Grace

An Object-Oriented Language for Novices

Andrew Black
Portland State U

Kim Bruce
Pomona College

James Noble
Victoria U. Wellington
Why?
Assumption: Programming Languages Matter

• You are going to teach object-oriented programming to 1st year students.

• Following Aristotle (and Brooks):
  ◦ What are the essential difficulties you must teach?
  ◦ What are the accidental difficulties imposed by the language you choose?

• How will you and your students divide your time?
Which language?

- **ECOOP 2010**: we don’t like the available options
  - Java, Scala, C++, C# and other “professional” languages — too complex for teaching
  - Smalltalk — no static types
  - Python — inconsistent method syntax, no encapsulation, “accidental” declarations ...

- All available options emphasize the *accidental*

- Group decision: design a modern object-oriented language specifically for teaching
Java has, but Grace does not:

1. Type-based overloading of methods.
2. `null`
3. Primitive data — `int`, `boolean`, `char`, `byte`, `short`, `long`, `float`, `double`.
4. Classes (as built-in non-objects).
5. Packages (as built-in non-objects).
6. Constructors (as distinct from methods) and `new`.
7. Object initializers (code in a class enclosed in `{ and }`)
8. `import *` — introduction of names invisibly.
9. Operations on variables, like `x++` meaning `x := x + 1`.
10. Multiple numeric types (so that, for example, `3.0` and `3` are different).
11. Numeric literals with `F` and `L`.
12. Integer arithmetic defined to wrap.
13. `==` as a built-in operation on objects.
14 static variables.
15 static methods.
16 static initializers.
17 final.
18 private (which is much more complicated than most people realize, since it interacts with the type system).
19 C-style for loops.
20 switch statements.
21 Class-types.
22 Packages
23 Package-based visibility.
24 Arrays (as a special built-in construct with their own special syntax and type rules).
25 Required semicolons.
26 () in method requests that take no parameters.
27 public static void main(String[] args) — necessary to run your code.
28 Object with “functional interfaces” treated as λ-expressions.
Grace has:

1. multi-part method names if(\_\_\_)then(\_\_\_)else(\_\_\_)
2. String interpolation: "The value of x is \{x\}"
3. Object expressions
4. Nested objects (lexical scope)
5. Closures w/correct scope
6. Operators defined as methods
7. match(\_\_\_)case(\_\_\_)... statement for examining variant types
Best of 20th Century-Technology

- Closures
- Assertions, unit testing, traces, and tools for finding errors
- High-level constructs for concurrency
- Support for immutable data
- Parameterized types (done right)
  
  e.g., List[String]
Talk Outline

• Meta-babble
• Quick Overview, terminology
• Objects and methods
• Classes
• $\lambda$-expressions
• Program and module structure

• Dialects
• Types
• Pattern-matching
• Exceptions
• Concurrency
• Teaching with Grace
• Dialects
Grace Fundamentals

- Everything is an object
- Simple method dispatch
- Single inheritance
- Types are interfaces (classes ≠ types)
  - Pedagogically, types come after objects
- Blocks: { syntax for \( \lambda \)-expressions }
- Extensible via libraries (control & data)
method average(in : InputStream) → Number {
    // reads numbers from 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 }
    total / count
}
method average(in : InputStream) → Number {
    // reads numbers from 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 }
    total / count
}

What questions do you have?
One true “method request”

• Like Smalltalk and Self:
  ° no static overloading
• a “method request” names the target, the method, and provides the arguments
• “dynamic dispatch” selects the correspondingly-named method in the receiver
• “method execution” occurs in the receiver

