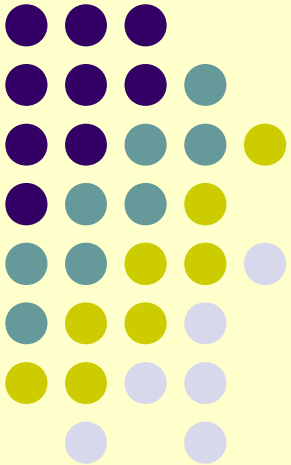
Forsaking Inheritance: Supercharged Delegation in DelphJ

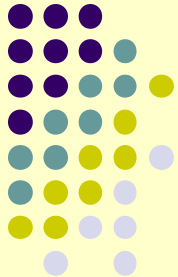
Yannis Smaragdakis
University of Athens

joint work with Prodromos Gerakios
and Aggelos Biboudis,
building on work by Shan Shan Huang

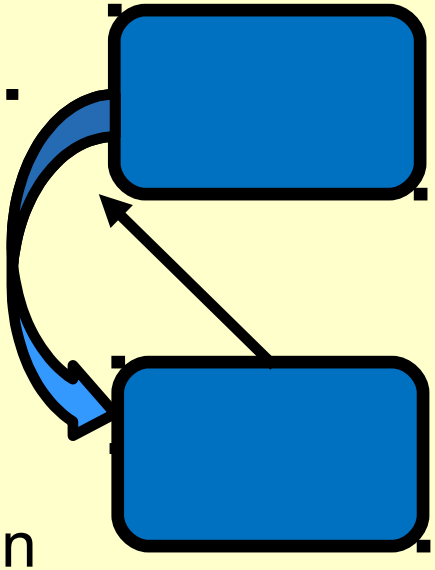


Research sponsors:
European Research Council
Established by the European Commission
European Union
European Social Fund
OPERATIONAL PROGRAMME
EDUCATION AND LIFELONG LEARNING
investing in knowledge society
MINISTRY OF EDUCATION & RELIGIOUS AFFAIRS, CULTURE & SPORTS
MANAGING AUTHORITY
NSRF 2007-2013
programme for development
EUROPEAN SOCIAL FUND
Co-financed by Greece and the European Union
LOGICBLOX®

Inheritance: A Love-Hate Affair (we love to hate it)

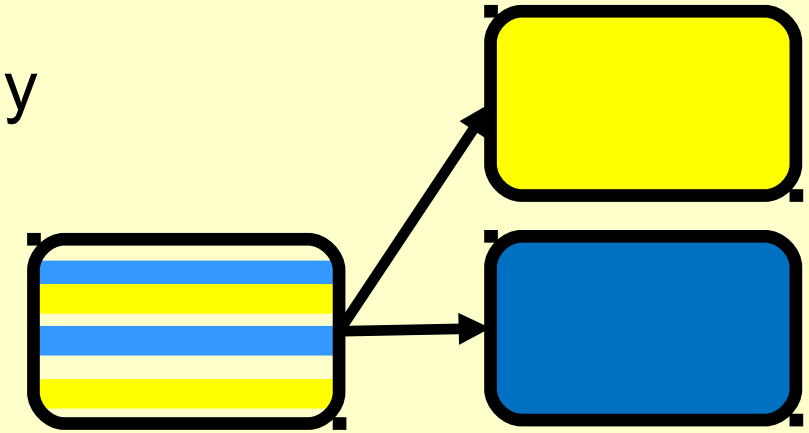


- Confusing
  - subtyping vs. subclassing
- Coarse-grained
  - inherit all-or-nothing
- Bad for reuse
  - a reuse mechanism that plays badly when one wants to reuse from more than one place!
- Rigid
  - fixed at subclass development time



