Exploring the Inheritance Design Space with Grace

Kim Bruce
Pomona College

Based on work by Tim Jones, Michael Homer, James Noble, & Andrew Black

WG2.16 meeting, 8/2017

In the Beginning

- In the 80's there were models of objects, classes and inheritance in a lovely functional garden with flowers, trees, and smiling researchers pondering the universe.
  - Players included Cardelli, Cook & Palsberg, Kamin, and Reddy.
- Later the sun shone more brightly as types were added to the models by Cook & the Abel group, and Mitchell & his group.

Approaching reality

- Different models with instance variables were proposed in early '90s:
  - Pierce & Turner, Bruce, Abadi & Cardelli, Fisher & Mitchell, Featherweight Java
  - Typically based on existential quantifiers & various numbers of fixed points
  - ... assignment came later ...

Lying in State

- Virtually all "real" OO languages are imperative.
- State provides added expressiveness, but makes everything harder
- Initialization (constructors) is big problem
  - Object can be visible while being initialized.
- Eventually lots of models, but larger design space ...

How does state impact the design of inheritance in OO languages?

Entering into *Grace*

- Grace: Object-based language aimed at teaching novices.
  - Everything is an object
    - Classes are definable: *methods returning objects*
    - Simple dynamic method dispatch
  - Simple, uniform syntax
  - Correspondingly simple semantics
  - Optionally typed
  - Blocks as first-class closures

Objects

```
def mySquare = object {

      var side := 10

      method area {
            side * side
      }

      method stretchBy(n) {
            side := side + n
      }
}
```

Classes

- ... generate objects:

```
class aSquareWithSide (s: Number) -> Square {
      var side: Number := s

      method area -> Number {
          side * side
      }

      method stretchBy (n: Number) -> Done {
           side := side + n
      }

      print "Created square with side {s}"
}
```

No separate constructors.
Type annotations can be omitted or included

Classes

- ... really methods returning fresh objects:

```
method aSquareWithSide (s: Number) -> Square {
    object {
        var side: Number := s

        method area -> Number {
            side * side
        }

        method stretchBy (n: Number) -> Done {
             side := side + n
        }

        print "Created square with side {s}”
    }
}
```

Extending Objects/Classes

- Notion of modifying and extending existing definitions pervasive in OO programming
  - But mechanisms are different
- What should we use for Grace?
- Focus of rest of talk

Objects vs. Classes

- Which is primitive?
- How to define extension?
  - object-based ⇒ delegation (*Abadi/Cardelli,Mitchell/Fisher*)
  - class-based ⇒ inheritance (*Cook et al, Bruce, Pierce et al, etc*)

Example

```
class graphic {
    method image { required }
    method draw { canvas.render(image) }
    var name := "A graphic"
    displayList.register(self)
    draw
    print (name)
}

def amelia = object {
    inherit graphic
    method image is override { images.amelia }
    self.name := "Amelia"
}
```

what image used?

what gets registered?

what is drawn?

which name is printed?

What happens when amelia created & invoke amelia.draw?

Inheritance Design Space

- Focus on variations in order/timing of
  - Creation (allocation)
  - Initialization
- ... when defining subobject from super

Issues

- Registration:
  - Does identity of object change during construction?
  - What is effect of the register method in superclass?
- Down-calls:
  - Can a method request in superclass invoke a method in subclass? What about during construction?
- Change at a distance:
  - Can ops on one object implicitly change another?