(We’re learning not to say “message-send” or “method call”.)
Method Requests

aPerson.printOn(outputStream)

printOn(outputStream) // implicit receiver

((x + y) > z) && q.not // operators are methods

while {! in.atEnd } do { print (in.readNumber) } // multi-part method name
Constructing Objects
Object constructors

object {
  def x : Number is public = 2
  def y : Number is public = 3
  method distanceTo(other : Point) → Number {
    ((x - other.x)^2 + (y - other.y)^2)^(1/2)
  }
}
object {
    def x : Number is public = 2
    def y : Number is public = 3
    method distanceTo(other : Point) → Number {
        ((x - other.x)^2 + (y - other.y)^2)^(1/2) 
    }
}
class x(x': Number)y(y': Number) → Point {
def x : Number is public = x’
def y : Number is public = y’
method distanceTo(other : Point) → Number {
    ((x - other.x)^2 + (y - other.y)^2)^(1/2) 
}
}
class `x(x': Number)y(y': Number) → Point` {
  `def x : Number is public = x'`
  `def y : Number is public = y'`
  `method distanceTo(other : Point) → Number` {
    `((x - other.x)^2 + (y - other.y)^2)^(1/2)`
  }
}`
Classes

- A Class is a shorthand for a factory method: a method that returns the result of an object constructor.

```plaintext
method x(x': Number)y(y': Number) \rightarrow Point {
    return object {
        def x : Number is public = x'
        def y : Number is public = y'
        method distanceTo(other:Point) \rightarrow Number {
            ((x - other.x)^2 + (y - other.y)^2)^(1/2)
        }
    }
}
```
class x(x')y(y') {
    def x is public = x'
    def y is public = y'
    method distanceTo other → {
        ((x - other.x)^2 + (y - other.y)^2)^(1/2)
    }
}

method x(x')y(y') {
    return object {
        def x is public = x'
        def y is public = y'
        method distanceTo(other) → {
            ((x - other.x)^2 + (y - other.y)^2)^(1/2)
        }
    }
}
Inheritance

class x(x: Number) y(y: Number) colour(c: Colour) {
    inherit cartesianPoint.x(x)y(y)
    def color : Colour is public = c
}

- Objects created by x(_)y(_)colour(_) have:
  - all the methods of a CartesianPoint.x(_)y(_), plus
  - methods colour and colour:=(_)
Uniform reference to attributes
Uniform reference to attributes

deObject.x
Uniform reference to attributes

theObject.x

// could be a request of a method, or access
// to a public variable: theObject knows which
Uniform reference to attributes

theObject.x

// could be a request of a method, or access
// to a public variable: theObject knows which

var x:Number := 3   // confidential variable
Uniform reference to attributes

theObject.x
  // could be a request of a method, or access
  // to a public variable: theObject knows which

var x:Number := 3 // confidential variable
var x:Number is public := 3 // public variable
Uniform reference to attributes

theObject.x
    // could be a request of a method, or access
    // to a public variable: theObject knows which

var x:Number := 3  // confidential variable
var x:Number is public := 3  // public variable
var x':Number := 3  // confidential
method x → Number { return x'}  // public
method x:= (newX:Number) → Done { x' := newX }  // public
Uniform reference to attributes

theObject.x
    // could be a request of a method, or access
    // to a public variable: theObject knows which

var x:Number := 3 // confidential variable
var x:Number is public := 3 // public variable
var x':Number := 3 // confidential
method x → Number { return x' } // public
method x:= (newX:Number) → Done { x' := newX } // public
method helper(...) → Done is confidential {...}
    // confidential method
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 {
    i → print(i)
}
Blocks

- Blocks are objects that represent functions
  - `{ this is a block } — a λ-expression`
  - blocks create objects that mimic functions (like Smalltalk)

```python
def welcomeAction := { print "Hello" }
```
Examples

if (x == 3) then ( print "3" )

