Attaining Grace in Teaching Programming

Andrew Black
Michael Homer
Kim Bruce
James Noble

gracelang.org
Time for a New Language for Novices

- Java nearly 20 years old, Python older
- State of the art has advanced
  - patches look like ... patches
- Too much overhead in popular languages
  - public static void main(String[] args)
**Grace User Model**

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

- Integrate proven new ideas in programming languages into a simple o-o language.
- Gracefully represent key concepts underlying o-o programming in a way that can be easily explained.
- Allow students to focus on the essential, rather than the accidental, difficulties of programming, problem solving and system modeling.
We are in the dog food business

User model:
Beginning students

Customer:
experienced instructors

The consumer is not the customer
Everything is an object
Simple dynamic method dispatch
Single inheritance
Types are interfaces (classes ≠ types)
  Types come after objects
Blocks are first-class closures
Advanced Features

- Pattern Matching
- Extensible via Libraries (control & data)
  - Modules as objects
- Dialects to
  - expand (vocabulary, not syntax)
  - provide initialization, and
  - restrict language (enforce constraints)
“Hello World” in Grace
"Hello World" in Grace

print "Hello World!"
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("212F is " + (toCelsius(212)) + " Celsius");
    }
}
Java vs. Grace

method toCelsius(f: Number) -> 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("212F is "+(toCelsius(212))+" Celsius");
    }
}
// reads numbers from in stream returns the average
method average(in : InputStream) -> Number {
  var total := 0
  var count := 0
  while { ! in.atEnd } do {
    count := count + 1
    total := total + in.readNumber
  }
  if (count == 0) then { 0 }
  else { total / count }
}
// reads numbers from in stream returns the average
method average(in : InputStream) -> Number {
    var total := 0
    var count := 0
    while { ! in.atEnd } do {
        count := count + 1
        total := total + in.readNumber
    }
    if (count == 0) then { 0 }
    else { total / count }
}
// reads numbers from in stream returns the average
method average(in : InputStream) -> Number {
    var total := 0
    var count := 0
    while { ! in.atEnd } do {
        count := count + 1
        total := total + in.readNumber
    }
    if (count == 0) then { 0 }
    else { total / count }
}
Grace Example

// reads numbers from in stream returns the average
method average(in : InputStream) -> Number {
    var total := 0
    var count := 0
    while { ! in.atEnd } do {
        count := count + 1
        total := total + in.readNumber
    }
    if (count == 0) then { 0 } else { total / count }
Grace Example

// reads numbers from in stream returns the average
method average(in : InputStream) -> Number {

  var total := 0
  var count := 0
  while { ! in.atEnd } do {
    count := count + 1
    total := total + in.readNumber
  }

  if (count == 0) then { 0 }
  else { total / count }
}
// reads numbers from in stream returns the average
method average(in : InputStream) -> Number {
    var total := 0
    var count := 0
    while { ! in.atEnd } do {
        count := count + 1
        total := total + in.readNumber
    }
    if (count == 0) then { 0 }
    else { total / count }
}
`// reads numbers from in stream returns the average`  
`method average(in : InputStream) -> Number {`  
`    var total := 0`  
`    var count := 0`  
`    while { ! in.atEnd } do {`  
`        count := count + 1`  
`        total := total + in.readNumber`  
`    }`  
`    if (count == 0) then { 0 }`  
`    else { total / count }`  
`}`
// reads numbers from in stream returns the average
method average(in : InputStream) -> Number {
    var total := 0
    var count := 0
    while { ! in.atEnd } do {
        count := count + 1
        total := total + in.readNumber
    }
    if (count == 0) then { 0 }
    else { total / count }
}
// reads numbers from in stream returns the average
method average(in : InputStream) -> Number {
    var total := 0
    var count := 0
    while { ! in.atEnd } do {
        count := count + 1
        total := total + in.readNumber
    }
    if (count == 0) then { 0 }
    else { total / count }
}
// reads numbers from in stream returns the average
method average(in : InputStream) -> Number {
    var total := 0
    var count := 0
    while { ! in.atEnd } do {
        count := count + 1
        total := total + in.readNumber
    }
    if (count == 0) then { 0 }
    else { total / count }
// reads numbers from in stream returns the average
method average(in : InputStream) -> Number {
    var total := 0
    var count := 0
    while { ! in.atEnd } do {
        count := count + 1
        total := total + in.readNumber
    }
    if (count == 0) then { 0 }
    else { total / count }
}
Simple “method request”

Like Smalltalk and Self:

