj, $name) {
    my $meth := callsame;
    substr($name, 0, 1) eq '!'
    || $name eq any(<parse CREATE Bool defined MATCH>) ??
        $meth !!
        -> $c, |$args {
            say ('|  ' x $indent) ~ BOLD() ~ $name ~ RESET();
            $indent++;
            my $result := $meth($obj, |$args);
            $indent--;
            ...
        }
}
```

Inside Grammar::Tracer

...and finally print some output about the result, and return whatever the rule did

```
method find_method($obj, $name) {
    my $meth := callsame;
    substr($name, 0, 1) eq '!'
    || $name eq any(<parse CREATE Bool defined MATCH>) ??
        $meth !!
        -> $c, |$args {
            say ('|  ' x $indent) ~ BOLD() ~ $name ~ RESET();
            $indent++;
            my $result := $meth($obj, |$args);
            $indent--;
            describe($result);
            $result
        }
}
```

Inside Grammar::Tracer

The (relatively boring) output methods aside, there's only one thing left to do

For performance, most method dispatches are done through a cache; we need to prevent publication of the cache, so that our find_method override is always called

```
method publish_method_cache($obj) {
    # Suppress this, so we always hit find_method.
}
```

Scope of our meta-class

When a use statement is done, it looks for EXPORTHOW and imports from it

Therefore, any grammars in any modules we are using will not end up traced - only the one that we are interested in ☺

Perl 6 is designed to ensure that language tweaks apply lexically → safe!

Really simple AOP

Aspect Oriented Programming helps to factor out cross-cutting concerns

For example, we may wish to apply logging to every method in a class

We can build a really, really simple AOP implementation for Perl 6 in around 30 lines

Aspects

For the purposes of this example, we'll mandate that all aspects will inherit from the case class MethodBoundaryAspect

```
my class MethodBoundaryAspect is export {
}
```

It is just a simple "marker" class, which we'll use to detect the usage of an aspect

Applying aspects to a class

The "is" keyword is a trait modifier, and maps to a (compile time) multiple dispatch

We add an extra implementation that will call add_aspect when an aspect is used

```
multi trait_mod:(Mu:U $type, MethodBoundaryAspect:U $aspect)
      is export {
    $aspect === MethodBoundaryAspect ??
        $type.HOW.add_parent($type, $aspect) !!
        $type.HOW.add_aspect($type, $aspect);
}
```

Custom meta-class

It starts off just the same…

```
my class ClassWithAspectsHOW is Mu is Metamodel::ClassHOW {
    ...
}
```

Custom meta-class

Added aspects are stored in an attribute

```
my class ClassWithAspectsHOW is Mu is Metamodel::ClassHOW {
    has @!aspects;
    method add_aspect(Mu $obj, MethodBoundaryAspect:U $aspect) {
        @!aspects.push($aspect);
    }
    ...
}
```

Custom meta-class

We hook compose to apply the aspects

```
my class ClassWithAspectsHOW is Metamodel::ClassHOW is Mu {
    has @!aspects;
    method add_aspect(Mu $obj, MethodBoundaryAspect:U $aspect) {
        @!aspects.push($aspect);
    }
    method compose(Mu $obj) {
        for @!aspects -> $a {
            self.apply_aspect($obj, $a);
        }
        callsame;
    }
}
```

Custom meta-class

Finally, the apply_aspect method

```
method apply_aspect(Mu $obj, $a) {
    for self.methods($obj, :local) -> $m {
        $m.wrap(-> $obj, |$args {
            $a.?entry($m.name, $obj, $args);
            my $result := callsame;
            $a.?exit($m.name, $obj, $args, $result);
            $result
        });
    }
}
```

Example of using AOP

```
use aspects;

class LoggingAspect is MethodBoundaryAspect {
    method entry($method, $obj, $args) {
        say "Called $method with $args";
    }
    method exit($method, $obj, $args, $result) {
        say "$method returned with $result.perl()";
    }
}

class Example is LoggingAspect {
    method double($x) { $x * 2 }
    method square($x) { $x ** 2 }
}

say Example.double(3);
say Example.square(3);
```

In conclusion...

Meta-programming opens up the declarative parts of the language for...

Introspection
Runtime creation
Tweaking and extending

All of the examples demonstrated today already work on Rakudo Perl 6 \☺/

Future directions

Make it possible to build meta-class implementations "from scratch", rather than subclassing an existing one

Announcements, so meta-objects can tell each other about runtime changes

More robustness, more optimizations

Thank you!

Questions?

Blog: http://6guts.wordpress.com/
Twitter: jnthnwrthngtn
Email: jnthn@jnthn.net