// type error
Examples

```plaintext
if (x == 3) then ( print "3" )
    // type error

if (x == 3) then  print "3"
```
Examples

if (x == 3) then ( print "3" )
    // type error

if (x == 3) then {print "3"} // no implicit call-by-name
Examples

if (x == 3) then (print "3")
   // type error

if (x == 3) then {print "3"}
   // no implicit call-by-name

block.apply // these are different!
Examples

if (x == 3) then (print "3")
    // type error

if (x == 3) then {print "3"}
    // no implicit call-by-name

block.apply  // these are different!
block         // application is never implicit
Program Structure & Modules
a whole Grace Program

print "Hello World"
def graceModule378 = object {
    print "Hello World"
}

a whole Grace Program
a whole Grace Program

def graceModule378 = object {
  print "Hello World"
}

every Grace file defines a module
Modules are Objects

in a file called *collections.grace*:

```python
def list is public = object { ... }
def set is public = object { ... }
def dictionary is public = object { ... }
```
Modules are Objects

in a file called bingoGame.grace:

```python
import "collections" as coll

def set = coll.set

def bingoCard = set.with "Free Space"
```

...
Recall "collections.grace"

def list is public = object { ... }
def set is public = object { ... }
def dictionary is public = object { ... }
def list is public = object { ... }
def set is public = object { ... }
def dictionary is public = object { ... }
import "collections" as coll

def list is public = object { ... }
def set is public = object { ... }
def dictionary is public = object { ... }
import "collections" as coll

def list is public = object { ... }
def set is public = object { ... }
def dictionary is public = object { ... }
import "collections" as coll

def temp917 = object {
    def list is public = object { ... }
    def set is public = object { ... }
    def dictionary is public = object { ... }
}

import "collections" as coll

def temp917 = object {
    def list is public = object {
        ...
    }
    def set is public = object {
        ...
    }
    def dictionary is public = object {
        ...
    }
}

def coll = temp917
Example: importing a module

in a file called bingoGame.grace:

```python
import "collections" as coll

def set = coll.set

def bingoCard = set.with "Free Space"

...```

Example: importing a module

in a file called bingoGame.grace:

```python
def coll = temp917
def set = coll.set
def bingoCard = set.with "Free Space"
...
```
Dialects

• “Outermost” object: defines methods without explicit receiver
  ◦ e.g., turtle graphics, loops with invariants, TDD

• Top level code of dialect runs before module in the dialect
  ◦ e.g. initialize canvas, turtle ...

• Dialect runs checker over AST of module in the dialect
  ◦ Can generate new errors, such as missing type annotations, use of [ ] or match()case() ...
Dialects can define control methods

// dialect = outermost enclosing object
method do (action: Block) unless (c: Boolean) {
  if (c) then (action.apply)
}

method repeat (n: Number) times (a: Block) {
  (1..n).do { _ → a.apply }

object {
  // your program here; sends messages to
  // implicit receiver outer
}
Types

• Types classify objects
  - Type come after objects, not before
  - Structural, Gradual, Optional

```typescript
type Point = interface {
  x  →  Number
  y  →  Number
  distanceTo (other: Point)  →  Number
}
```

• Interfaces are sets of method signatures
• Types can take types as parameters (a.k.a. Generics)
No null pointer exceptions!
No null pointer exceptions!
No null pointer exceptions!

- No null
No null pointer exceptions!

- No null
- *Accessing* uninitialized variable is an error
No null pointer exceptions!

- No null
- *Accessing* uninitialized variable is an error
- Define objects for empty lists, empty trees, etc., and give them appropriate behavior
No null pointer exceptions!

- No null
- *Accessing* uninitialized variable is an error
- Define objects for empty lists, empty trees, etc., and give them appropriate behavior