Alternative: Delegation

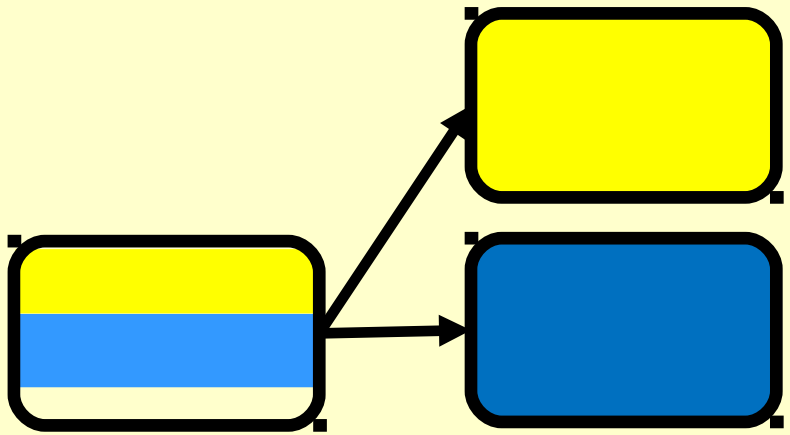
- ```
  class Refinement {
      Base b;
      void foo() {… b.foo(); … }
  }
  ```
- Completely manual
  - need to forward explicitly




  ```

Our Past Work: Morphing

- Can make delegation more automatic
  - *consultation* or *forwarding*
- ```
  class Logger {
    Subj ref;
    …
    <R,A>[m] for (R m(A): Subj.methods)
    R m (A a) {
      log(m.name, a);
      return ref.m(a);
    }
  }
  ```

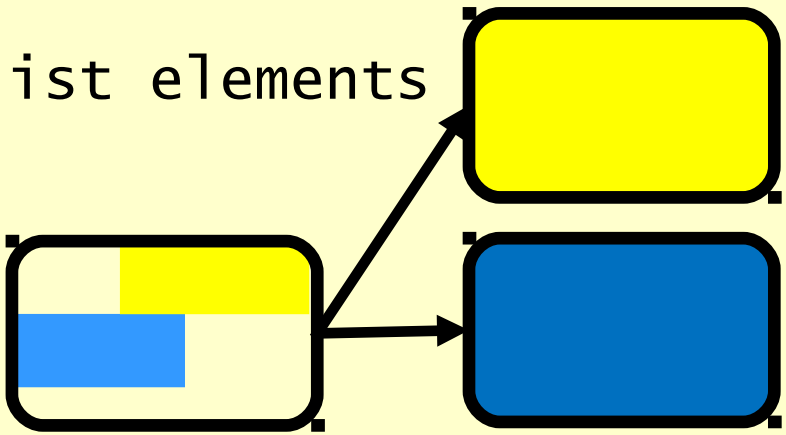



  ```

More Morphing

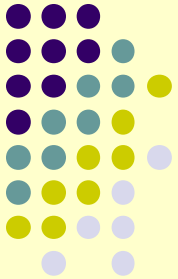
- Can do a lot more

- ```
  class Listify {
    Subj ref;
    …
    <R,A>[m] for (R m(A): Subj.methods)
    R m (List<A> a) {
      … // call m for all list elements
    }
  }
  ```

