

Perl 6 Tutorial



Андрей Шитов & Джонатан Вортингтон
Український воркшоп «Перл мова» 2008

Perl 6 Tutorial: Introduction

The Plan

- About two hours
- Two speakers (Андрей and myself)
- Two languages (English and Russian)
- Two things you should do:
 - Ask questions if you don't understand
 - Yell at me to slow down if I talk too fast

Perl 6 Tutorial: Introduction

What we'll be talking about

- Introduction – Джонатан
- Basic Syntax – Андрей
- Junctions – Джонатан
- Contexts And Subs – Андрей
- Object Orientation – Джонатан
- Smart Matching – Андрей
- Rules And Grammars – Джонатан

Perl 6 Tutorial: Introduction

What is Perl 6?

- The next version of the Perl language
- A complete re-design
- No official implementation
 - Official grammar (specifies syntax)
 - Official specification (specifies semantics)
 - Official test suite (considered part of the specification)

Perl 6 Tutorial: Introduction

Implementations

- There are several implementation efforts; amongst them are:
 - Pugs – written in Haskell; has many features but not maintained much now
 - kp6 – bootstrapping approach; aiming to write Perl 6 in Perl 6, coming along
 - Rakudo – Perl 6 compiler for Parrot; perhaps the most active effort at the moment

Perl 6 Tutorial: Introduction

Isn't it taking a while?

- Yes! 😊
- Because...
 - Designing a large, complex and good language takes a long time
 - Implementing that language also takes a long time and can be hard to do right
 - Most people working on this are volunteers

Perl 6 Tutorial: Introduction

When will we get a full implementation?

- The fun answer: Christmas
 - However, which year is not specified
 - “Every day will be like Christmas when we have Perl 6!”
- The self-referential answer: when there is an implementation that meets the specification and passes the test suite
- Too early to give a realistic date yet

Perl 6 Tutorial: Introduction

What can I run today?

- The vast majority of the code examples that we will be showing today work in Rakudo Perl 6
- Admittedly, I sort of cheated...
 - I hack on Rakudo...
 - ...and over the last couple of weeks, have been implementing features that I planned to talk about here today. 😊

Perl 6 Tutorial: Introduction

How can you help?

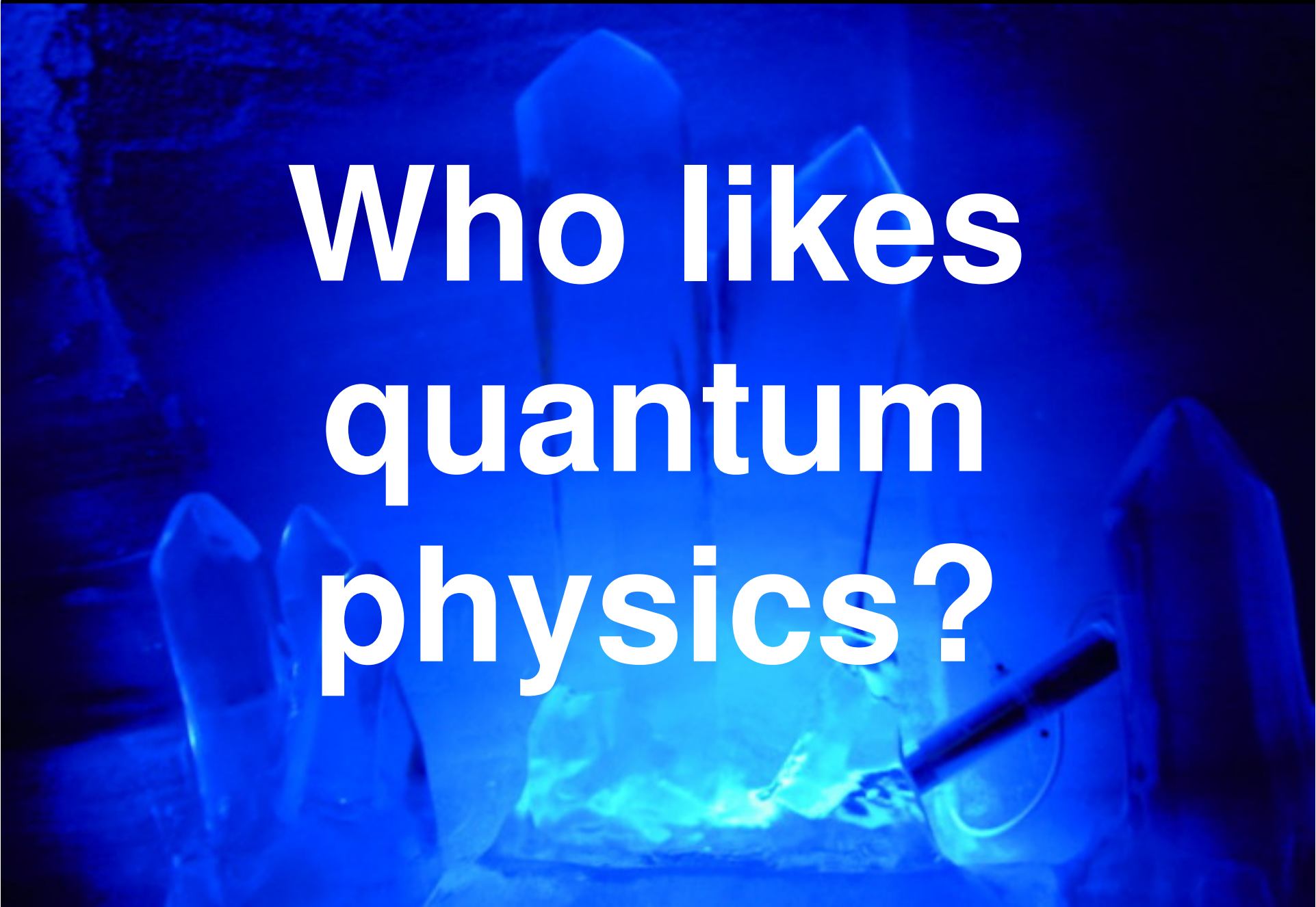
- Grab a copy of one of the Perl 6 implementations that is under active development, play with it and report any problems
- Tell us what you've found hard to understand in today's talk; Perl 6 is meant to be understandable
- Or come along and join us in hacking on the implementation