- no type-dependent 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 trying to learn not to say "message-send" or "method call").
Uniform reference to attributes

theObject.x

  // could be method request or variable access

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

theObject.x
// could be method request or variable access

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

theObject.x
  // could be method request or variable access

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

theObject.x
   // could be method request or variable access

var x:Number := 3  // confidential variable
var x:Number is public := 3  // public variable

{ var x’:Number := 3
  method x -> Number { x’ }  // public
  method x:= (newX:Number) { x’ := newX }  // public
}
“Lambdas are relegated to relative obscurity until Java makes them popular by not having them.” James Iry

Grace has λs. We call them “blocks”:

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

Blocks are objects that represent functions

\{ this is a block \} — a $\lambda$-expression

blocks create objects that mimic functions (like Smalltalk)

def welcomeAction = \{ print "Hello" \}
Blocks

- Blocks are objects that represent functions

- \{ this is a block \} — a \( \lambda \)-expression

- blocks create objects that mimic functions (like Smalltalk)

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

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

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

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

```python
welcomeAction.apply
```

```python
object { method apply
    { print "Hello" } }
```
Constructing Objects
Object constructors

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

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

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

<table>
<thead>
<tr>
<th>x</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>3</td>
</tr>
<tr>
<td>distanceTo(_)</td>
<td>...</td>
</tr>
</tbody>
</table>
class x (x': Number) y (y': Number) -> Point {
    def x : Number = x'
    def y : Number = y'
    method distanceTo (other : Point) -> Number {
        ((x - other.x)^2 + (y - other.y)^2).sqrt
    }
}
class x (x’: Number) y (y’: Number) -> Point {
    def x : Number = x’
    def y : Number = y’
    method distanceTo (other : Point) -> Number {
        ((x - other.x)^2 + (y - other.y)^2).sqrt
    }
}
class x (x’: Number) y (y’: Number) -> Point {
    def x : Number = x’
    def y : Number = y’
    method distanceTo (other : Point) -> Number {
        ((x - other.x)^2 + (y - other.y)^2).sqrt
    }
}
Classes are Factory Methods

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

class x (x’: Number) y (y’: Number)
    colour (c’ : Colour) {
        inherits x (x’) y (y’)
        def c : Colour is public = c’
    }

More on inheritance later!
Implicit initialization

Object & class expressions can contain executable code

object {
    def x : Number = 2
    def y : Number = 3
    print “Just set x to {x}”
    method distanceTo(other : Point) -> Number {
        ((x - other.x)^2 + (y - other.y)^2).sqrt
    }
    print “I’m {self.distanceTo(origin)} from origin”
}
Classes

- Classes are an implementation concept
- Inheritance via object extension
- Classes are not types
Types

- Types are for classification
  - Types logically come after objects, not before
  - Structural, Gradual, Optional

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

- Types are sets of (public) method signatures
- Types can take types as parameters (a.k.a. Generics)
Ask me about:

- (the missing) null pointer exceptions
- Pattern matching
- Blocks as partial functions
- Exceptions
- Modules as Objects
- Unit tests

Dialects for:

- restricting the language
- extending the language

Teaching Experience

- Graphics
- How you can help
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 { 0 }
    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>
  \[ x : (A | B) \equiv x : A \lor x : B \]

- Algebraic constructors:
  - T1 & T2: intersection, conforms to T1 and T2
    - Used to extend types
    - E.g., type ColorPoint = Point & {c -> Color}
  - union and subtraction available, but rarely used

- Type parameters don't need variance annotations
Match - Case

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

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

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

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

```javascript
var blk := { n : Number -> n * 2 }

blk.apply 2
  // -> 4

blk.match 2
  // -> SuccessfulMatch(4)

blk.apply "text"
  // Runtime error: wanted Number, got String

blk.match "text"
  // -> FailedMatch("text")
```
Pattern-matching through method dispatch

match (s)
case p₁
case p₂

s: Scrutinee
p: Pattern
Pattern-matching through method dispatch

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

match \( (s) \)
case \( p_1 \)
case \( p_2 \)
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 p₁
case p₂
Pattern-matching through method dispatch

match does different things in different patterns:
- ...
- destructuring patterns can extract "fields" from scrutinee
Pattern-matching through method dispatch
Pattern-matching through method dispatch
Pattern-matching through method dispatch

extract returns a tuple containing the “internal state” of the scrutinee

extract returns a tuple containing the “internal state” of the scrutinee

match(s)

MatchResult

p:Pattern

extract
Pattern-matching through method dispatch

- s: Scrutinee
- p: Pattern
- match(s)
- MatchResult
- extract
Pattern-matching through method dispatch

s:Scrutinee  

match(s)

