Clarifying Roles

Jonathan Worthington

German Perl Workshop 2007

The Perl 6 Object Model

- The Perl 6 object model attempts to improve on the Perl 5 one
  - Nicer, more declarative syntax
  - One way to do things, rather than the many that appeared in Perl 5 (but you can still do other stuff if you like)
  - Roles – the subject of this talk
- Before roles, a look at classes…

Classes In Perl 6

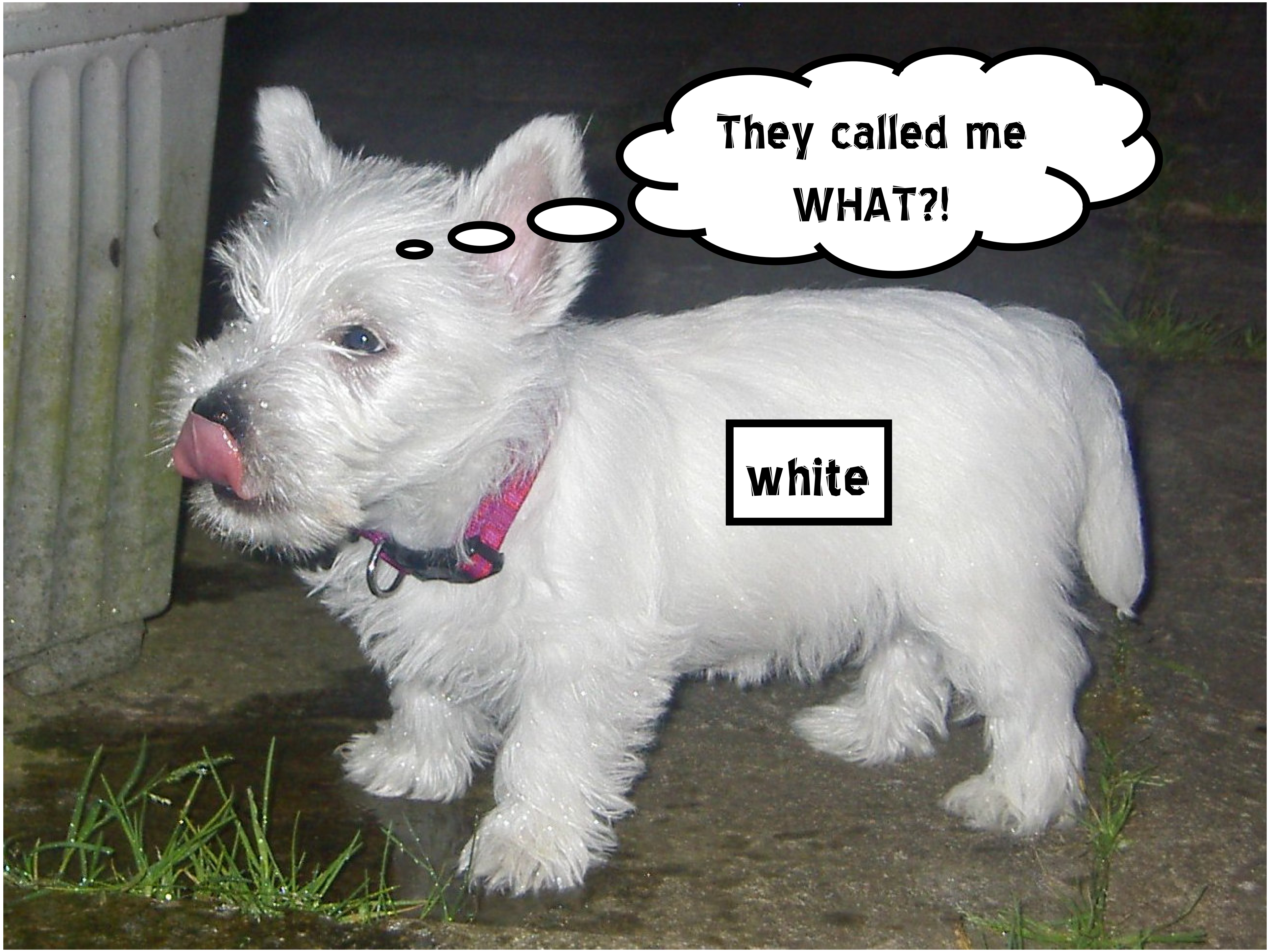
- Introduce a class using the **`class`** keyword
  - With a block:

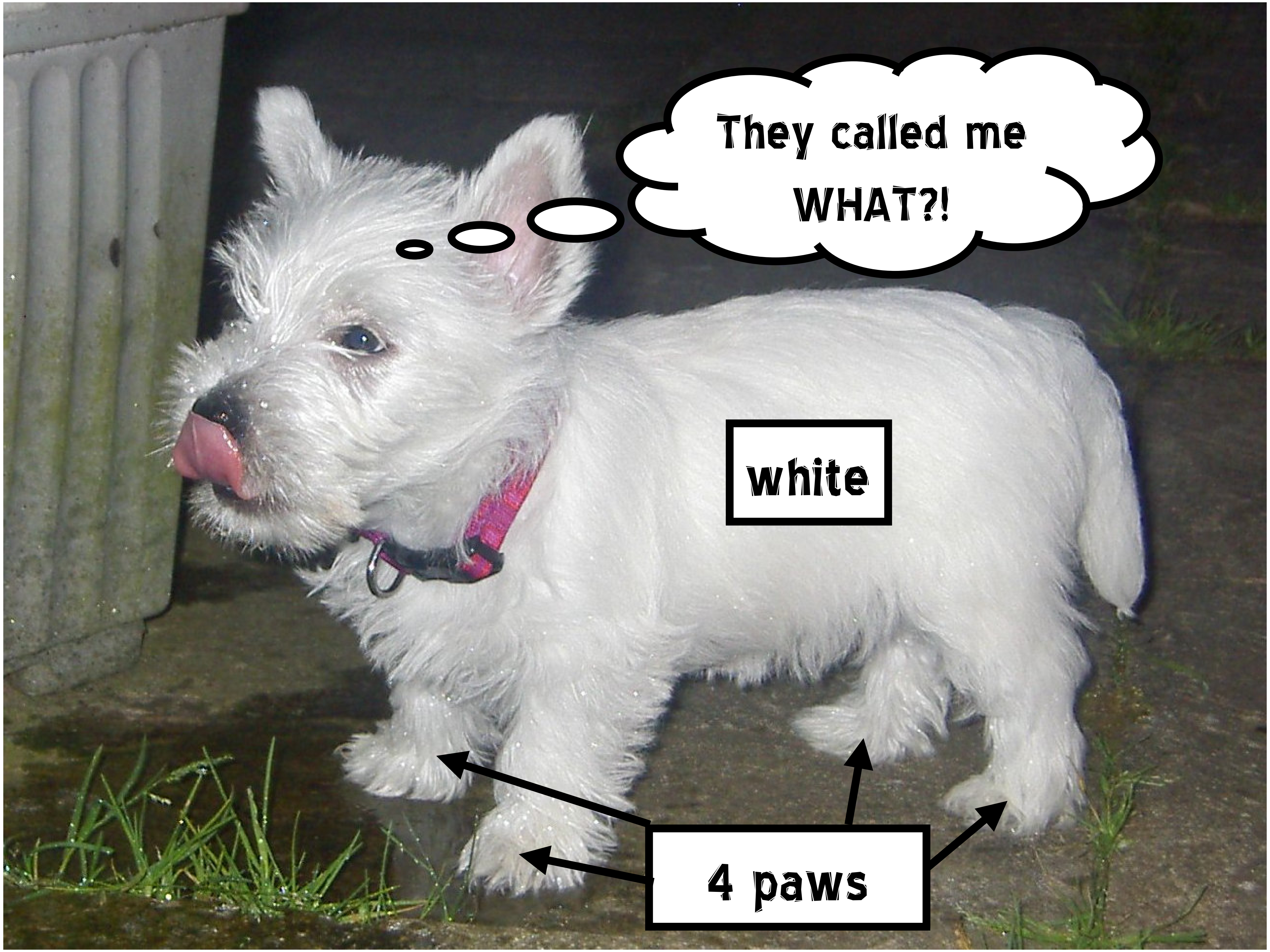
```
class Puppy {
    …
}
```

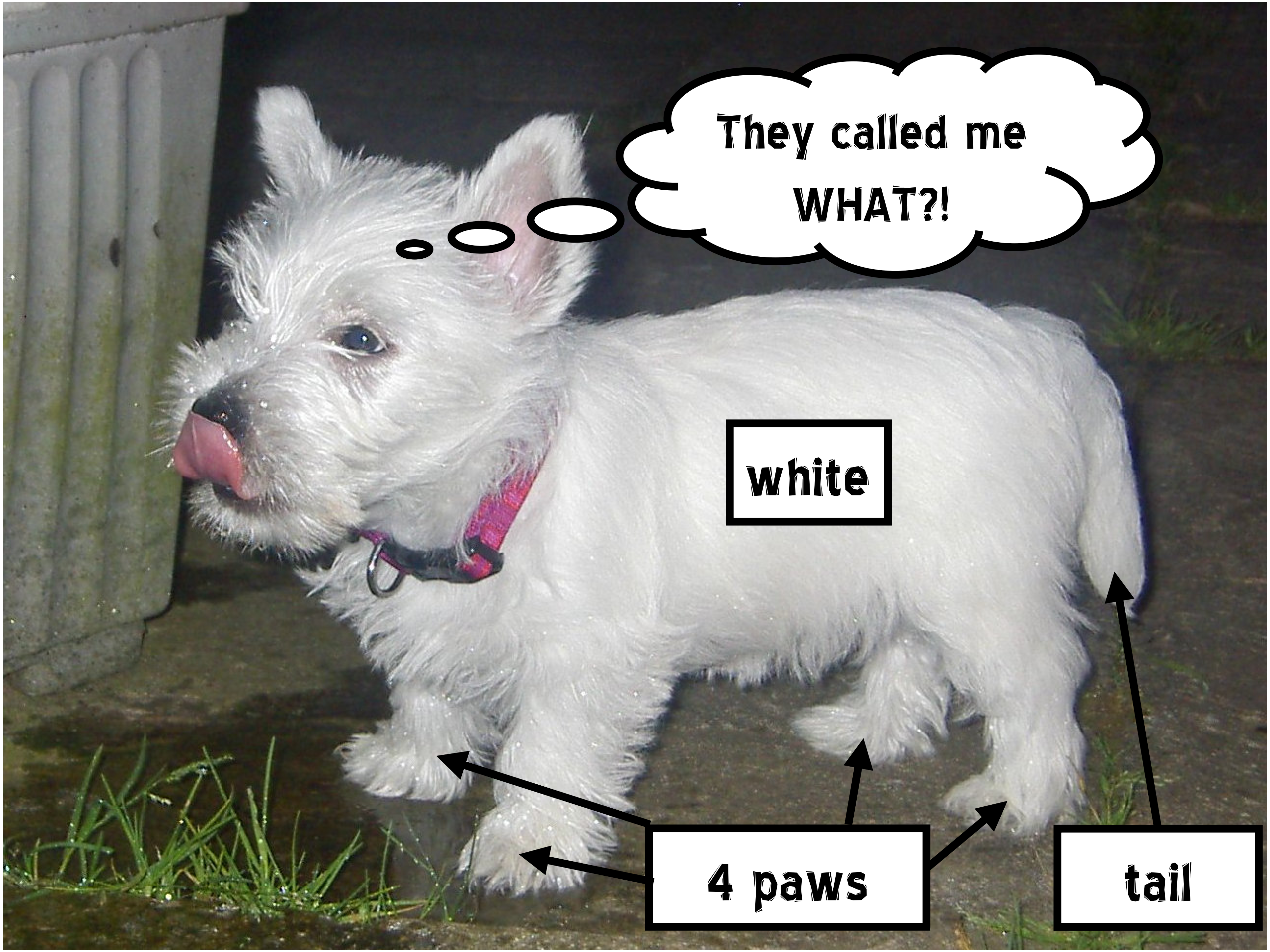
  - Or without to declare that the rest of the file describes the class.