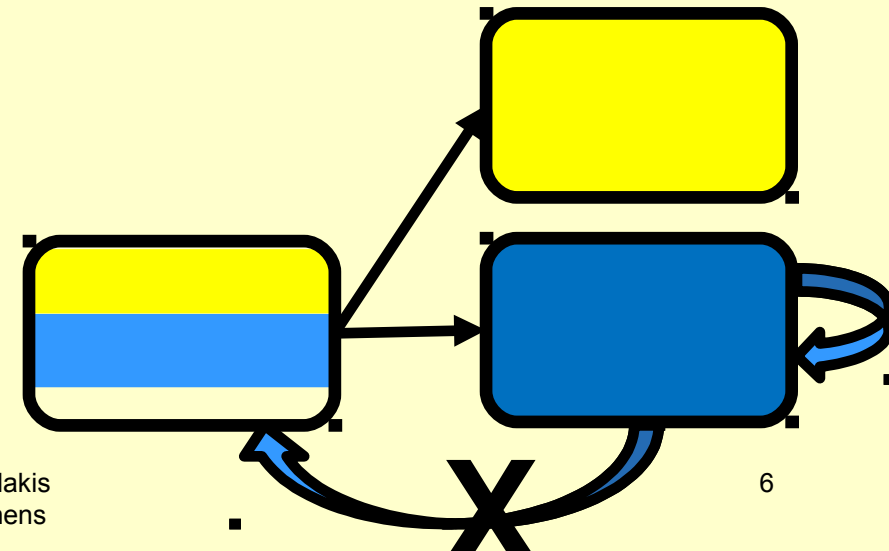



  ```

Morphing Still No Substitute For Inheritance

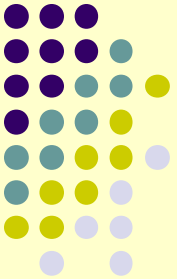


- No *late binding*
  - cannot change reused functionality
- ```
  class C {
    Subj ref; // Subj defines and calls foo
    …
    <R,A>[m] for (R m(A): Subj.methods)
    R m (A a) { …
      return ref.m(a);
    }
    void foo() {…}
  }
  ```




  ```

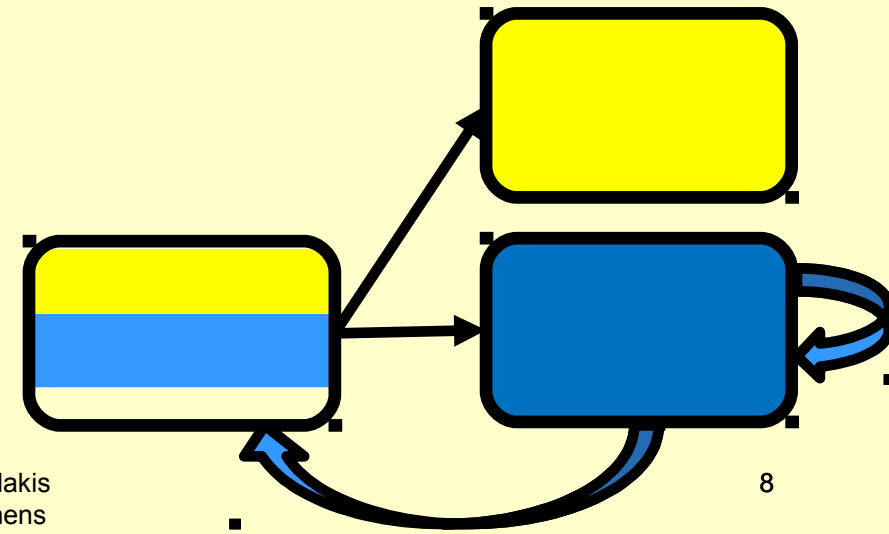
Mechanisms Combining Delegation and Late Binding



- There are past mechanisms combining delegation and late binding
  - Kniesel's work, Ostermann's, others
- But this makes delegation be more like inheritance
  - automatically forward all methods, not the ones chosen
- Need to combine with morphing
  - we next see our current design

New Construct: subobject

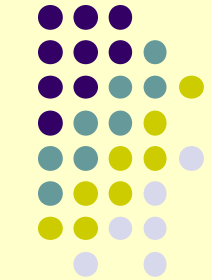
- Per-field late binding designations
  - obvious question: is field mutable?
-
```
class Logger {
   subobject Subj ref;
   …
   <R,A>[m] for (R m(A): Subj.methods)
   R m (A a) {
     log(m.name, a);
     return ref.m(a);
   }
}
```



The Good Part

- Works fine for our original problems
  - e.g., multiple subobjects
-
```
class GradStudent {
  subobject Student sref;
  subobject Employee eref;
  …
  <R,A>[m]
  for (R m(A): Student.methods;
       no R m(A): Employee.methods)
  R m (A a) { … }
  … // handle other two cases
}
```