**Any questions
so far?**

Perl 6 Tutorial: Basic Syntax

~~ Андрей ~~

Junctions



**Who likes
quantum
physics?**

Quantum Superposition

- The theory: if you don't know what state something is in, then it is simultaneously in all states until you observe it.
- Example:
 - I'm about to give my junctions talk
 - However, you haven't observed it yet
 - Therefore, my talk is simultaneously awful and totally awesome

Perl 6 Tutorial: Junctions

One value that is many at the same time

- We know that an array contains many values, and a scalar contains a single value

```
my $scalar = 42;  
my @array = (1, 2, 3);
```

- A junction...
 - Is a scalar, but...
 - Has many values at the same time
 - "A quantum superposition of values"

Perl 6 Tutorial: Junctions

A simple example

- How often do you find yourself writing things like:

```
if $drink eq 'vodka' || $drink eq 'beer' {  
    say "Don't get drunk on it!";  
}
```

- With junctions we can write this as:

```
if $drink eq 'vodka' | 'beer' {  
    say "Don't get drunk on it!";  
}
```

- "vodka" | "beer" is a junction

Perl 6 Tutorial: Junctions

Another example

- Another example: looping while a counter stays below both of two different limits:

```
while $count < $lim_a && $count < $lim_b {  
    ...  
}
```

- Can be re-written as:

```
while $count < $lim_a & $lim_b {  
    ...  
}
```

Perl 6 Tutorial: Junctions

Constructing Junctions From Arrays

- There are functions to construct junctions from arrays too:

```
if all(@scores) > $pass_mark {  
    say "Everybody passed!";  
}  
if $word eq none(@mat) {  
    say "$word is not swearing!";  
}  
if any(@scores_a) > all(@scores_b) {  
    say "Someone in class A beat all of " ~  
        "the people in class B!";  
}
```

Perl 6 Tutorial: Junctions

Types Of Junction

- **"any"** junctions – require one or more of the values to match the condition (|)
- **"all"** junctions require all of the values to match the condition (&)
- **"one"** junctions require exactly one of the values to match the condition (^)
- **"none"** junctions require none of the values in the junction to match the condition

Perl 6 Tutorial: Junctions

Auto-threading

- You can use any operator on a junction that you would have used with a scalar, and the result will be a new junction of results.

```
my $a = 1 | 2;  
$a++;           # Now we have 3 | 4  
$a = $a + 5;    # Now we have 8 | 9  
  
my $res = "foo" ~ "bar" & "baz";  
           # "foobar" & "foobaz"
```

Perl 6 Tutorial: Junctions

Auto-threading

- If you call a function and one of the arguments to it is a junction, by default it will call the function once for each value in the junction.
- Therefore:

```
say "hello" & "goodbye";
```

- Will call the **say** function twice, producing two lines of output

Perl 6 Tutorial: Junctions

Auto-threading

- If you have multiple functional arguments, it will call it on all permutations.
- So if greet works like this:

```
greet("Hello", "Andrew"); # Hello Andrew
```

- Then with two junctions, we can do:

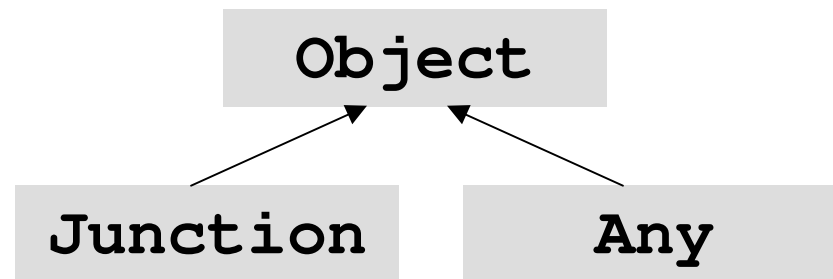
```
greet("Hi" & "Bye", "Andrew" & "Jonathan");
```

- And get four lines of output (**BUT** we get no promises about the ordering)

Perl 6 Tutorial: Junctions

If you don't want to auto-thread...

- Perl 6 has a type system
- The top level of the class hierarchy looks like this:



- The default type of a parameter to a sub is Any; if you give something a type of Junction or Object, it won't thread.