```
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

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

Accessor Methods

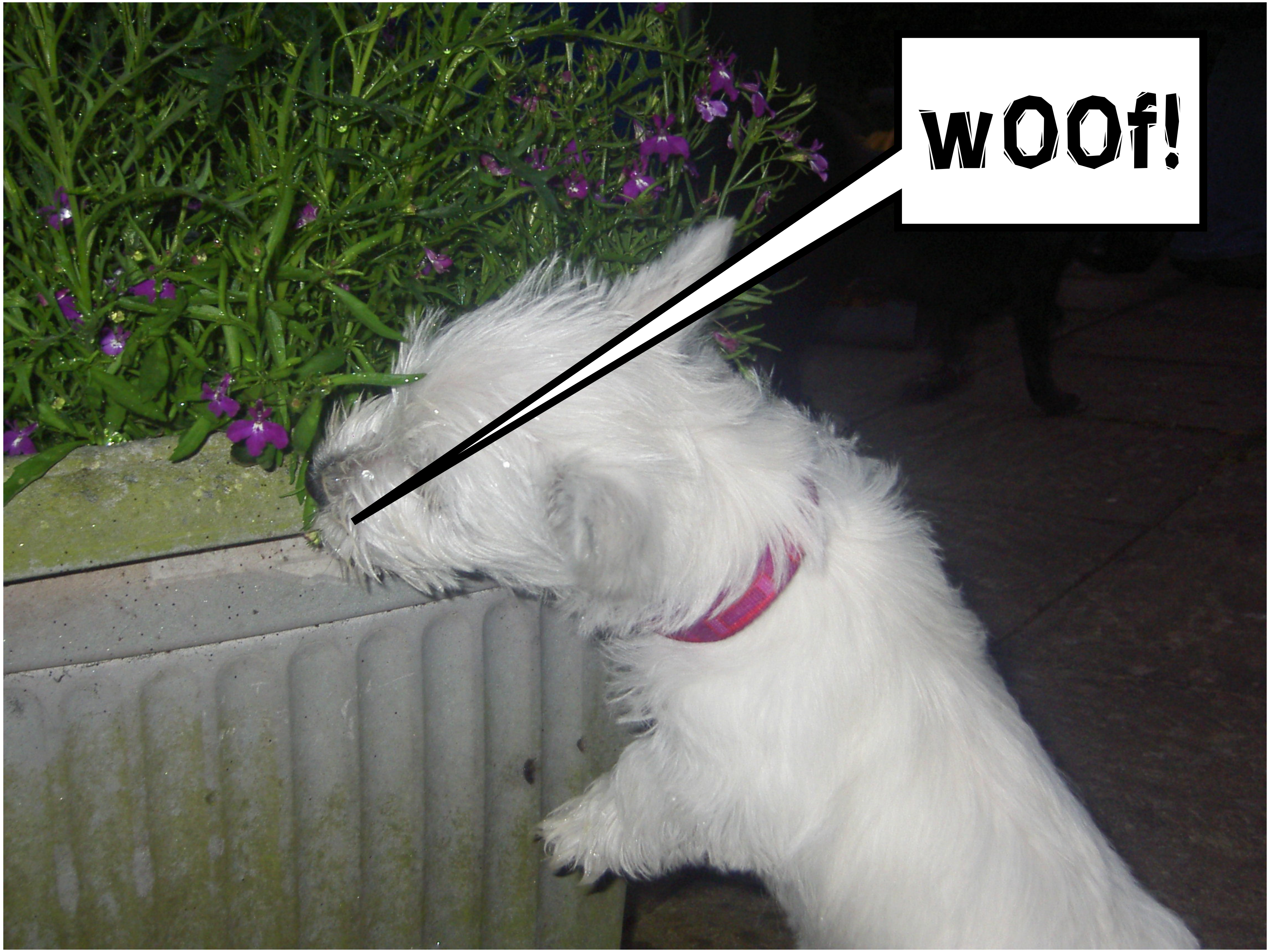
- We want to allow outside access to some of the attributes
- Writing accessor methods is boring!
- `$.` means it is automatically generated