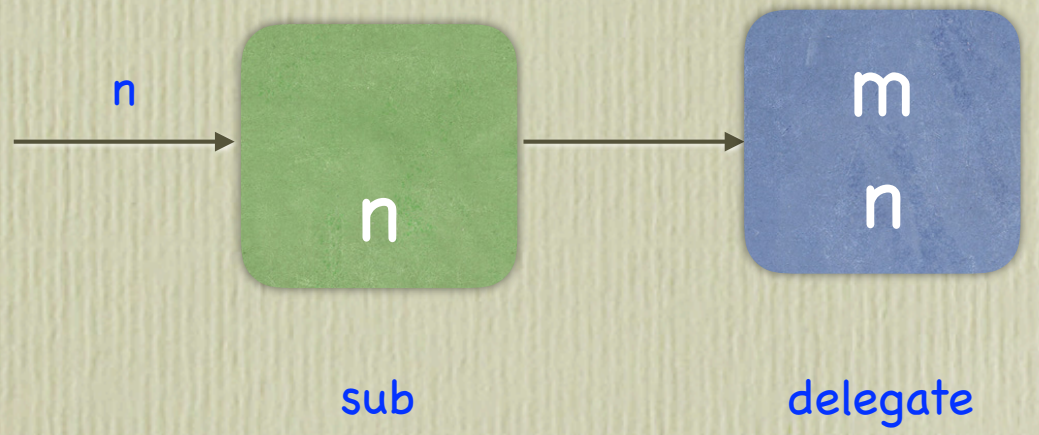
More Issues

- Pre-existence
  - Can an object inherit from an existing object?
- Stability
  - Is the implementation of methods the same through an object's lifetime? I.e., what happens between execute super constructor and use in sub-object?
- Simplicity
  - Easy to explain — but lead to common mechanisms

Inheritance models

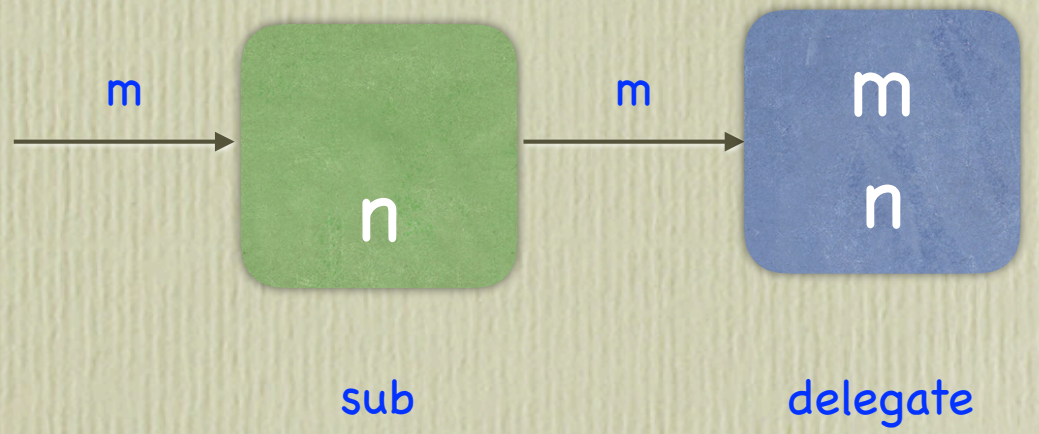
- Objects:
  - Delegation
  - Concatenation
- Emulating Classes:
  - Merged identity
  - Uniform identity

Delegation



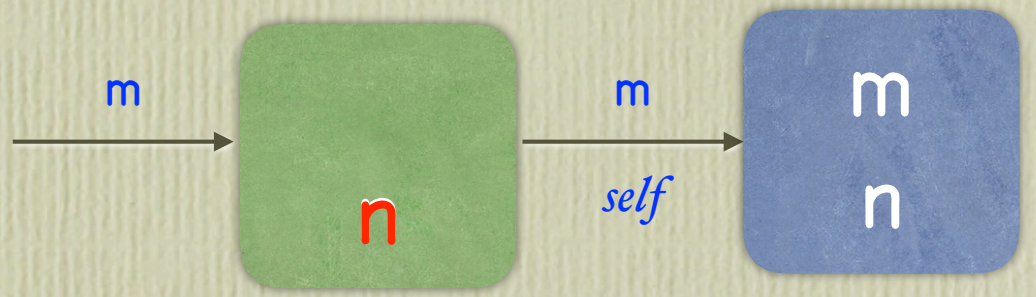
sub.n handled locally ...

Delegation



sub.m goes to delegate ...

Delegation



which n?

What if delegate has: method m {... *self*.n ...}
Invoke: sub.m

use *self* from sub ...

Forwarding doesn't update self...

Example

```
def graphic = object {
    method image { required }

    method draw { canvas.render(image) } – downcall fine

    var name := “A graphic”

    displayList.register(self) – fails to register amelia
    draw                – crashes
    print (name)        – prints “A graphic”
}

def amelia = object {
    inherits graphic

    method image is override { images.amelia } – not stable

    self.name := “Amelia"  – changes value in graphic
}                                object
```

Delegation

- Inherited methods redirected to super-object
  - But, value of self in inherited method reset to subobject.
  - Down-calls fine after construction, but *not* during initialization
    - Superobject initialized before subobject created
    - Registration in superclass will not work for sub-object
  - Not stable structure before & after construction
  - Can inherit from existing object
    - But mutation to inherited field visible to other inheritors (shared)
- Example:
  - Self, Lua, & Javascript (*but no action at distance*)

