Lecture 24: OO Languages: Grace

CSC 131
Spring, 2019

Kim Bruce

Eiffel

- Introduced in 1985 by Bertrand Meyer
- Design goals:
  - Promote clear and elegant programming.
  - Support object-oriented design, including "design-by-contract"
- Design-by-contract is most important impact

Design by Contract

- Treat method calls as contractual obligations
  - Client must ensure that preconditions of the method are met when sending a message.
  - If client meets the preconditions then the routine guarantees that the postconditions will hold on exit.
  - Both parties may also guarantee that certain properties (the class invariant) hold on entrance to methods and again on exit.

Class Definition

```
class
    HELLO_WORLD
create
    make
feature
    make
        do
            print ("Hello, world!%N")
        end
    -- other method defs
invariant
    -- class invariant
end
```

Method Definition

```
connect_to_server (server: SOCKET)
    -- Connect to a server or give up after 10 attempts.
  require
    server /= Void and then server.address /= Void
  local
    attempts: INTEGER
  do
    server.connect
  ensure
   connected: server.is_connected
  rescue
    if attempts < 10 then
      attempts := attempts + 1
      retry
    end
  end
```

Inheritance & Assertions

- What changes can you make in preconditions and postconditions of method when override?
- Need to maintain contract as masquerades.
- *Answer is homework question!*

Static Typing Issues

- In Eiffel subclass, can
  - specialize type of instance variables
  - specialize return type of methods
  - specialize parameter type of methods
- First & third lead to errors
- Several proposals made to fix, including whole-program analysis
  - None appear to have been implemented

like Current

```
class LINKABLE [G]

feature
   item: G;
   right: like Current;

   putRight (other: like Current) is
      do
         right := other
      ensure
         chained: right = other
      end;

end -- class LINKABLE
```

Type *like Current* is type of class

```
class BILINKABLE [G] inherit LINKABLE [G]
           redefine
              putRight
           end

feature
   left: like Current;   -- Left neighbor

   putRight (other: like Current) is
         -- Put `other' to right of current cell.
      do
         right := other;
         if (other /= Void) then
            other.simplePutLeft (Current)
         end
       end;

    putLeft (other: like Current) is ...
```

Very Flexible

like Java extends

- Define
  - class LINKEDLIST[NODE -> LINKABLE] ...
  - Can instantiate with LINKABLE to get singly-linked list or BILINKABLE to get doubly-linked list.
- Can't do in Java or C++!
  - Why?
- Type Unsafe
  - *See next week's homework – implicit change of parameter type*
  - Subclass, but not subtype

Grace

Grace

- New language designed for teaching novices
  - Under development at Pomona, Portland State, and Victoria University, Wellington, NZ
  - Several published papers, nearly complete implementations
- Goal: Integrate current ideas in programming languages into a simple, general-purpose language aimed at novices.

Why New Language for Novices?

- Most popular languages too complex & low-level.
- Complexity necessary for professionals, but ...
  - "Accidental complexity" of language can overwhelm "essential complexity".
  - Minimize language complexity so can focus on programming/design complexity.

Existing Languages Woefully Out-of-date

- C (1972), C++ (1983), Python (1989), Java (1994)
- History of pedagogical languages:
  - Basic, Logo, Pascal
  - *... but not recently!*
  - *Miniworlds different: Alice, Karel the Robot, Greenfoot*

Java Problems

- **public static void** main(String [] args)
- Primitives *versus* objects, "==" *versus* "equals"
- Flawed implementation of generics
- Static *versus* instance – on variables & methods
- float *versus* double *versus* int *versus* long

Python Problems

```
>>> class aClass:
    """A simple example class"""
    val = 47
    def f(self):
        return 'hello world'

>>> x = aClass()
>>> x.value = 17
>>> x.val
47
>>> x.f()
'hello world'
```

disappearing `self`?

uncaught typos

no information hiding

Fine for scripting, but not large-scale software development

What if we could have:

- Low syntactic overhead of Python, *but with*
  - information hiding
  - consistent method declaration & use
  - required variable declarations
  - optional (& gradual) type-checking
  - direct definition of objects
  - first-class functions

Hello World in Grace:

```
print "hello world"
```


Objects

Consistent indenting is required! But no semicolons.