```python
def emptyList = object {
    method length { o }
    method isEmpty { true }
    method head {
        noValue.raise "can't take the head of an empty list"
    }
    method tail { ... }
}
```
Type Operations

• Variants: Point | nil, Leaf[X] | Node[X]
  - x: (A | B) ≡ x: A ∨ x: B

• Algebraic constructors:
  - T₁ & T₂: intersection, conforms to T₁ and T₂
    1. Used to extend types
  - T₃ + T₄: union, conforms to T₃ or T₄
  - T₅ - T₆: structural subtraction, T₅ without T₆

• Type parameters don't need variance annotations
Match – Case

match (x) // x : o | String | Student

  // match against a constant
case { o → print("Zero") }

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

  // destructuring match, binding variables ...
case { Student(name, id) → print (name) }
Pattern-matching through method dispatch
Pattern-matching through method dispatch

match (s)
case p₁
case p₂
match \((s)\)
case \(p_1\)
case \(p_2\)

Pattern-matching through method dispatch
Pattern-matching through method dispatch
Pattern-matching through method dispatch

match(s)

match\( (s) \)
case \( p_1 \)
case \( p_2 \)

match...case

p:Pattern
match (s)
case p₁
case p₂
Pattern-matching through method dispatch

match \( (s) \)
case \( p_1 \)
case \( p_2 \)

match does different things in different patterns:
- Type patterns ask \( s \) for its type
- Literal patterns check for \( = \)
- etc
Pattern-matching through method dispatch

match (s)
case $p_1$
case $p_2$
Teaching with Grace
Designed for Flexibility

• We are not trying to prescribe how to teach programming

• Grace tries to make it possible to teach in many styles, e.g.,

✓ procedural first  ✓ object-graphics
✓ objects first   ✓ functional?
✓ turtle graphics ✓ test-driven
method toCelsius(f:Number) {
    if (f < -459.4) then { Error.raise "{f}°F is below absolute zero" }
    (f - 32) * (5 / 9)
}

print "212°F is \{toCelsius(212)\}°C"
public class Celsius {

    public static double toCelsius(double f) {
        if (f < -459.4) {
            throw new RuntimeException(f+"° Fahrenheit is below absolute zero");
        }
        return (f - 32.0) * (5.0 / 9.0);
    }

    public static void main(String[] args) {
        System.out.println("212°F is " + toCelsius(212) + "°C");
    }
}
public class Celsius {

    public static double toCelsius(double f) {
        if (f < -459.4) {
            throw new RuntimeException(f + "° Fahrenheit is below absolute zero");
        }
        return (f - 32.0) * (5.0 / 9.0);
    }

    public static void main(String[] args) {
        System.out.println("212°F is " + toCelsius(212) + "°C");
    }
}
public class Celsius {

    public static double toCelsius(double f) {
        if (f < -459.6) {
            throw new RuntimeException(
                f+"° Fahrenheit is below absolute zero");
        }
        return (f - 32.0) * (5.0 / 9.0);
    }

    public static void main(String[] args) {
        System.out.println("212°F is "+ toCelsius(212) + "°C");
    }
}
public class Celsius {

    public static double toCelsius(double f) {
        if (f < -459.4) {
            throw new RuntimeException(f + "° Fahrenheit is below absolute zero");
        }
        return (f - 32.0) * (5.0 / 9.0);
    }

    public static void main(String[] args) {
        System.out.println("212°F is " + toCelsius(212) + "°C");
    }
}
Turtle graphics

dialect "logo"

def length = 150
def root2 = 2^0.5
def diagonal = length * root2
lineWidth := 2
square(length)
turnRight(45)
penUp
forward(diagonal)
turnLeft(90)
penDown
roof(diagonal/2)

method roof(slope) {
  lineColor := red
  forward(slope)
turnLeft(90)
  forward(slope)
}

method square(len) {
  repeat 4 times {
    forward(len)
turnLeft(90)
  }
turnRight(90)
}

sample programs/house.grace
Turtle graphics

dialect "logo"

def length = 150

def root2 = 2^0.5

def diagonal = length * root2

lineWidth := 2

square(length)

turnRight(45)

penUp

forward(diagonal)

turnLeft(90)

penDown

roof(diagonal/2)

method roof(slope) {
    lineColor := red
    forward(slope)
    turnLeft(90)
    forward(slope)
}

