Understanding Roles, Constraints And Classes

Jonathan Worthington
London Perl Workshop 2006
Perl 6 will be out for Christmas.
Understanding Roles, Constraints And Classes

I got Perl 6!

Socks. 😞
Not this Christmas.
Understanding Roles, Constraints And Classes

My feet are warm. 😊

Oh, it’s an IOU.
Perl 6

- The language has been designed from the ground up; the implementation has been built from scratch
- Language wise, not backward compatible (Perl 5 programs will not usually be valid Perl 6 programs), but:
  - Source code translator
  - `use perl5:Some::Module;`
Object Oriented Perl

- Many things have changed in Perl 6
- Object orientation is no exception
  - Nicer syntax
  - Attempts to provide 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 – more later!
Classes
Understanding Roles, Constraints And 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
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
Understanding Roles, Constraints And Classes

What Are Classes Used For?

- Providing a route to polymorphism
  - This means that the same code can safely operate of values of different types
  - Inheritance relationships state that one class can be used in place of another
  - Essentially, enables more re-use
Classes In Perl 6

- Introduce a class using the `class` keyword
  - With a block:
    ```perl
    class Puppy {
        ...
    }
    ```
  - Or without to declare that the rest of the file describes the class.
    ```perl
    class Puppy;
    ```
They called me WHAT?!
They called me WHAT?!

white

4 paws
They called me WHAT?!

white

4 paws
tail
Attributes

- Introduced using the `has` keyword

```perl
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

```java
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

```ruby
class Puppy {
    has $.name is rw;
    has $.colour;
    has @paws;
    has $tail;
}
```
chew($stuff)
woof!

chew($stuff)
wag_tail
play_in_garden

woof!

chew($stuff)

wag_tail
Methods

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

```perl
method bark() {
    say "woof!";
}
```

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

```perl
method chew($item) {
    $item.damage++;
}
```
Attributes In Methods

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

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

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

```ruby
method play_in_garden() {
  $colour = 'black';
}
```
Attributes In Methods

- If there is a conflict with a lexical variable, you can use `$!colour`.

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

- This is because all (private) attributes inside the class really have the `!` in their name; can use it to emphasize privateness.

```ruby
has $!tail;
```
Understanding Roles, Constraints And Classes

Consuming A Class

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

```perl
my $puppy = Puppy.new(
    name => 'Rosey',
    colour => 'white'
);
$puppy.bark(); # w00f!
say $puppy.colour; # white
$puppy.play_in_garden();
say $puppy.colour; # black
```
A Note On Instantiation

- Another common way to write the instantiation code is this

```perl
my Puppy $puppy .= new(
    name => 'Rosey',
    colour => 'white'
);
```

- The `.=` method means “call a method on myself and assign the result to me”
- `$puppy` is undefined, but we know its class, so can call the `new` method
Delegation

- Sometimes, one of the attributes contains a method that we want to expose in the current class; we could write a method like this:

```plaintext
method wag() {
    $tail.wag();
}
```

- Use delegation instead; modify the declaration of `$tail`

```plaintext
has $tail handles 'wag';
```
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 \texttt{is} keyword

```java
class Dog {
    ...
}
class Puppy is Dog {
    ...
}
```
Multiple Inheritance

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

```python
class Puppy is Dog is Pet {
    ...
}
```
Roles
In Perl 6, roles take on the main role of software 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?
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
Mix-ins

- A mix-in is a group of one or more methods that cannot 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
Understanding Roles, Constraints And Classes

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
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
Creating Roles

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

```perl
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

```java
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;
  ```
Additional Safety

- We want to be sure that when we compose our role, the items in @paws will have the step method.

- Assuming the Paw class has the step method, we can add a type annotation to the has declaration in both the role and the class, stating that elements of the array must be of the class Paw.

has Paw @paws;
Parametric Polymorphism

- Polymorphism = code can work with values of different types
- Parametric = a type has a type variable in that we replace with a type parameter
- What is the type of the invocant (self) for a method in a role?
  - That of the class we compose it into
Understanding Roles, Constraints And Classes

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.
Constraints
Refinement Types

- A type classifies a value
  - For example, 42 is an integer
- Therefore for each type there is a (possibly infinite) set of values that could be classified as that type
- Constrains are refinement types
  - Take an existing type
  - Restrict the values in it further
EvenInt

- An EvenInt will be a refinement of the Int type that can only hold even values
- Declare it using the `subset` keyword

```perl
my subset EvenInt of Int
  where { $^n % 2 == 0 };
```

- Variables with the secondary sigil `^` hold parameters that the block has been passed; the lexicographically first name gets the first parameter, etc.
Making Walk More General

- We may want to use the `Walk` role for humans too.
- Humans have feet, not paws.
- We’d like `@paws` to contain something that has the `step` method, but in reality it may contain `Foot` or `Paw` objects.
Making Walk More General

- Define a refinement type that requires the `step` method (Any = any type)

```
my subset Walkable of Any
  where { .can('step') };
```

- Use this in the `has` declaration in the class and the role

```
has Walkable @paws;
```
Review
Random Bits

# Refinement type for things that have the
# step method
my subset Walkable of Any
    where { .can('step') };

# A Paw class
class Paw {
    method step() {
        say "plod";
    }
}

The Walk Role

role Walk { 
    has Walkable @paws;

    method walk($num_steps) {
        for 1..$num_steps {
            .step for @!paws;
        }
    }
}
class Dog does Walk {
    has $.name is rw;
    has $.colour;
    has Walkable @paws = (Paw.new() xx 4);
    has $tail handles 'wag';

    method bark() {
        say "WOOF!";
    }
    method play_in_garden() {
        $colour = 'black';
    }
}
The Puppy Class

class Puppy is Dog {
    # Add a chew method.
    method chew($item) {
        $item.damage++;
    }

    # Override Dog’s bark method
    method bark() {
        say "w00f!";
    }
}

And Finally…

pugs> my Puppy $puppy .= new(
    name => 'Rosey',
    colour => 'white'
);

pugs> $puppy.walk(2);
plod
plod
plod
plod
plod
plod
plod
plod
plod
plod
The End
woof!
Questions?