Questions on Junctions?

Object Orientation

Classes

What Are Classes Used For?

- Instance Management
 - Classes “create” objects
 - Alternatively, you can view a class as a kind of blueprint for how to create an object
 - Classes define both the state and behaviour that an object has, and relate them

Perl 6 Tutorial: Object Orientation

What Are Classes Used For?

- Code re-use
 - We often try to design classes to do one particular thing
 - That means that, ideally, they can be re-used to do that thing multiple times, potentially in multiple programs

What Are Classes Used For?

- Providing a route to polymorphism
 - This means that the same code can safely operate on values of different types
 - Inheritance relationships state that a subclass can be used in place of any of its parent classes
 - Enables more code re-use

Perl 6 Tutorial: Object Orientation

Classes In Perl 6

- Introduce a class using the `class` keyword
 - With a block:

```
class Puppy {  
    ...  
}
```

- Or without to declare that the rest of the file describes the class.

```
class Puppy;
```





**They called me
WHAT?!**



They called me
WHAT?!

white



They called me
WHAT?!

white

4 paws



They called me
WHAT?!

white

4 paws

tail

Perl 6 Tutorial: Object Orientation

Attributes

- Introduced using the **has** keyword

```
class Puppy {  
    has $name;  
    has $colour;  
    has @paws;  
    has $tail;  
}
```

- All attributes in Perl 6 are stored in an opaque data type
- Hidden to code outside of the class

Perl 6 Tutorial: Object Orientation

Accessor Methods

- We want to allow outside access to some of the attributes
- Writing accessor methods is boring!
- `$.` means it is automatically generated

```
class Puppy {  
    has $.name;  
    has $.colour;  
    has @paws;  
    has $tail;  
}
```