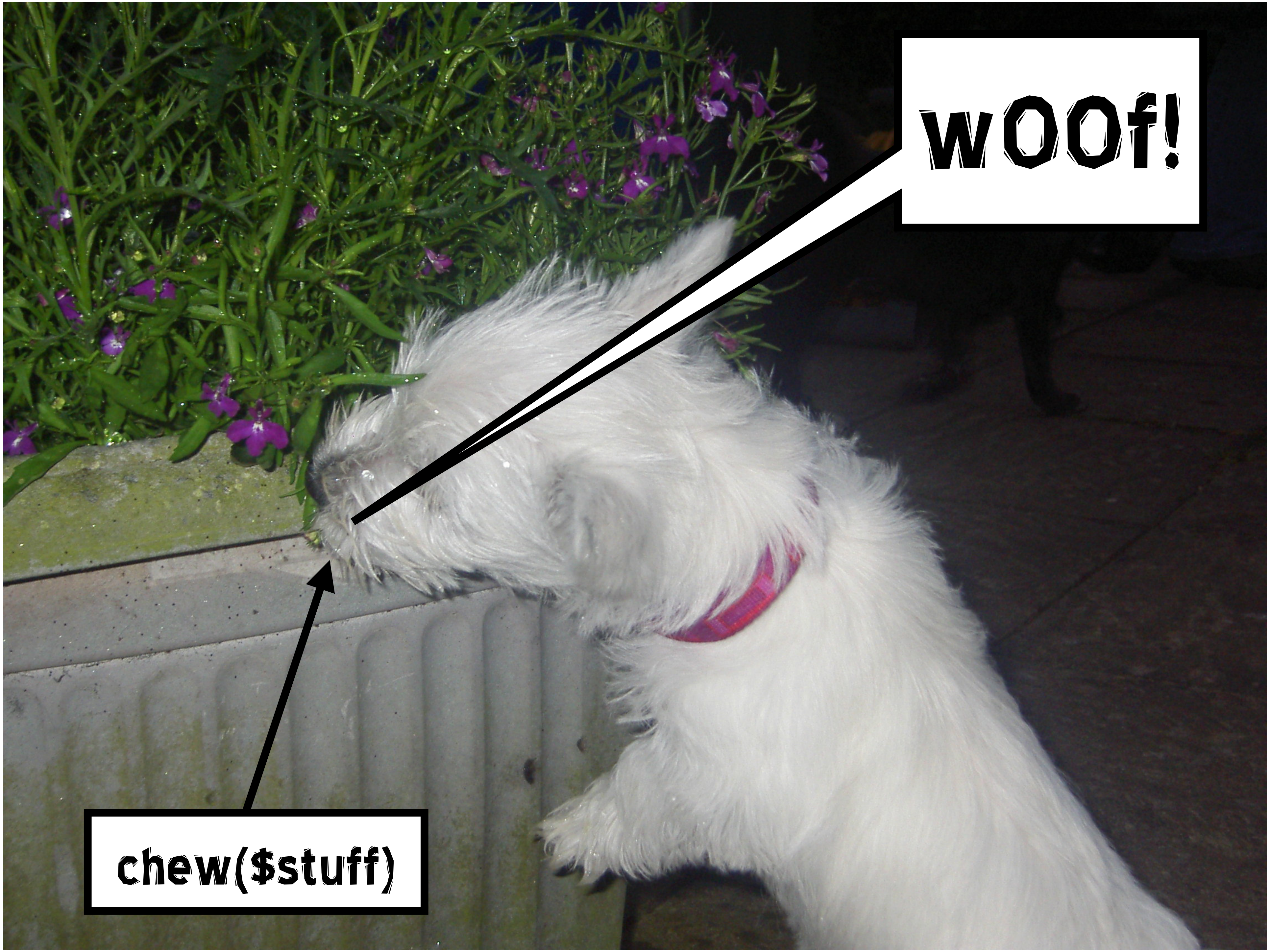
```
class Puppy {
    has $.name;
    has $.colour;
    has @paws;
    has $tail;
}
```

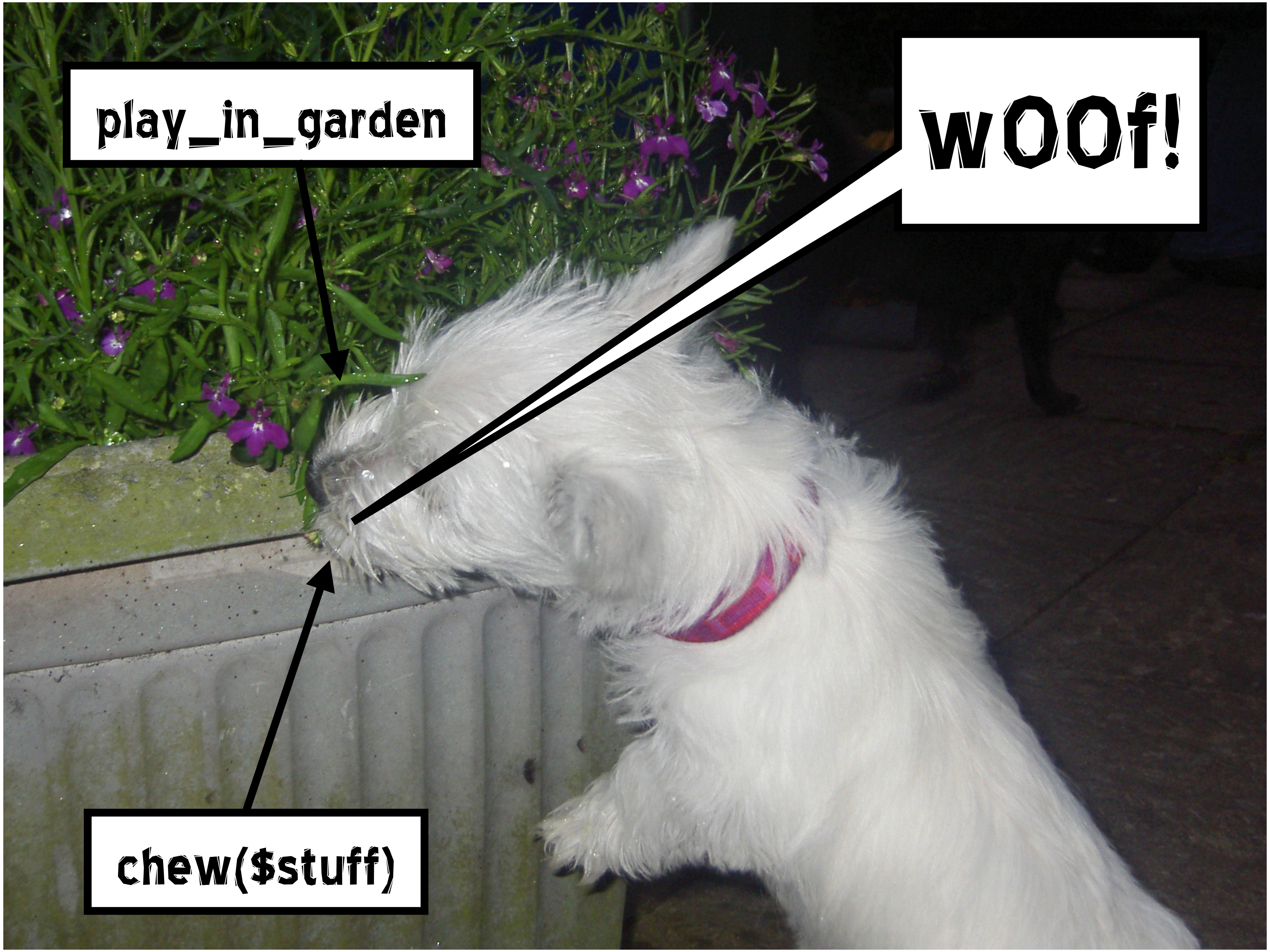
Mutator Methods

- 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)

Clarifying Roles

Methods

- The new **`method`** keyword is used to introduce a method

