Grace

A New Educational Object-Oriented Programming Language

Andrew Black

gracelang.org

Kim Bruce

James Noble

Target Users

- First year students in OO CS1 or CS2
  - objects early or late,
  - static or dynamic typing,
  - functionals first or scripting first or …
- Second year students
- Faculty & TAs — assignments and libraries

High Level Goal

- Integrate proven newer ideas in programming languages into a simple language for teaching
  - with features that cleanly represent key concepts
  - so that students can focus on the essential, rather than accidental, complexities of programming and modelling.

Design Principles

- Low overhead for simple programs
- Simple semantic model that encourages thinking about the program
- Optional and gradual typing, including solid generics
- Power of functional constructs
- Support for immutables
- High level constructs for concurrency/parallelism
- Assertions, traces and tools for finding contradictions

Warning!

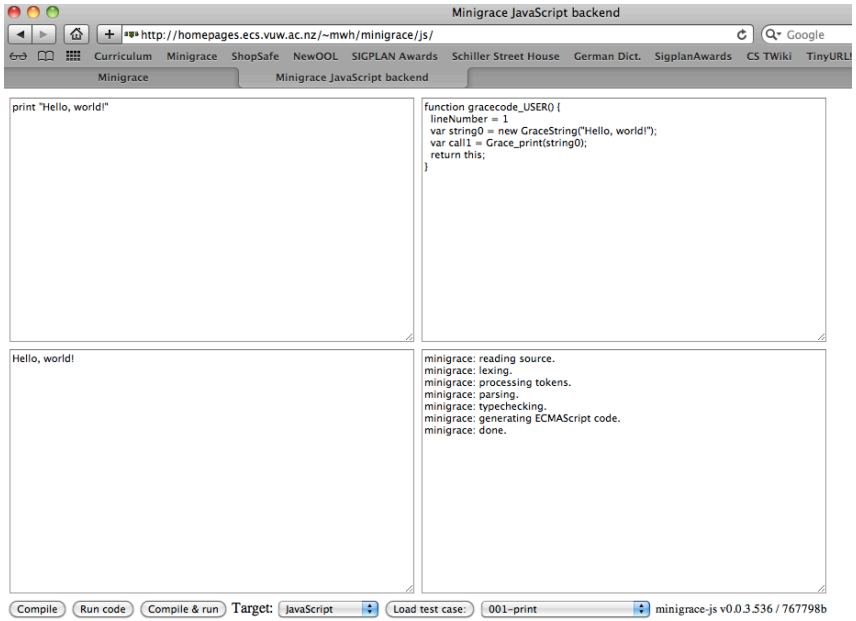
- Design is ongoing
  - You can still influence the design!
- Ambitious goals
- Still disagree on details
- We're not looking for innovative features, but for innovative combination of features to help novices learn to program.

Grace Fundamentals

- Everything is an object
- Simple method dispatch
- Single inheritance
- Types are interfaces (classes ≠ types)
- Blocks are first-class closures
- Extensible via Libraries (control & data)

Grace Fundamentals

- Language should be familiar
  - Java / C / Python / Eiffel / Scala programmers should be able to read Grace programs and recognize concepts
- Language levels for teaching



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?

Numbers

- Numbers are either
  - rational (exact) or
  - irrational (approximate)
- (10/3) * 6 = 20
- All numeric literals denote rational numbers

One true "method request"

- Like Smalltalk and Self:
  - no overloading
  - "method request" names the method and provides the arguments
  - "dynamic dispatch" selects the correspondingly-named method in the receiver
  - "method execution" occurs in the receiver

(We're trying to learn not to say "message-send" or "method call".)

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-expressions

- "Lambdas are relegated to relative obscurity until Java makes them popular by not having them." James Iry
- Grace has λs. We call them blocks:

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

Blocks

- Blocks are represented as objects
  - resulting object has an apply method
  - like Smalltalk, but with {->} and apply

```
def orderingFunction = { a, b -> a.name ≤ b.name }
```

```
if orderingFunction.apply(x, y) then { ... }
```

```
object { method apply(a, b) {
    a.name ≤ b.name } }
```


Libraries can define control

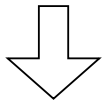
```
def Grace = object {
          // outermost enclosing object
          // methods requested on implicit self

  method if (c) then (t : Block) else (f : Block) {
      c.ifTrue ( t ) else ( f ) }

  method while (c : Block) do (a : Block) {
      c.apply.ifTrue( { a.apply; while (c) do (a) } )
  ...
}
```


Object constructors

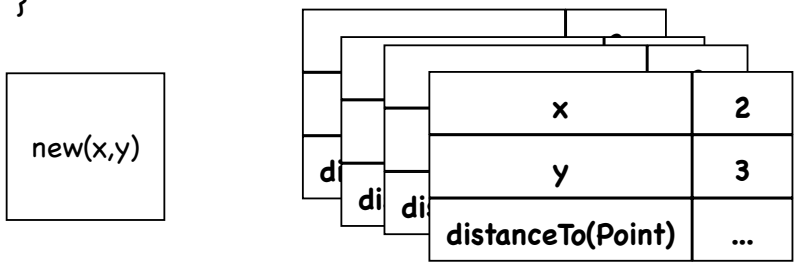
```
object {
  def x : Number = 2
  def y : Number = 3
  method distanceTo(other : APoint) -> Number {
    ((x - other.x)^2 + (y - other.y)^2) }
}
```



x	**2**
y	**3**
distanceTo(Point)	**...**

Classes

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



Classes

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


Class: Summary

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

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


Classes are not for Classificaiton

- Classes are an implementation concept
- Inheritance via object extension
- Classes are not types
  - Clases don't even play at being types on TV

Types

- Types are for classification
  - Structural, Gradual, Optional

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

- Types are sets of method request signatures
- Reified Generics

No null pointer exceptions!

Type Operations

- Algebraic constructors:
  - T1 & T2: union of methods in T1 and T2
  - T3 + T4: intersection of methods in T3 and T4
  - T5 - T6: every method in T5 but not in T6
- Variants: Point | nil, ?Point, Leaf<X> | Node<X>
  - x : (A | B) ≡ x : A ∨ x : B
- Generics — no variance annotations needed!

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) }
```

Object Nesting

- Object Nesting (gBeta, Newspeak)
  - nesting defines a **dialect:**
    - object has access to surrounding definitions
    - program written in multiple dialects
      - typed libraries written by instructors
      - untyped code written by students
- Language levels remove features for teaching

Asynchrony & Parallelism

- Hypothesis: we don't know what to do about parallelism!
- Conclusion: we must support different "models"
  - Software Transactional Memory (Clojure)
  - Actors (Scala, Akka, Erlang)
  - Locks (Java)
  - Atomic Sets
  - ...

Why Consider Using Grace?

- Clean Syntax
- Simple uniform model
  - no static features, no overloading, no null, etc.
  - Everything is an object (even lamdbas)
- Modern features
  - Generics done right, closures, pattern matching
  - Syntax supporting design of own control structures

Why Consider Using Grace?

- Easy transition between dynamic & static type-checking
- High level support for parallelism and concurrency (planned)
  - Likely adopt concurrency constructs similar to those in Habanero Java at Rice:
    - async{stmts}, finish {stmts}, future f := async{...}, forall(...) {stmts}, isolated{stmts}
  - Support for immutable objects

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?

Help!

- Supporters
- Programmers
- Implementers
- Library Writers
- IDE Developers!!!!
- Testers
- Teachers
- Students
- Tech Writers
- Textbook Authors
- Blog editors
- Community Builders

http://gracelang.org