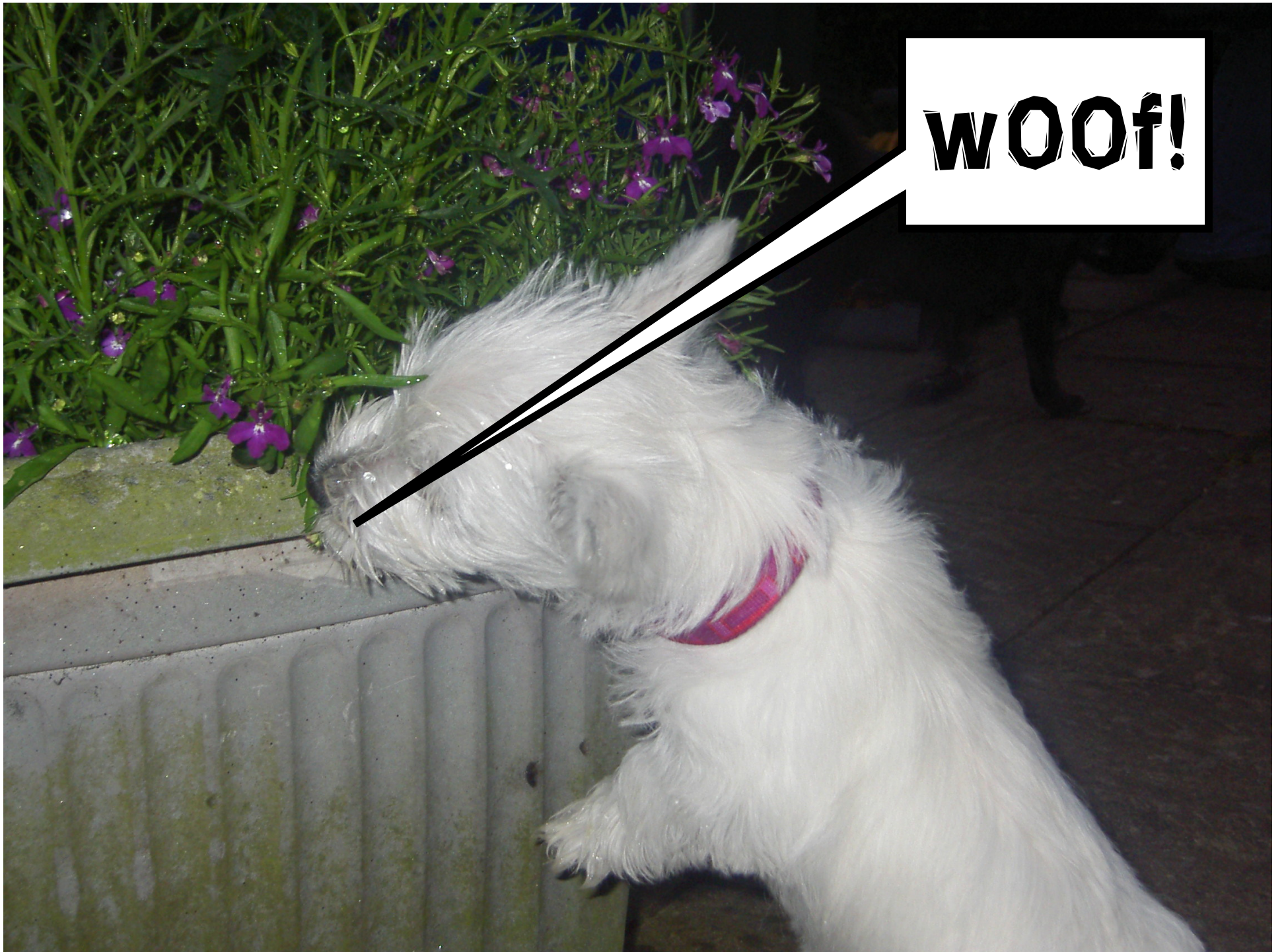
Perl 6 Tutorial: Object Orientation

Mutator Methods

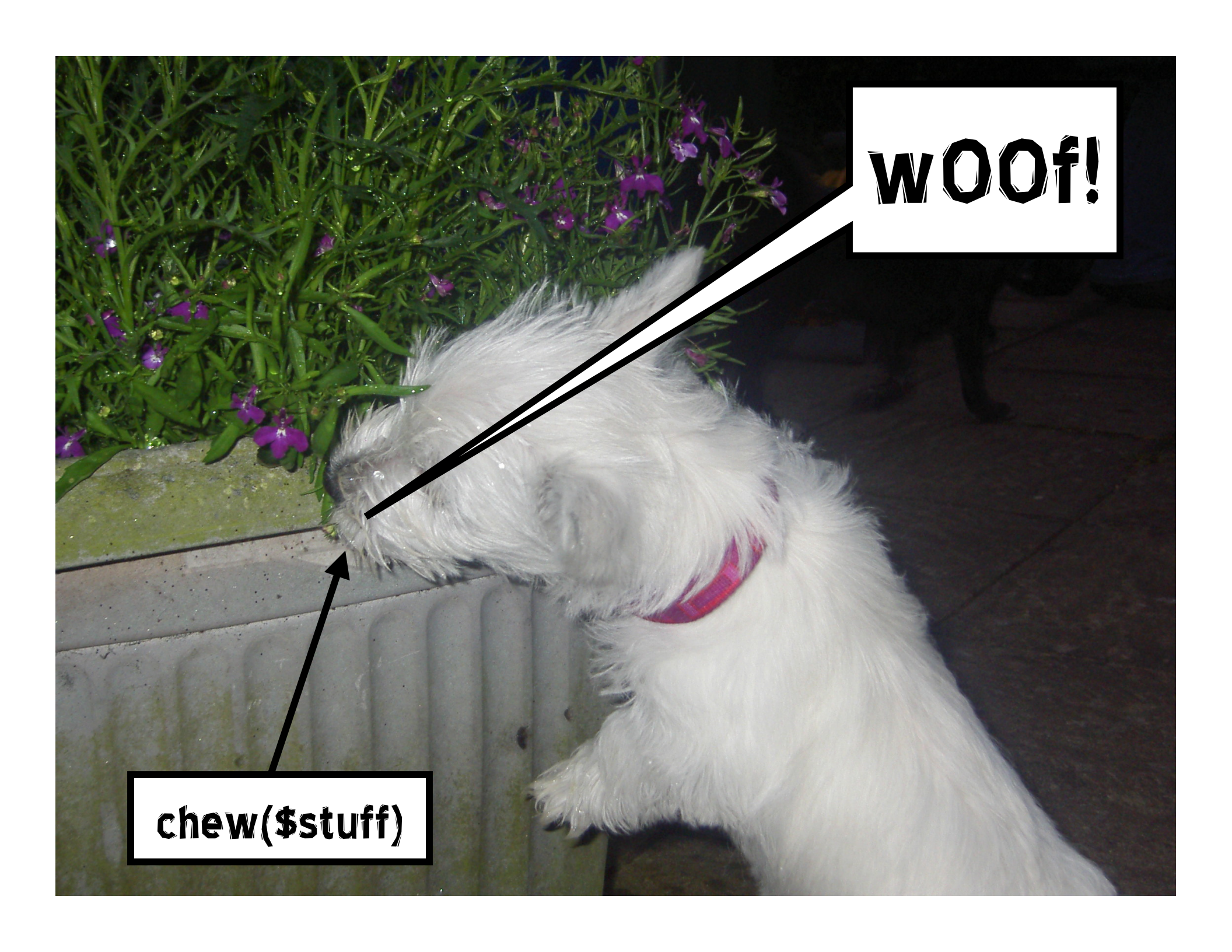
- We should be able to change some of the attributes
- Use `is rw` to generate a mutator method too

```
class Puppy {  
    has $.name is rw;  
    has $.colour;  
    has @paws;  
    has $tail;  
}
```



woof!