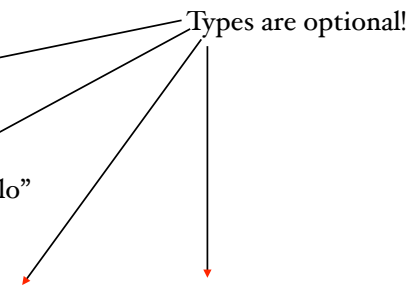
```
def mySquare = object {
      var side := 10
      method area {
            side * side
      }
      method stretchBy(n) {
            side := side + n
      }
}
```

Defaults: instance variables and constants are confidential (protected), methods are public

Annotations can override the defaults

Objects contain declarations

- definitions:
  - def x:Number = 17
- variables:
  - var y: String := “hello”
- methods:
  - method m(w:Number,z:String) -> Done {...}
- types:
  - type Point = interface {x -> Number
    y -> Number ...}



Typed Objects

```
type Square = interface {                    like Void
      area -> Number
      stretchBy(n:Number) -> Done
}

def mySquare: Square = object {
      var side: Number := 10
      method area -> Number {
            side * side
      }
      method stretchBy (n:Number) -> Done {
            side := side + n
      }
}
```


Classes

- Classes take parameters and generate objects

```
class squareWithSide (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}"
}
```

Type annotations can be omitted or included

Or Object w/Factory Method

```
method squareWithSide (s:Number) -> Square {
   object{
      var side: Number := s
      method area -> Number {
         side * side
      }
      method stretchBy(n:Number)-> Done {
         side := side + n
      }
      print "Created square with side {side}"
   }
}
```

What is type of square?

Blocks

- Syntax for anonymous functions

```
def double = {n -> n * n}
double.apply(7)  // returns 49
// block is implicitly object w/apply method

def nums = aList.from(1)to(100)
def squares = nums.map {n -> n * n}
```

function

multipart method names

Blocks can take 0 or more parameters

Blocks

- Blocks make it simple to define new "control structures" as methods

block, evaluated repeatedly

```
while {boolExp} do { someStuff }
```

boolean expression, evaluated once

```
squares.forEach {n ->
        if (n.isEven) then {print n}
}
```

Parentheses can be dropped if argument bounded by {} or ""
No parens needed for parameterless methods

Error Actions

- Grace encourages the use of blocks to specify error actions or default values:

```
var x := table.at(key)ifAbsent{
            return unknown(key)
        }
```

Running Grace

- Compiler generates Javascript
- Use web-based editor/compiler at http://web.cecs.pdx.edu/~grace/ide/

Grace on the Web

- Go to:
  - http://web.cecs.pdx.edu/~grace/ide/
  - Click on document icon with plus:  to start new file or click on up arrow to load existing program.
  - "Run" button under edit window will compile and execute code.
  - Right-click down arrow (& "Save link as...") to save.

Sample Grace Code

- See ComplexNumbers.grace

Avoid Hoare's "Billion Dollar Mistake"

- No built-in **null**
- Accessing uninitialized variable is error
- Replace **null** by:
  - sentinel objects, or
  - error actions

Sentinel Objects

A real object, tailored for the situation, *e.g.*:

```
def emptyList = object {
    method asString {"<emptyList>"}
    method do(action) {}
    method map(function) {self}
    method size {0}
}
```

name for object being defined

Sentinel Objects

Simplifies code, eliminates testing for **null**

```
class aListHead(fst) tail (rest) {
    method asString {"({fst}:{rest})"}
    method head {fst}
    method tail {rest}
    method do(action) {
        action.apply(head)
        tail.do(action)
    }
    method map(function) {
        aListHead (function.apply (head))
                  tail (tail.map (function))
    }
    method size {1 + tail.size}
}
```

boolean expression, evaluated once

Error Actions

- Grace encourages the use of blocks to specify error actions or default values:

```
var x := table.at(key)ifAbsent{
            return unknown(key)
         }
```

Works great for listeners as well!

Pattern Matching

- Provides type-safe switch/case

```
match(myVal)
    case{ n: Number -> "The number {n} seen"}
    case{ s: String -> "The string "++s++" seen"}
    case{ true: Boolean -> "This is true!"}
```


Variant Types

```
type NumOrString = Number | String
var x: NumOrString := if (…) then (…) else (…)
match(x)
   case {x': Number -> "value of x is {x'}}
   case {s: String -> "value of x is" ++ s}
```

`val: A | B` iff `val:A` or `val:B`

Allows elimination of null

Modules in Grace

- Code in separate files imported as though in an object with given name:
  - import "myfile" as libName
  - libName.m(...)
-