MatchResult

tuple

extract

p:Pattern
Exceptions
Exceptions as Patterns

def myError = Error.refine "MyError"
def negativeError = myError.refine "NegativeError"
try {
    ...
    negativeError.raise "\{value\} < 0"
    ...
} catch {e: negativeError ->
    print "be more positive: \{e\}"
} catch {e: Error -> print "Unexpected Error: \{e\}"
}
a whole Grace Program

print "Hello World"
a whole Grace Program

def graceModule = object {
    print "Hello World"
}

Thursday, 17 March 2016
def graceModule = object {
    print "Hello World"
}

every Grace file defines a module
Modules are Objects

in a file called collections.grace:

```plaintext
def aList is public = object { ... }
def anArray is public = object { ... }
def aSet is public = object { ... }
def aDictionary is public = object { ... }
```
Interpreting “Import”

```
import "collections" as coll
```
Interpreting “Import”

```python
import "collections" as coll
```
Interpreting “Import”

import "collections" as coll

def temp917 = object {
    def aList is public = object { … }
    def anArray is public = object { … }
    def aSet is public = object { … }
    def aDictionary is public = object { … }
}

Thursday, 17 March 2016
Modules are Objects

in a file called bingoGame.grace:

```python
import "collections" as coll
def aSet = coll.aSet
def bingoCard = aSet.with "Free Space"
...```

Thursday, 17 March 2016
Modules are Objects

in a file called bingoGame.grace:

```python
def coll = temp917
def aSet = coll.aSet
def bingoCard = aSet.with "Free Space"
...
```
Dialects for Teaching
Dialects

- Allows methods without explicit receiver
  - e.g., loops with invariants
- Top-level code in dialect runs first
  - Initialize canvas, turtle initialization,...
- Run an included checker over AST
  - Can exclude code
  - e.g., require type annotations, invariants
  - Pluggable static typing as part of dialect
Dialects Simple to Use

dialect "simpleControl"

// your program here;

// Uses constructs of dialect
In Module "simpleControl":

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

... ...

method while (c : Block) do (a : Block) {
  if (c) then {
    a.apply
    while (c) do (a) }
}
```
Dialects provide outer scope

object { // outermost enclosing object
    method if (c) then (t : Block) else (f : Block) {
        c.ifTrue ( t ) else ( f )
    }
    ...
    method while (c : Block) do (a : Block) {
        if (c) then { a.apply; while (c) do (a) }
    }
}

object {
    // your program here; sends messages to
    // implicit receiver outer
}
What Was Tricky?
Inheritance in Grace
Inheritance in Grace

Key concept in OO languages
Inheritance in Grace

- Key concept in OO languages
- Grace treats objects as primitive,
  - Classes as syntactic sugar
Inheritance in Grace

- Key concept in OO languages
- Grace treats objects as primitive,
  - Classes as syntactic sugar
- How to define inheritance from object?
Inheritance in Grace

- Key concept in OO languages
- Grace treats objects as primitive,
  - Classes as syntactic sugar
- How to define inheritance from object?
- Abadi-Cardelli punted
  - Though talked about delegation
  - Defined inheritance for classes only
A Taste of Theory

Semantics of OO languages

- Classes are generators of fixed points (Cook, …)
  - Contain “pre-methods” (parameterized on self)
- Objects are the fixed points (give meaning to self!!)
  - Methods have self baked in.
A Taste of Theory

Semantics of OO languages

- Classes are generators of fixed points (Cook, ...)
  - Contain "pre-methods" (parameterized on self)
- Objects are the fixed points (give meaning to self!!)
  - Methods have self baked in.

Inheritance on classes

- Copy (pre)methods from superclass
- Add/replace (pre)methods from subclass
- Super is collection of (pre)methods from superclass.
Initialization of Fields for New Objects

- Allocate space for all features
- Run initializers for superclasses
- Run initializers for subclass
Initialization can request methods on self
Tricky Bits

- Initialization can request methods on self
- Object can be released into wild before initialization completed
  - Request method with self as parameter
  - E.g., register graphics object on canvas
Tricky Bits

- Initialization can request methods on self
- Object can be released into wild before initialization completed
  - Request method with self as parameter
  - E.g., register graphics object on canvas

Issues:
- What is meaning of self during initialization?
- Initialize with original method, or with override?
  - C++ vs Java semantics
Inheriting from Objects
Inheriting from Objects

- Inheriting methods is same
  - Actually keep pre-methods, not methods
Inheriting from Objects

- Inheriting methods is same
  - Actually keep pre-methods, not methods
- Allocate space for features
Inheriting from Objects

- Inheriting methods is same
- Actually keep pre-methods, not methods
- Allocate space for features
- How to initialize inherited fields?
  - Clone?
  - Should everything support cloning?
  - What kind of clone?
Inheriting from Objects

- Inheriting methods is same
  - Actually keep pre-methods, not methods
- Allocate space for features
- How to initialize inherited fields?
  - Clone?
  - Should everything support cloning?
  - What kind of clone?