method square(len) {
    repeat 4 times {
        forward(len)
        turnLeft(90)
    }
}

sample programs/house.grace
import "objectdraw" as od

object {
    inherit od.aGraphicApplication.size(400,400)
    var cloth  // item to be moved
    method onMousePress(mousePoint){
        cloth := od.aFilledRect.at(mousePoint)size(100,100)on(canvas)
        cloth.color := od.red
    }

    method onMouseDrag(mousePoint) → Done{
        cloth.moveTo(mousePoint)
    }

    startGraphics  // pop up window and start graphics
}
import "gUnit" as GU

method toCelsius(f: Number) {
    if (f < -459.4) then { Error.raise "{f}°F is below absolute zero" }
    (f - 32) * (5 / 9)
}

class forMethod(m) {
    inherit GU.aTestCase.forMethod(m)

    method testZero {
        assert(toCelsius(32)) shouldBe (0)
    }

    method testBoiling {
        assert(toCelsius(212)) shouldBe (100)
    }
}
method testBoiling {
    assert(toCelsius(212)) shouldBe (100)
}
method testAlaska {
    assert(toCelsius(-40)) shouldBe (-40)
}
method testTooCold {
    assert{toCelsius(-500)} shouldRaise (Error)
}

def tests = GU.aTestSuite.fromTestMethodsInClass(aTempTest)
tests.runAndPrintResults
Too Complicated!

• gUnit uses inheritance, methods, naming conventions, setup & teardown methods ...

• Instead, we have a TDD dialect, and a BDD dialect
dialect "minitest"
import "sys" as sys
import "random" as random
import "unicode" as unicode
import "linkedListWithMergesort" as list

def start = sys.elapsedTime

testSuiteNamed "list tests" with {
    test "list.empty size" by {
        assert (list.empty.size) shouldBe 0
    }
    test "list.empty do" by {
        list.empty.do { each -> failBecause "list.empty.do did!" }
        assert (true)
    }
    test "list.empty asDebugString" by {
        assert (list.empty.asDebugString) shouldBe "⊥"
    }
    test "list.empty asString" by {
        assert (list.empty.asString) shouldBe "[]"
    }
}

list tests: 31 run, 0 failed, 0 errors
palindrome tests: 8 run, 0 failed, 0 errors
time taken: 0.169s
dialect "minispec"
import "date" as date
import "io" as io

def shortFile = io.open("io-specify-hi.txt","w")
shortFile.write "hi"
shortFile.close

describe "io" with {
  specify "read returns file contents" by {
    def fs = io.open("io-specify-hi.txt","r")
    expect (fs.read) toBe "hi"
  }
  specify "size returns file size" by {
    def fs = io.open("io-specify-hi.txt","r")
    expect (fs.size) toBe 2
  }
  specify "getline on empty file returns an empty string" by {
    def fileName = "aNewFile{date.now}.txt"
    def fs = io.open(fileName, "rw") // create new empty file
    expect (fs.getline) toBe "" orSay "getline on empty file did not return empty string"
  }
  specify "getline on long file reads lines" by {

My plans for Rmod

• Implement Grace inside Pharo
  ◦ Compile Grace to Pharo objects
  ◦ interoperate with Pharo objects
  ◦ challenge: implementing objects with lexical scope
method counterPair {
    var counter: Number := 0
    def countUp = object {
        method inc { counter := counter + 1 }
        method value { counter }
    }
    def countDown = object {
        method dec { counter := counter – 1 }
        method value { counter }
    }
    object {
        method up { countUp }
        method down { countDown }
    }
}
method counterPair {

   var counter:Number := 0

   def countUp = object {
      method inc { counter := counter + 1 }
      method value { counter }
   }

   def countDown = object {
      method dec { counter := counter - 1 }
      method value { counter }
   }

   object {
      method up { countUp }
      method down { countDown }
   }

}
method counterPair {
  var counter: Number := 0
  def countUp = object {
    method inc { counter := counter + 1 }
    method value { counter }
  }
  def countDown = object {
    method dec { counter := counter - 1 }
    method value { counter }
  }
  object {
    method up { countUp }
    method down { countDown }
  }
}
method counterPair {
  var counter:Number := 0
  def countUp = object {
    method inc { counter := counter + 1 }