Concatenation

- Allocate space for new
  - including features of old
- Shallow clone super into new space
  - Initializes
- Add new methods and instance variables
- Run initialization of sub

Example

```
def graphic = object {
    method image { required }

    method draw { canvas.render(image) } – downcall fine

    var name := “A graphic”

    displayList.register(self) – fails to register amelia
    draw                  – downcall fails in constructor
    print (name)  – prints “A graphic” when create amelia
}

def amelia = object {
    inherits graphic

    method image is override { images.amelia } – not stable

    self.name := “Amelia"                    – updates fine
}
```

Concatenation

- Make shallow copy, then add changes to front
  - Execute first version of method found
  - Like delegating to shallow clone of super.
  - Down calls only after construction over
  - Registration fails on sub-object
    - Super-object initialized before cloning
  - No effect at distance on super state because cloned
  - All objects must be (shallow) cloneable to be inherited from
- Example:
  - Kevo, can implement in Javascript

Emulating Class Inheritance

Merged Identity

- Allocate space for new
  - including features of old

- Initialize super in new space
- Add & override methods and instance variables from sub
- Run initialization of sub

Example

```
class graphic {
    method image { required }

    method draw { canvas.render(image) } – downcall fine

    var name := “A graphic”

    displayList.register(self) – registers amelia
              – though visible as graphic initially
    draw          – down call fails
    print (name)  – prints “A graphic” when create amelia
}

def amelia = object {
    inherits graphic

    method image is override { images.amelia } – not stable

    self.name := "Amelia"
}
```

Merged Identity

- Parent object constructed & initialized, mutated to child at the point of inheritance.
  - Must inherit from fresh object
  - Down-calls will work only after construction over
    - Self changes only after initialization of super.
    - Overridden methods stay accessible with super
    - Methods not stable during initialization
  - Registration fine (identity stable)
- Example: C++

Uniform Identity

- Allocate space for new
  - including features of old
- Add methods from super
- Add & override methods from sub
- Run initialization of super
- Run initialization of sub

Same (new) self!!

Example

```
class graphic {
    method image { required }

    method draw { canvas.render(image) } – correct downcall

    var name := “A graphic”

    displayList.register(self) – registers amelia
    draw                – correctly uses overridden image
    print (name)  – prints “A graphic” when create amelia
}

def amelia = object {
    inherits graphic

    method image is override { images.amelia }

    self.name := "Amelia"
}
```

Uniform Identity

- Allocate structure for full sub-object, with new/revised methods. Then initialize top down.
  - Must inherit from fresh object
  - Down-calls fine during construction & later
  - Stable, though may observe uninitialized fields.
  - Registration fine (identity stable)
- Like Java, C#

Comparison

	Registration	Downcall	Distance	Super-object can exist?	Stable
Delegation	no	no*	yes	existing	no
Concatenation	no	no*	no	existing	no
Merged	yes	no*	no	fresh	no*
Uniform	yes	yes	no	fresh	yes

* = change after constructor

Which to Choose?

- Delegation & Concatenation both reasonable
  - Except wanted registration and down-calls to work.
  - Concatenation requirement for shallow clone problematic.
  - Delegation action at distance may be confusing to novices.
- Uniform identity attractive
  - supports registration, down-calls, stability.
  - Requirement for fresh objects limiting.

Multiple Inheritance

- Even more complex
- Decided to use traits
  - Restricted to no explicit state
    - Avoids issues with initialization
  - Can inherit one superclass, use many traits
  - Can exclude methods from parent
    - Forced to resolve conflicts
  - Alias inherited methods to get effect of super

Traits

```
class catfish {
   use cat
      alias catSpeak = speak
   use fish
      alias fishSpeak = speak
   method speak {
      catSpeak
      fishSpeak
   }
}
```

Summary

- Looked at how features of inheritance useful in examining how to do reuse.
  - ECOOP 2016 paper provides formal semantics.
  - JOT paper "Grace's Inheritance" this spring
  - Complex, but provides insights.
- Capturing classical inheritance has challenges in object-based languages.

Whither Grace

- Settled (so far) on uniform identity plus traits.
  - though not everyone happy.
  - advantages from earlier slide.
  - ... and similarity to existing languages.
- In practice, inheritance from classes straightforward.
  - but limited from objects.
  - requires planned reuse.

Grace

- More info (including language spec) available at gracelang.org
- Text, objectdraw graphics library, and other teaching materials available.

Grace in Action

- Used three times in introductory courses at Pomona.
- Used four times at Portland State at variety of levels.
- Very successful for introducing novices to OO programming.

Questions?