Rejected delegation as too complex for novices
Our Solution

- Allow inheritance from new objects only
- To create object by inheritance
  - Create method suite as before
  - Allocate space for fields
  - Run initialization code for creating super-object
    - but on new object
  - Run initialization code for sub-object
Issues

- Works fine when inheriting from classes
- With objects, not so nice:
  - Write clone-like methods if want to inherit
  - What about immutable objects?
    - Why write clone when no initialization??
- Is there a better solution?
Issues

- Works fine when inheriting from classes

- With objects, not so nice:
  - Write clone-like methods if want to inherit
  - What about immutable objects?
    - Why write clone when no initialization??

- Is there a better solution?

- Simplest solution for immutable objects:
  - Change superobject to a class
import "x" as x                // keep types
import "x" as x : Dynamic      // throw types away

import "xSpec" as xSpec        // separate spec and impl
import "xImpl" as x : xSpec.T

import "xSpec" as spec         // I conform to external spec
assertType<spec.T>(self)

type ExpectedType = { ... }    // import confirms to local spec
import "x" as x : ExpectedType
Asynchrony & Parallelism
Asynchrony & Parallelism

- Hypothesis: we don't know what to do about parallelism!
Asynchrony & Parallelism

Hypothesis: we don’t know what to do about parallelism!

Conclusion: we must support different “models”
Asynchrony & Parallelism

Hypothesis: we don’t know what to do about parallelism!

Conclusion: we must support different “models”

Software Transactional Memory (Clojure)
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)
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)
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
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
- ...

Thursday, 17 March 2016
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
  - ✓ objects first
  - ✓ turtle graphics
  - ✓ object-graphics
  - ✓ functional?
  - ✓ test-driven
Turtle graphics

dialect "logo"

def length = 150
def diagonal = length * 2.sqrt
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)
tturnRight(90)
    }
}

sample programs/house.grace
Turtle graphics

dialect "logo"

def length = 150
def diagonal = length * 2.sqrt
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)
        turnRight(90)
    }
}

sample programs/house.grace
objectdraw Graphics

dialect objectdraw

object {
    inherits aGraphicApplication.size(400,400)
    var cloth // item to be moved

    method onMousePress(mousePoint) {
        cloth := aFilledRect.at(mousePoint).size(100,100).on(canvas)
        cloth.color := red
    }

    method onMouseDrag(mousePoint)->Done {
        cloth.moveTo(mousePoint)
    }

    startGraphics // pop up window and start graphics
}

Thursday, 17 March 2016
Unit testing dialect

dialect "minitest"

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

testSuiteNamed "temperature conversion" with {
    test "zero" by {
        assert(toCelsius(32)) shouldBe (0)
    }
}
test "Boiling" by {
    assert(toCelsius(212)) shouldBe (100)
}

test "Alaska" by {
    assert(toCelsius(-40)) shouldBe (-40)
}

test "TooCold" by {
    assert(toCelsius(-500)) shouldRaise (Error)
}

dialect "minitest"

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

testSuiteNamed "temperature conversion" with {

test "zero" by {
    assert(toCelsius 32) shouldBe 0
}

test "Boiling" by {
    assert(toCelsius 212) shouldBe 100
}

test "Alaska" by {
    assert(toCelsius(-40)) shouldBe (-40)
}

test "TooCold" by {
    assert(toCelsius(-500)) shouldRaise (Error)
}
}

Run

temperature conversion: 4 run, 0 failed, 0 errors
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

2014 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?
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, 2014: experimental classroom use

2015 1.0 language spec, robust implementations, textbooks, initial adopters for CS1/CS2

2016? ready for general adoption?
Help!

- Supporters
- Programmers
- Implementers
- Library Writers
- IDE Developers
- Testers
- Spec critics
- Teachers
- Students
- Tech Writers
- Textbook Authors
- Blog editors
- Community Builders
No conclusions — we aren’t done yet

Questions
Comments
Suggestions
Brickbats
Workshop at ECOOP this summer

http://gracelang.org

http://web.cecs.pdx.edu/~grace/minigrace/exp/