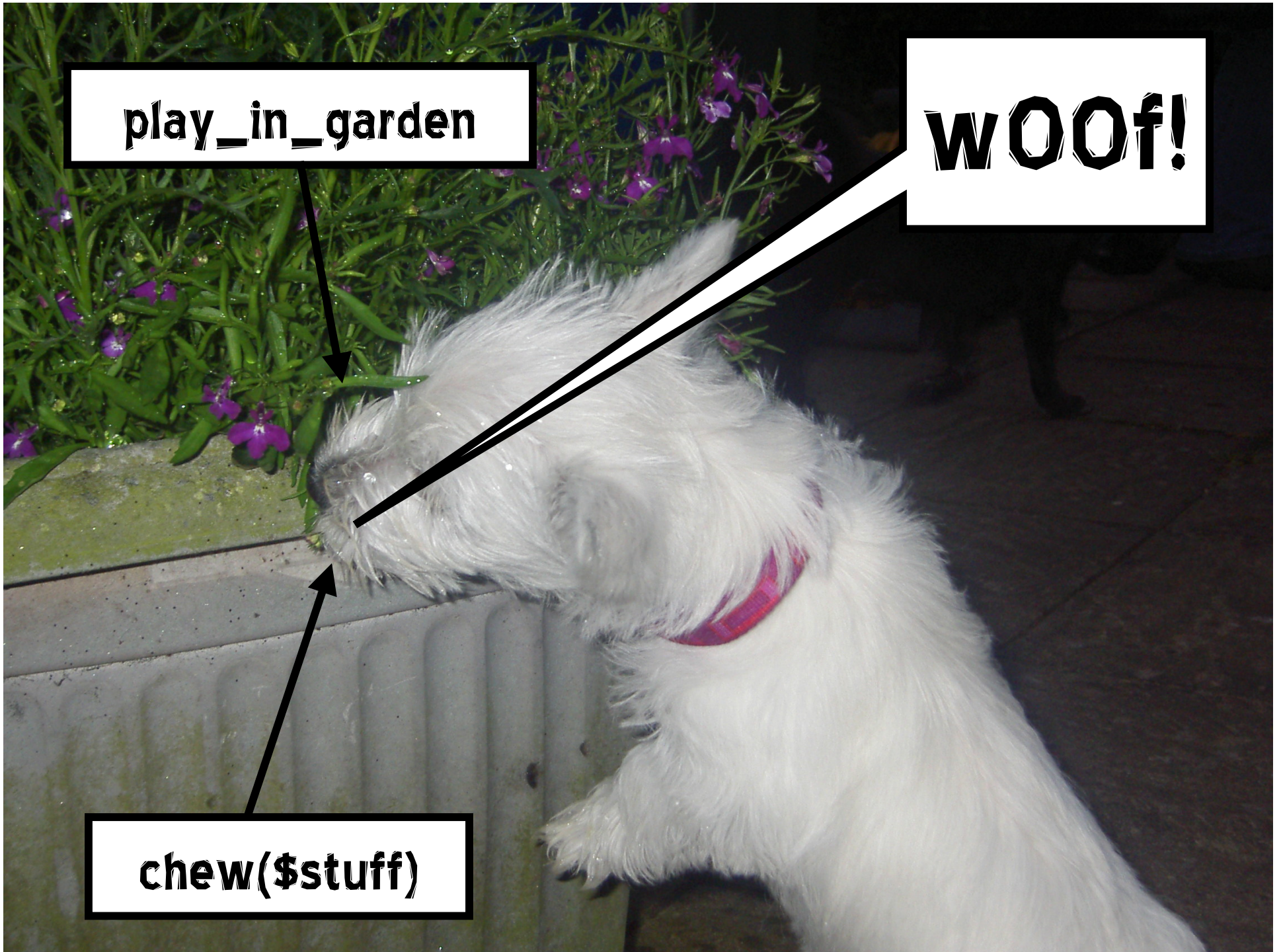
wOOf!

chew(\$stuff)

play_in_garden

woof!

chew(\$stuff)

