Today, you have been hearing about all sorts of clever new languages, language features, and uses of language features …

And now for something completely different …

Now, we are going to tell you about a boring language with no new language features, or uses of language features …

And now for something completely different …

Grace
A New Educational
Object-Oriented Programming
Language

Andrew Black
Kim Bruce
James Noble

Suppose:

- You are going to teach object-oriented programming to 1st year students.
- What language would you choose?
Which language?

- ECOOP 2010: we don't like the available options
- "Professional" languages too complex for teaching (Scala, C#, Java ...)
- Smalltalk doesn't support static typing; Python has inconsistent method syntax, no encapsulation
- Group decision: design a modern object-oriented language specifically for teaching

High Level Goal

- "A Haskell for OO"
- Integrate proven newer ideas in programming languages into a simple language for teaching
  - language features represent key concepts cleanly
  - allow students to focus on the essential, rather than accidental, complexities of programming and modelling.

Objectives

- Low overhead for simple programs
- Good IDE support for novices
- Simple semantic model
- Support a variety of approaches to teaching
  - Objects-first and objects-late
  - Untyped, Typeful and Gradually-typed
- Easy transition to other languages

Best of 20th Century-Technology

- Closures
- Assertions, unit testing, traces, and tools for finding errors
- High level constructs for concurrency
- Support for immutable data
- Generics (done right)
Influences

- Static world:
  - Eiffel, Java, C#, Scala, ...
- Dynamic world:
  - Smalltalk, Python, Scheme/Racket, ...

Simplest Programs

- Hello, World!
  ```plaintext```
  print "Hello, World"
  ```plaintext```
- “Top level” code is considered to be inside the “default object”
  ```plaintext```
  object {
    print "Hello, World"
  }
  ```plaintext```
- An object with 0 methods and 1 statement
  ```plaintext```
  Object can contain code that is executed when created

Simple methods

- Methods can also be defined and used at the “top level”:
  ```plaintext```
  method celsiusToFahrenheit (temp) {
    ((temp * 9) / 5) + 32
  }
  print "20° Celsius is {celsiusToFahrenheit 20}° Fahrenheit"
  ```plaintext```

Types are optional

- The same code with type annotations:
  ```plaintext```
  method celsiusToFahrenheit (temp: Number) -> Number {
    ((temp * 9) / 5) + 32
  }
  print "20° Celsius is {celsiusToFahrenheit 20}° Fahrenheit"
  ```plaintext```
- Programmer decides whether typing is static, dynamic or ...
- All options are type-safe
Clean Concepts

- **Numbers**
  - $23 \times 10^{11}$, $1.75$, $1.414214$, $-1$ (all exact)

- **Methods on Numbers**
  - $20 + 43$, $7/4$, $20$.factorial (all exact)
  - $2 \sqrt{\pi}$ (approximate)

- **Objects**
  - ```
  object {
    method radius { 5 }
    method area { (radius^2)*\pi }
  }
  ```

- **Constants in Objects are Accessed as Methods**
  - ```
  disk.radius       answers 5
  disk.area         answers -78.53981...
  ```

- **Constant Binding**
  ```
  def cost = quantity * unitPrice
  def disk = object {
    def radius = 5
    method area { (radius^2)*\pi }
  }
  ```

- **So, it doesn’t matter if we define**
  ```
  def disk = object {
    def radius = 5
    method area { (radius^2)*\pi }
  }
  ```
  or
  ```
  def disk' = object {
    method radius { 5 }
    method area { (radius^2)*\pi }
  }```
variable binding

var sum := 0
var speed := 2
var invoiceDate := aDate.today

methods and blocks can have temporary variables

objects can have instance variables

Instance variables

def adjustableDisk = object {
    var radius := 5
    method area { (radius^2)*π }
}

Instance variables bindings can be changed using methods (unless they are confidential):

adjustableDisk.radius := 1

Classes codify factories:

class aDisk.ofRadius(r) {
    method radius { r }
    method area { (radius^2)*π }
    method > (other) {
        radius > other.radius
    }
}

def myDisk = aDisk.ofRadius(7)
def yourDisk = aDisk.ofRadius(8)

