Object Inheritance Without Classes

Timothy Jones

Victoria University of Wellington

tim@ecs.vuw.ac.nz

April 28, 2016

Inheritance

Code reuse

Relationships between classes

Inheritance

```
abstract class Graphic {
   var name := "A graphic"
   var canvas
   constructor(canvas) {
      self.canvas := canvas
      canvas.register(self)
      draw
   }
   abstract method image
   method draw {
      canvas.render(image)
   }
}
```

Inheritance

```
abstract class Graphic {
  var name := "A graphic"
  var canvas
  constructor(canvas) {
    self.canvas := canvas
    canvas.register(self)
    draw
  }
  abstract method image
  method draw {
    canvas.render(image)
  }
}
```

```
class Amelia extends Graphic {
  def image = images.amelia
  constructor {
    super(canvas)
    name := "Amelia"
  }
}

def amelia = new Amelia
```

Foundations

What is a class?

- A factory (constructs new objects)
- A type (classifies the constructed objects)

Foundations

What is a class?

- A factory (constructs new objects)
- A type (classifies the constructed objects)

Other solutions to typing (dynamic, structural)

We can implement factories with methods and objects

Foundations

What is a class?

- A factory (constructs new objects)
- A type (classifies the constructed objects)

Other solutions to typing (dynamic, structural)

We can implement factories with methods and objects

Are classes redundant?

Objects-First

Objects define their own state and behaviour

```
object {
  // Methods and initialisation code
  ...
}
```

Classes are constructors for objects with the same implementation

```
method make {
  object { ··· }
}
```

Inheritance

Code reuse

- Imperative rather than declarative
- *Implementation* reuse

Relationships between ~~classes~~ objects

In Other Languages

Self

```
( | parent* = factory new. | )
```

JavaScript

```
Bar.prototype = foo;
```

Lua, Emerald, Tcl, E, Kevo ...

Translating Classes

```
abstract class Graphic {
   var name := "A graphic"
   var canvas
   constructor(canvas) {
      self.canvas := canvas
      canvas.register(self)
      draw
   }
   abstract method image
   method draw {
      canvas.render(image)
   }
}
```

Translating Classes

```
method graphic(canvas) {
  object {
    var name := "A graphic"
    canvas.register(self)
    draw

    method image { abstract }
    method draw {
      canvas.render(image)
    }
  }
}
```

Translating Classes

```
method graphic(canvas) {
  object {
    var name := "A graphic"
    canvas.register(self)
    draw

    method image { abstract }
    method draw {
      canvas.render(image)
    }
  }
}
```

```
class Amelia extends Graphic {
  def image = images.amelia
  constructor {
    super(canvas)
    name := "Amelia"
  }
}

def amelia = new Amelia
```

Translating Classes

```
method graphic(canvas) {
  object {
    var name := "A graphic"
    canvas.register(self)
    draw

    method image { abstract }
    method draw {
      canvas.render(image)
    }
  }
}
```

```
def amelia = object {
  inherit graphic(canvas)
  def image = images.amelia
  name := "Amelia"
}
```

Semantics

What does this mean?

```
inherit graphic(canvas)
```

Do the inherit semantics actually allow us to implement classes?

- Let's investigate different object inheritance semanticses

Concerns

Considering these aspects:

- Registration
- Down-calls
- Action at a Distance
- Stability
- Preëxistence
- Multiplicity

Registration

```
method graphic(canvas) {
  object {
    var name := "A graphic"
    canvas.register( self )
    draw

    method image { abstract }
    method draw {
      canvas.render(image)
    }
  }
}
```

```
def amelia = object {
  inherit graphic(canvas)
  def image = images.amelia
  name := "Amelia"
}
```

Down-calls

```
method graphic(canvas) {
  object {
    var name := "A graphic"
    canvas.register(self)
    draw

    method image { abstract }
    method draw {
      canvas.render( image )
    }
  }
}
```

```
def amelia = object {
  inherit graphic(canvas)
  def image = images.amelia
  name := "Amelia"
}
```

Action at a Distance

```
method graphic(canvas) {
  object {
    var name := "A graphic"
    canvas.register(self)
    draw

    method image { abstract }
    method draw {
      canvas.render(image)
    }
  }
}
```

```
def amelia = object {
  inherit graphic(canvas)
  def image = images.amelia
  name := "Amelia"
}
```

Stability

```
method graphic(canvas) {
  object {
    var name := "A graphic"
    canvas.register(self)
    draw

    method image { abstract }
    method draw {
      canvas.render(image)
    }
  }
}
```

```
def amelia = object {
  inherit graphic(canvas)
  def image = images.amelia
  name := "Amelia"
}
```

Preëxistence

```
method graphic(canvas) {
  object {
    var name := "A graphic"
    canvas.register(self)
    draw

    method image { abstract }
    method draw {
      canvas.render(image)
    }
  }
}
```

```
def parent = graphic(canvas)

def amelia = object {
  inherit parent
  def image = images.amelia
  name := "Amelia"
}
```

Multiplicity

```
method graphic(canvas) {
  object {
    var name := "A graphic"
    canvas.register(self)
    draw

    method image { abstract }
    method draw {
      canvas.render(image)
    }
  }
}
```

```
def amelia = object {
  inherit graphic(canvas)
  inherit other
  def image = images.amelia
  name := "Amelia"
}
```

Object Inheritance

Objects inherit directly from one another

Three foundational models:

- Forwarding (as in E)
- Delegation (as in JavaScript and Self)
- Concatenation (as in Kevo)

Forwarding

Requests to inherited methods go directly to inherited object

- Simplest semantics
- No down-calls (cannot modify existing implementation)

```
method image { abstract }
method draw {
  canvas.render( image )
}
```

Delegation

Requests to inherited methods have self bound to original object

- The standard semantics of object inheritance
- Surprising behaviour if you're used to classes

```
def amelia = object {
  inherit parent
  def image = images.amelia
  name := "Amelia"
}
```

Concatenation

Copy the methods and fields from the inherited object

- Removes direct relationship between inheritor and inheritee
- Changes to inherited object are not reflected in inheriting object
- Potentially costly clone operation

Registration

None of the models support registration

```
canvas.register( self )
```

Emulating Classes

Objects inherit from calls to constructor methods

Two class-like models

- Merged Identity (as in C++)
- Uniform Identity (as in Java)

Merged Identity

Inheriting object 'becomes' the inherited object

- Registered identities *eventually* resolve to the intended object
- Objects not stable during construction

draw

```
method image { abstract }
method draw {
  canvas.render(image)
}
```

Uniform Identity

Inherited initialisation code runs as the inheriting object

Basically magic

Emulating Classes

Not very satisfactory as foundational models

Other languages (JavaScript, E) manage to do it nicely

Multiple Inheritance

Every model except merged identity

Various different conflict resolution schemes

Implementation

Formal description of each model's semantics

Runnable semantics with PLT Redex

Conclusion

No obviously superior semantics for object inheritance

Emulating classes requires magic or complicated language features

Ultimately depends on the design goals for the language