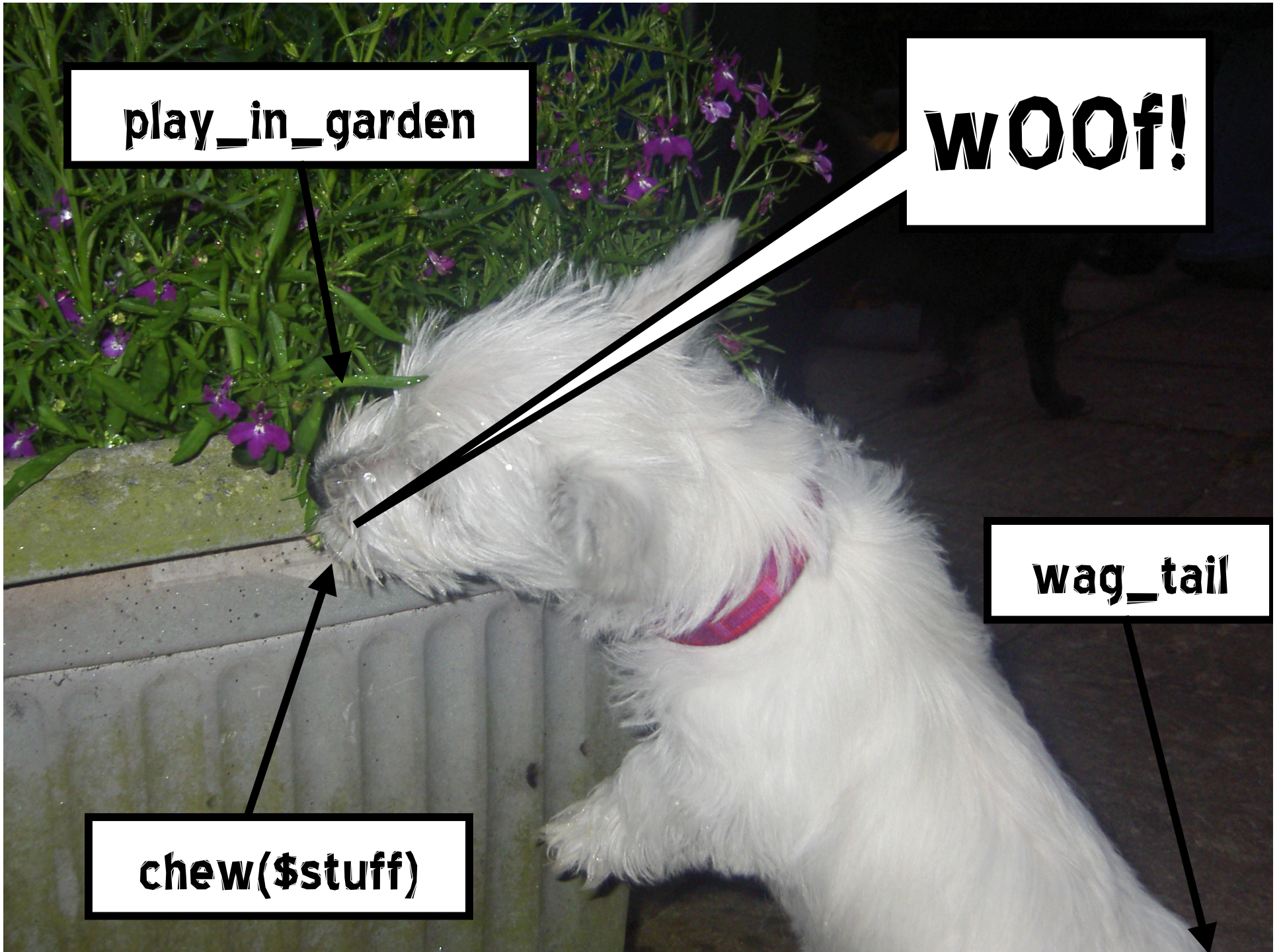


play_in_garden

woof!

wag_tail

chew(\$stuff)



Perl 6 Tutorial: Object Orientation

Methods

- The new **method** keyword is used to define a method

```
method bark() {  
    say "w00f!";  
}
```

- Parameters go in a parameter list; the invocant is optional!

```
method chew($item) {  
    $item.damage++;  
}
```

Perl 6 Tutorial: Object Orientation

Attributes In Methods

- Attributes can be accessed with the `$.` syntax, via their accessor

```
method play_in_garden() {  
    $.colour = 'black';  
}
```

- To get at the actual storage location, `$!colour` can be used

```
method play_in_garden() {  
    $!colour = 'black';  
}
```

Perl 6 Tutorial: Object Orientation

Using A Class

- A default **new** method is generated for you that sets attributes
- Also note that **->** has become **.**

```
my $puppy = Puppy.new(  
    :name('Rosey'),  
    :colour('white')  
);  
$puppy.bark();           # w00f!  
say $puppy.colour;       # white  
$puppy.play_in_garden();  
say $puppy.colour;       # black
```

Perl 6 Tutorial: Object Orientation

Inheritance

- A puppy is really a dog, so we want to implement a Dog class and have Puppy inherit from it
- Inheritance is achieved using the **is** keyword

```
class Dog {  
    ...  
}  
class Puppy is Dog {  
    ...  
}
```

Perl 6 Tutorial: Object Orientation

Multiple Inheritance

- Multiple inheritance is possible too; use multiple **is** statements

```
class Puppy is Dog is Pet {  
    ...  
}
```


Roles

Perl 6 Tutorial: Object Orientation

In Search Of Greater Re-use

- In Perl 6, roles take on the task of re-use, leaving classes to deal with instance management
- We need to implement a **walk** method for our **Dog** class
- However, we want to re-use that in the **Cat** and **Pony** classes too
- What are our options?

Perl 6 Tutorial: Object Orientation

The Java, C# Answer

- There's only single inheritance
- You can write an interface, which specifies that a class must implement a **walk** method
- Write a separate class that implements the **walk** method
- You can use delegation (hand coded)
- Sucks

Perl 6 Tutorial: Object Orientation

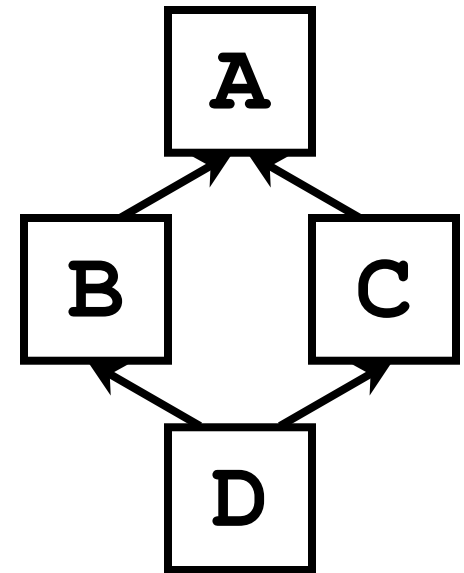
The Multiple Inheritance Answer

- Write a separate class that implements the **walk** method
- Inherit from it to get the method
- Feels wrong linguistically
 - “A dog is a walk” – err, no
 - “A dog does walk” – what we want
- Multiple inheritance has issues...