    method value { counter }
  }
  def countDown = object {
    method dec { counter := counter - 1 }
    method value { counter }
  }
  object {
    method up { countUp }
    method down { countDown }
  }
}
method counterPair {
    var counter:Number := 0
    def countUp = object {
        method inc { counter := counter + 1 }
        method value { counter }
    }
    def countDown = object {
        method dec { counter := counter - 1 }
        method value { counter }
    }
    object {
        method up { countUp }
        method down { countDown }
    }
}
method counterPair {
    var counter:Number := 0
    def countUp = object {
        method inc { counter := counter + 1 }
        method value { counter }
    }
    def countDown = object {
        method dec { counter := counter - 1 }
        method value { counter }
    }
    object {
        method up { countUp }
        method down { countDown }
    }
}
method counterPair {
    var counter: Number := 0
    def countUp = object {
        method inc { counter := counter + 1 }
        method value { counter }
    }
    def countDown = object {
        method dec { counter := counter - 1 }
        method value { counter }
    }
    object {
        method up { countUp }
        method down { countDown }
    }
}
method counterPair {
    var counter:Number := 0
    def countUp = object {
        method inc { counter := counter + 1 }
        method value { counter }
    }
    def countDown = object {
        method dec { counter := counter - 1 }
        method value { counter }
    }
    object {
        method up { countUp }
        method down { countDown }
    }
}
Tentative Plan

• Build module compiler using SmaCC
  ◦ Roughly follow design of existing Grace→JS compiler

• Generate Smalltalk source for ease of debugging
  ◦ design Smalltalk representations for nested objects

• Later:
  ◦ better IDE for Grace using Pharo
  ◦ Generate bytecode rather than source
Your input is needed
Your input is needed

• The reason that I’m here is that I know ...
Your input is needed

- The reason that I’m here is that I know ...

that I don’t know how to do this!
Your input is needed

• The reason that I’m here is that *I know* ...

that I don’t know how to do this!

• So: if you have opinions, suggestion, better ideas
Your input is needed

• The reason that I’m here is that I know ...

that I don’t know how to do this!

• So: if you have opinions, suggestion, better ideas
don’t keep quiet!
Your input is needed

- The reason that I’m here is that *I know* ...
  that I don’t know how to do this!

- So: if you have opinions, suggestion, better ideas
  don’t keep quiet!

- Example: perhaps I should generate the Pharo IR?
Your input is needed

• The reason that I’m here is that I know ...

that I don’t know how to do this!

• So: if you have opinions, suggestion, better ideas

don’t keep quiet!

• Example: perhaps I should generate the Pharo IR?
Your input is needed

• The reason that I’m here is that I know ...

that I don’t know how to do this!

• So: if you have opinions, suggestion, better ideas
don’t keep quiet!

• Example: perhaps I should generate the Pharo
IR?
http://gracelang.org

http://www.cs.pdx.edu/~grace/ide
Classes in Grace

• ... 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 in Grace

• ... 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}"
public class SquareWithSide implements Square {
    private int side;

    public SquareWithSide(int s) {
        side = s;
        System.out.println("Created square with side" + s);
    }

    public int area() {
        return side * side;
    }

    public void stretchBy (int n) {
        side = side + n;
    }
}
public class SquareWithSide implements Square {
   private int side;

   public SquareWithSide(int s) {
      side = s;
      System.out.println("Created square with side" + s);
   }

   public int area() {
      return side * side;
   }

   public void stretchBy (int n) {
      side = side + n;
   }
}

Create object with
new SquareWithSide(20)
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}"