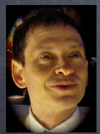
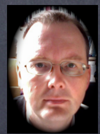
Grace

A New Educational Object-Oriented Programming Language



Andrew Black

Kim Bruce



James Noble

gracelang.org

Grace User Model

- First year students in OO CS1 or CS2
  - objects early or late,
  - static or dynamic types,
  - functionals first or scriptings first or ...
- Second year students
- Faculty & TAs — assignments and libraries
- Researchers wanting an experimental vehicle
- Language Designers wanting a good example

Grace Example

```
method average(in : InputStream) -> Number
// reads numbers from in stream and averages them
{  var total := 0
   var count := 0
   while { ! in.atEnd } do {
      count := count + 1
      total := total + in.readNumber }
   if (count = 0) then {return 0}
   return total / count }
```

Any questions?

Method Requests

```
aPerson.printOn(outputStream)

printOn(outputStream)   // implicit self

((x + y) > z) && !q        // operators are methods

while { ! in.atEnd } do { print (in.readNumber) }
                           // multi-part method name
```

λ-Blocks

- for (1..10) do // multi-part method name
  { i : Number -> print(i) }

```
const welcomeAction := { print "Hello" }
```

```
welcomeAction.apply
```

```
object { method apply
                 { print "Hello" } }
```

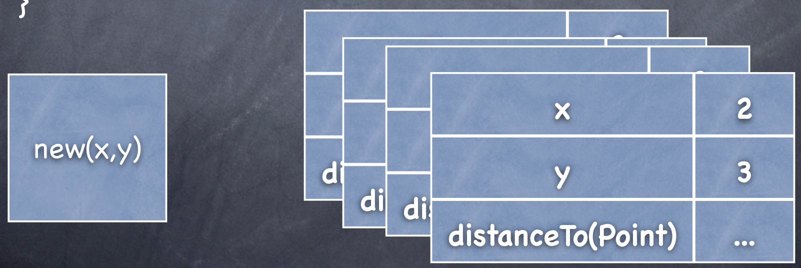

Object constructors

```
object {
   def x : Number = 2
   def y : Number = 3
   method distanceTo(other : Point) -> Number {
      ((x - other.x)^2 + (y - other.y)^2) }
}
```

x	2
y	3
distanceTo(Point)	...

Classes

```
class CartesianPoint.new(x': Number, y': Number) {
   def x : Number = x'
   def y : Number = y'
   method distanceTo(other : Point) -> Number {
      ((x - other.x)^2 + (y - other.y)^2) }
}
```



Classes

```
def CartesianPoint = object {
   method new (x': Number, y' : Number) {
      return object {
         def x : Number = x'
         def y : Number = y'
         method distanceTo(other:Point)->Number {
              ((x - other.x)^2 + (y - other.y)^2) }
      }
   }
}
```

Object Nesting

```
class SuperClass.new {
   def m := "in superclass. "
}


def out := object {
  def m := "in enclosing object. "
  def inner := object extends SuperClass {
    method foo { print (m) }
   }
}   // how to resolve lexical vs dynamic binding?
```



```
class SuperClass.new {
 function f { "function in superclass. " }
 method m    { "method in superclass. " }
}

def out := object {
 function f { "function in enclosing object. " }
 method m    { "method in enclosing object. " }

 def inner := object extends SuperClass {

   method test {
      f          // prints "function in enclosing object."
      self.f    // no such method error
      m         // no such function error
      self.m    // prints "method in superclass."
   }
 }
```

GedankenSprache

Types

- Types describe objects
  - Structural, Gradual, Optional

```
type Point = {
   x -> Number
   y -> Number
   distanceTo (other:Point) -> Number
}
```

- Types are sets of method request signatures
- Reified Generics

Type Algebra

- Variants: Point | nil, ?Point, Leaf<X> | Node<X>
  - $x : (A \mid B) \equiv x : A \vee x : B$
- Algebraic constructors:
  - T1 & T2: intersection, conforms to T1 and T2
  - T3 + T4: union, conforms to T3 or T4
  - T5 - T6: structural subtraction, T5 without T6
- Generics — no variance annotations needed!

lisp-1 vs lisp-3

```
var gerald : Person := Person.new("Gerald")

 // Person as a type
 // Person as a class

class CountedDispenser.new< T > {
   var count : Number
   method new() -> T {
       count := count + 1
       return T.new();
   }
}
```


lisp-1 vs lisp-3

```
var gerald : PersonT := PersonC.new("Gerald")

 // PersonT as a type
 // PersonC as a class

class CountedDispenser.new< T >(f : {new -> T}) {
   var count : Number
   method new -> T {
       count := count + 1
       return f.new();
   }
}
```


Match / Case

```
match ( x )                    // x : 0 | String | Student

 // match against a literal constant or singleton object
   case { 0 -> print("Zero") }

 // typematch, binding a variable
   case { s : String -> print(s) }

 // destructuring match, binding variables ...
   case { Student(name, id) -> print (name) }
```


Match / Case

```
type Pattern<T> = {
   try(Any) -> FailedMatch | SuccessfulMatch<T>
}

method filter ( input : List, pat : Pattern ) -> List {
   def output = List.new
   for (input) do { i ->
     match ( x )
        case { pat -> output.add(x) }
        case { _ ->  }
} }
```

Match / Case

```
match ( x )
    case { true | false -> print("Either / Or") }
    case { true || false -> print("True!") }
    case { #true | #false -> print("Either / Or") }

match ( true )
    case { foo && bar -> print("Both foo & bar") }
    case { foo -> print("Just Foo") }
    case { bar -> print("Just Bar") }
    case { _ -> print("Neither") }
```


Schedule

- 2011: 0.1, 0.2 and 0.5 language releases, hopefully prototype implementations
  - 3 implementations in progress
- 2012 0.8 language spec, mostly complete implementations
- 2013 0.9 language spec, reference implementation, experimental classroom use
- 2014 1.0 language spec, robust implementations, textbooks, initial adopters for CS1/CS2
- 2015 ready for general adoption?

http://gracelang.org