Perl 6 Tutorial: Object Orientation

Multiple Inheritance Issues

- The diamond inheritance problem
 - Do we get two copies of A's state?
 - If B and C both have a **walk** method, which do we choose?
- Implementing multiple inheritance is tricky too



Perl 6 Tutorial: Object Orientation

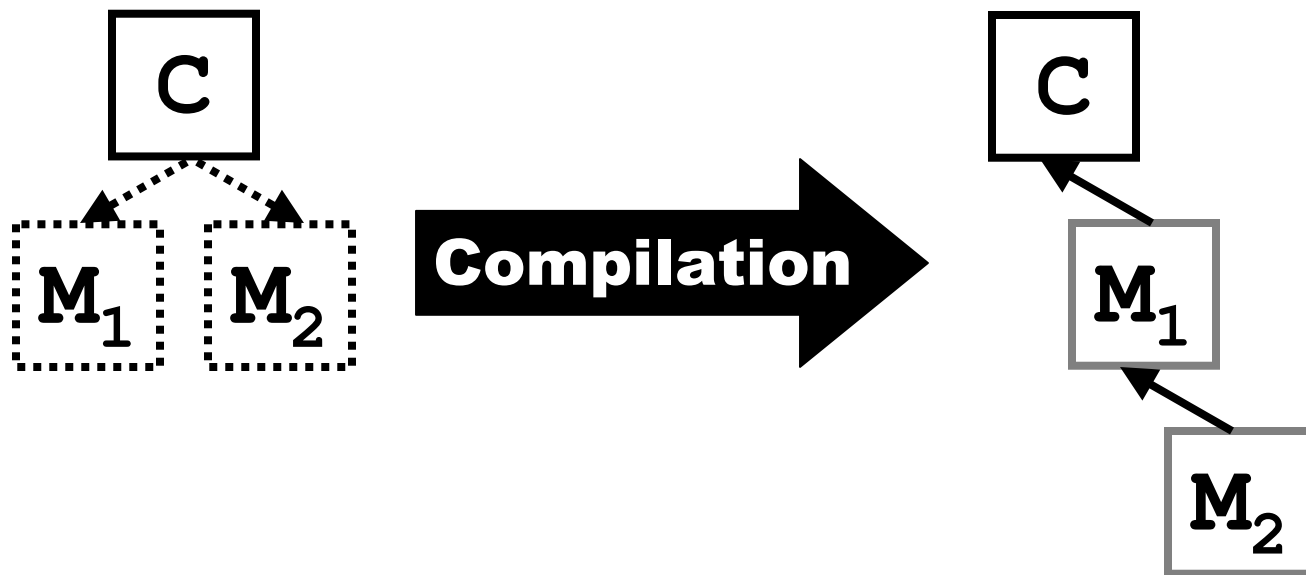
Mix-ins

- A mix-in is a group of one or more methods that can not be instantiated on their own
- We take a class and “mix them in” to it
- Essentially, these methods are added to the methods of that class
- Write a **Walk** mixin with the **walk** method, mix it in.

Perl 6 Tutorial: Object Orientation

How Mix-ins Work

- Defined in terms of single inheritance



- Take C and derive anonymous classes with methods of M_1 , then M_2

Perl 6 Tutorial: Object Orientation

Issues With Mix-ins

- If M_1 and M_2 both have methods of the same name, which one is chosen is dependent on the order that we mix in
 - Fragile hierarchies problem again
- Further, mix-ins end up overriding a method of that name in the class, so you can't decide which mix-in's method to actually call in the class itself

Perl 6 Tutorial: Object Orientation

The Heart Of The Problem

- The common theme in our problems is the inheritance mechanism
- Need something else in addition
- We want
 - To let the class be able to override any methods coming from elsewhere
 - Explicit detection and resolution of conflicting methods

Perl 6 Tutorial: Object Orientation

Flattening Composition

- A role, like a mix-in, is a group of methods
- If a class **does** a role, then it will have the methods from that role, however:
 - If two roles provide the same method, it's an error, unless the class provides a method of that name
 - Class methods override role methods

Perl 6 Tutorial: Object Orientation

Creating Roles

- Roles are declared using the `role` keyword
- Methods declared just as in classes

```
role Walk {  
    method walk($num_steps) {  
        for 1..$num_steps {  
            .step for @paws;  
        }  
    }  
}
```

Perl 6 Tutorial: Object Orientation

Composing Roles Into A Class

- Roles are composed into a class using the **does** keyword

```
class Dog does Walk {  
    ...  
}
```

- Can compose as many roles into a class as you want
- Conflict checking done at compile time
- Works? Not quite...

Perl 6 Tutorial: Object Orientation

Composing Roles Into A Class

- Notice this line in the `walk` method:

```
.step for @paws;
```

- Can state that a role “shares” an attribute with the class it is composed into using `has` without `.` or `!`

```
has @paws;
```

Questions?