```
method bark() {
    say “w00f!”;
}
```

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

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

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';
}
```

Consuming 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
```

Clarifying Roles

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 {
     …
}
```

Clarifying Roles

Multiple Inheritance

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

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

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?

Clarifying Roles

The Java, C# Answer

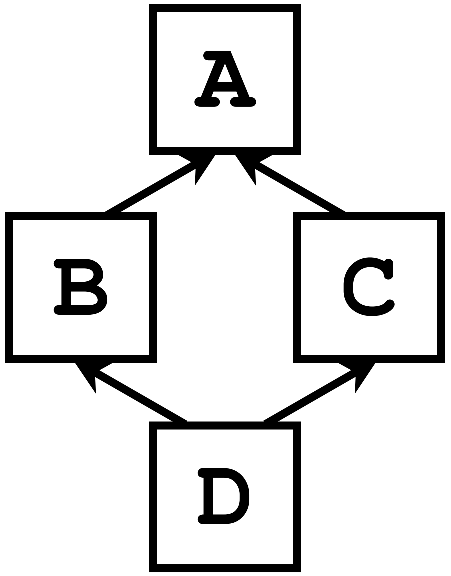
- There's only single inheritance
- You can write an interface, which specifies that a class must implement a **`walk`** method
- Write a separate class that implements the **`walk`** method
- You can use delegation (hand coded)
- Sucks

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…

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



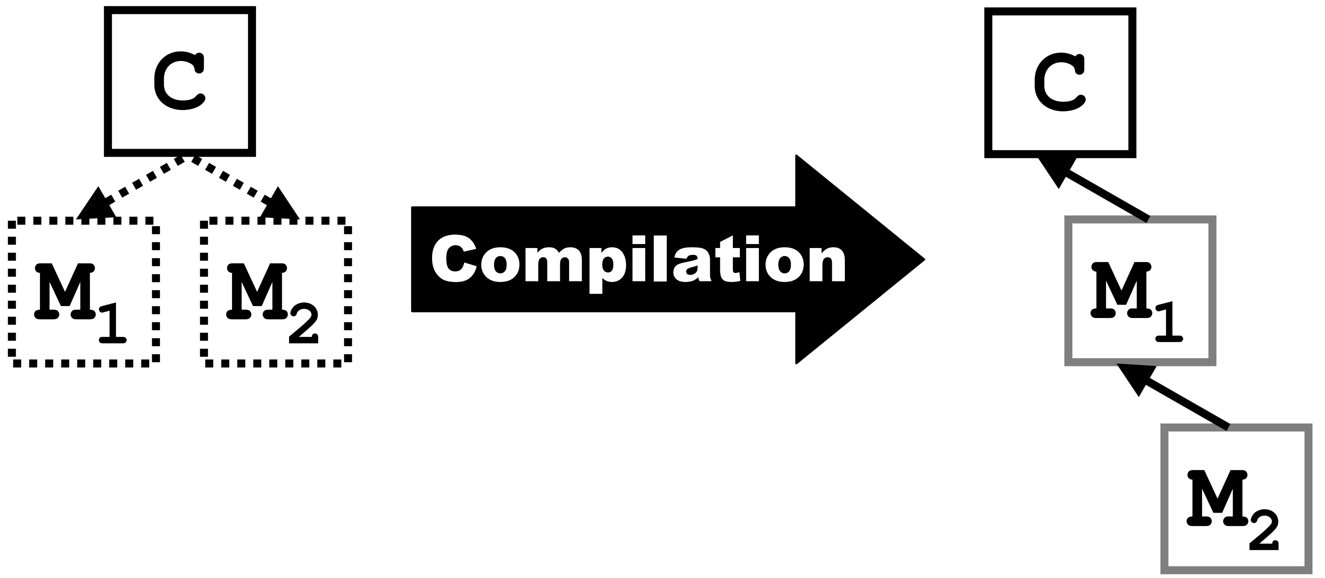
Clarifying Roles

Mix-ins

- A mix-in is a group of one or more methods than 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.

How Mix-ins Work

- Defined in terms of single inheritance



- C with M_1 and M_2 mixed in is, essentially, an anonymous subclass

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 class hierarchies 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

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

<u>Flattening Composition</u>

- 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

Clarifying Roles

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;
        }
    }
}
```

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…

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;
```

- Note: to use this currently in Pugs, you must use:

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

Parametric Polymorphism

- Polymorphism = code can work with values of different types
- Parametric = a type takes a parameter; we pass a type variable whenever we use the type
- What is the type of the invocant (self) for a method in a role?
  - That of the class we compose it into

Parametric Polymorphism

- The types of roles are therefore parametric
- They are parameterised on the type of the class that we compose the role into
  - Compose Walk into class Dog, the invocant has type Dog
  - Compose Walk into class Cat, the invocant has type Cat

Clarifying Roles

The End

Clarifying Roles

w00f!

Clarifying Roles

Questions?