Object factories:

def aDisk = object {
    method ofRadius(r) {
        object {
            method radius { r }
            method area { (radius^2)*π }
            method > (other) {
                radius > other.radius
            }
        }
    }
}
def myDisk = aDisk.ofRadius(7)
def yourDisk = aDisk.ofRadius(8)
**Object composition:**

```plaintext
def hole = aDisk.ofRadius (h/2)
def outside = aDisk.ofRadius (d/2)
method area { outside.area - hole.area }
```

```plaintext
class aWasher.holeDiameter (h) outerDiameter (d) {
def hole = aDisk.ofRadius (h/2)
def outside = aDisk.ofRadius (d/2)
method area { outside.area - hole.area }
}
```

Grace supports multipart method names ("mixfix")

**Object inheritance:**

```plaintext
def cylinder = object {
inherits aDisk.ofRadius (r)
def height = h
method volume { area * height }
}
```

```plaintext
class aCylinder.baseRadius (r) height (h) {
inherits aDisk.ofRadius (r)
def height = h
method volume { area * height }
}
```

Grace supports multipart method names ("mixfix")

**Returning multiple results**

Grace does not support multiple results. But it's easy to return an object:

```plaintext
method split (filename) {
def dot = filename.indexOf(".")
object {
  def base = filename.upto (dot-1)
  def extension = filename.from (dot+1)
}
}
```

Grace answers an object with 2 methods

**Closures**

With or without parameters:

```plaintext
{ print "hello" }
{ x,y -> print ("adding " ++ x ++ " to " ++ y ++ " gives " ++ (x+y))}
```

represented by objects with “apply” method

```plaintext
object { method apply(x,y) { print ... } }
```

Real lexical scope
Building Control Structures

- Closures support definition of control constructs in libraries:
  - class List {
    method forEach (actionClosure) {...}
  }
  - myList.forEach {x -> ...}

Delayed Evaluation Visible

- if (someCond) then {C} else {D}
- while {someCond} do {C}
- if (someCond) then {C} else {
  if (otherCond) {D} else {E}}

Other Grace Features

- Types (= interfaces) ≠ classes
- Visibility: public & confidential
- Support for immutable objects
- Equals & hashcode built-in (like Eclipse)
- Number consists of Rationals & Binary64 floats

Typing Disciplines

- Experimentalist (flower child):
  - Dynamic typing: Do what you want — we’ll make sure it’s safe at run-time ...
- TRC regulated:
  - Static typing: We’ll make sure everything is safe before we let you do it.
  - But semantics of type-safe programs are same either way.
  - ... though some may not be allowed by TRC.
All Disciplines Interoperate

- Mixing disciplines helps students/programmers migrate from dynamically to statically typed languages.
- What does a type annotation mean in a dynamically typed language?
  - Represents a claim - generates a dynamic check like "assert s.nonempty"
- What does a type annotation mean in a statically typed language?
  - Represents provably correct assertion

Advanced Features

Pattern Matching

```scala
method matchTest (x: Number) {
  match(x)
    case {1 -> "one"}
    case {2 -> "two"}
    case {_ -> "lots"}
}
```

Variant Types

- Object types don't contain null value
- Avoid Hoare’s “billion dollar mistake”
- Construct as needed from singleton and variant types:
  - def notThere = object { method asString {...}...}
  - type Result = String | notThere
Using a variant

```java
method doSomething(key: KeyType) {
    match(table.valueOf(key))
        case {v:String ->
            out.println(... ++ v)
            lastValue := v
        case {notThere ->
            out.println(... ++ " is empty")
        }
}
```

Provide more powerful pattern matching?

---

Language Levels

- Accomplished via libraries
- Libraries package together classes and objects
  - "use" object or class ⇒ inherit public features
- Need to develop useful pedagogical IDEs

---

Why Consider Using Grace?

- Clean Syntax
- Simple uniform semantic model
  - no static features, no overloading, no null, etc.
  - Everything is an object (even lambdas)
- Modern features
  - Generics done right, closures, case/pattern matching
  - Syntax supporting design of control structures
- 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}, futures f := async{...},forall(...) {stmts}, isolated{stmts}
  - Support for immutable objects
Current State of Grace

- 2011: 0.1, 0.2 and 0.3 language releases, prototype implementations ✔
- 3 implementations in progress, spec at 0.35
- 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

Information, blog, discussion:
http://www.gracelang.org
Try Grace in your browser:
http://homepages.ecs.vuw.ac.nz/~mwh/minigrace/js/