Perl 6 Tutorial: Smart Matching

~~ Андрей ~~

Regexes & Grammars

Perl 6 Tutorial: Regexes And Grammars

What I'll Cover

- Could spend a whole day on this topic
- In this quick review, I'll look at:
 - What's not changing from Perl 5
 - Some of the changes to notable features
- Then we will have a look at named regexes along with grammars

Perl 6 Tutorial: Regexes And Grammars

What's Staying The Same

- You can still write regexes between slashes
- The `?`, `+` and `*` quantifiers
- `??`, `+`? and `*?` lazy quantifiers
- `(...)` is still used for capturing
- Character class shortcuts: `\d`, `\w`, `\s`
- `|` for alternations (but semantics are different; use `||` for the Perl 5 ones)

Perl 6 Tutorial: Regexes And Grammars

Change: Literals And Syntax

- Anything that is a number, a letter or the underscore is a literal

```
/foo_123/      # All literals
```

- Anything else is syntax
- You use a backslash (\) to make literals syntax and to make syntax literals

```
/\<\w+\>/      # \< and \> are literals  
                # \w is syntax
```

Perl 6 Tutorial: Regexes And Grammars

Change: Whitespace

- Now what was the x modifier in Perl 5 is the default
- This means that spaces don't match anything – they are syntax

```
/abc/      # matches abc  
/a b c/    # the same
```

Perl 6 Tutorial: Regexes And Grammars

Change: Quoting

- Single quotes interpret all inside them as a literal (aside from \')
- Can re-write:

```
/\<\w+\>/
```

As the slightly neater:

```
/'<' \w+ '>'/
```

- Spaces are literal in quotes too:

```
/'a b c'/ # requires the spaces
```

Perl 6 Tutorial: Regexes And Grammars

Change: Grouping

- A non-capturing group is now written as `[...]` (rather than `(?:...)` in Perl 5)

```
/[foo|bar|baz]+/
```

- Character classes are now `<[...]>`; they are negated with `-`, combined with `+` or `-` and ranges are expressed with `..`

```
/<[A..Z]>/          # uppercase letter...  
/<[A..Z] - [AEIOU]>/ # ...but not a vowel  
/<[\w + [-]]>       # anything in \w or a -
```

Perl 6 Tutorial: Regexes And Grammars

Change: s and m

- The **s** and **m** modifiers are gone
- **.** now always matches anything, including a new line character
- Use **\N** for anything but a new line
- **^** and **\$** always mean start and end of the string
- **^^** and **\$\$** always mean start and end of a line

Perl 6 Tutorial: Regexes And Grammars

Matching

- To match against a pattern, use `~~`

```
if $event ~~ /\d**4/ { ... }
```

- Negated form is `!~~`

```
if $event ~~ /\d**4/ { say "missing year"; }
```

- `$/` holds the match object; when used as a string, it is the matched text

```
my $event = "Ukrainian Perl Workshop 2008";  
if $event ~~ /\d**4/ {  
    say "Held in $/"; # Held in 2008  
}
```


Perl 6 Tutorial: Regexes And Grammars

Named Regexes

- You can now declare a regex with a name, just like a sub or method

```
regex Year { \d**4 }; # 4 digits
```

- Then name it to match against it:

```
if $event ~~ Year { ... }
```

Perl 6 Tutorial: Regexes And Grammars

Calling Other Regexes

- You can "call" one regex from another, making it easier to build up complex patterns.

```
regex Year { \d**4 };
regex Place { Ukrainian | Dutch | German };
regex Workshop {
    <Place> \s Perl \s Workshop \s <Year>
};
regex YAPC {
    'YAPC::' ['EU' | 'NA' | 'Asia'] \s <Year>
};
regex PerlEvent { <Workshop> | <YAPC> };
```

Perl 6 Tutorial: Regexes And Grammars

The Match Object

- Can extract the year from a list of event names like this:

```
for @events -> $ev {  
    if $ev ~~ PerlEvent {  
        if $/<YAPC> {  
            say $/<YAPC><Year>;  
        } else {  
            say $/<Workshop><Year>;  
        }  
    } else {  
        say "$ev was not a Perl event.";  
    }  
}
```

Perl 6 Tutorial: Regexes And Grammars

rule and token

- By default, regexes backtrack
- Not very efficient for building parsers
- If you use **token** or **rule** instead of **regex**, it will not backtrack
- Additionally, **rule** will replace any literal spaces in the regex with a call to **ws** (**<.ws>**), which you can customize for the thing you are parsing

Perl 6 Tutorial: Regexes And Grammars

Using Perl 6 to parse Perl 6

- Rakudo's Perl 6 compiler has its parser written using Perl 6 rules

```
rule if_statement {  
    $<sym>=[if]  
    <EXPR> <block>  
    [ 'elsif' <EXPR> <block> ]*  
    [ 'else' $<else>=<block> ]?  
    {*}  
}
```

- The regex implementation is one of the most complete parts of Rakudo

Closing Thoughts

Perl 6 Tutorial: Closing Thoughts

Links

- Follow the progress of Rakudo or get involved at:
www.rakudo.org
- Perl 6 language specification available at:
dev.perl.org/perl6/
- Unofficial Perl 6 FAQ at:
www.programmersheaven.com/2/Perl6-FAQ

Perl 6 Tutorial: Closing Thoughts

Closing Thoughts

- Thanks for having me at your workshop, even though I can't speak **any (' Russian ' , ' Ukrainian ')**
- Perl 6 has taken a while to design, and is taking a while to implement (because they designed a lot of things ;-))
- Plenty of people working to make it happen; more help always needed!

Questions?