Subtlety: Accidental Overriding (avoided)

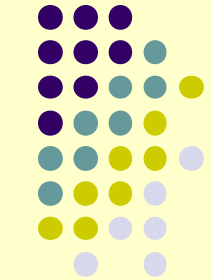


- ```
  interface I { void meth(); }
  class Unsuspecting implements I {…}

  class C {
    subobject I ref;
    C(I i) { ref = i; }
    …
    void foo() {…}
  }
  C c = new C(new Unsuspecting());
  ```
- If `Unsuspecting` defines a `foo`, should `c` override it with `c`'s version?


  ```

Subtlety: Accidental Overriding (avoided)

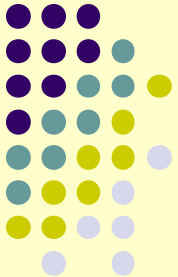


- ```
  interface I { void meth(); }
  class Unsuspecting implements I {…}

  class C {
    subobject I ref;
    C(I i) { ref = i; }
    …
    void foo() {…}
  }
  C c = new C(new Unsuspecting());
  ```
- Our policy: can override only non-final methods that are declared in static type of subobject field

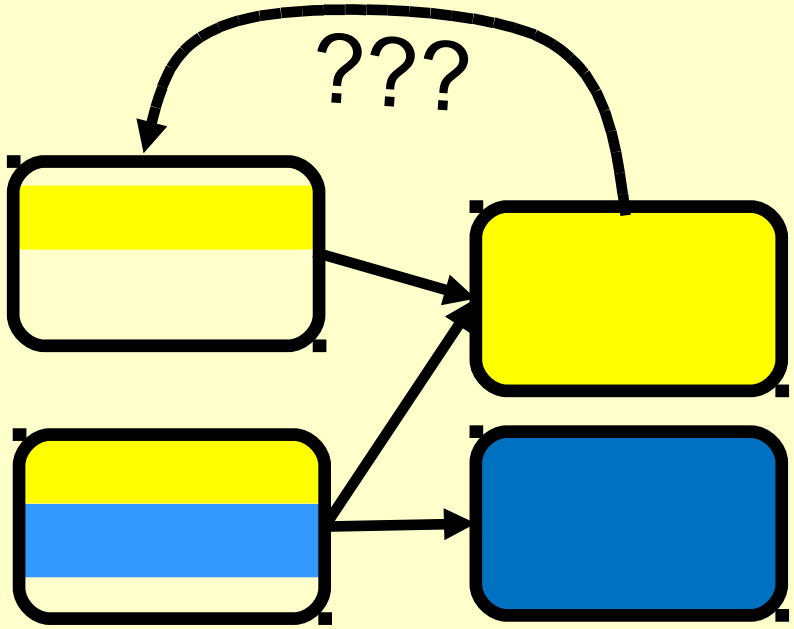

  ```

Subtlety: Per-Field Late Binding?

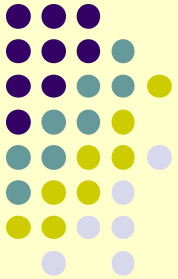


- Inheritance has it easy: the superclass subobject is both *owned* and *immutable*
  - we explored a fully liberal design
  - subobjects can be aliased by multiple wrapper objects
  - subobject fields are mutable
- Severe consequences for execution (and semantics)
  - alternative past designs had the object itself keep a notion of “self”, different from “this”

Aliasing



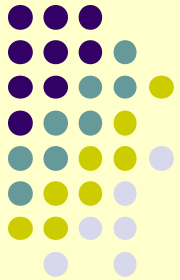
Access Paths



- Wrapping of subobject captured in references!
  - references in our design are heavy-duty
- ```
  class Wrapper {
    subobject Subj ref; …
  }
  Subj subj = new Subj();           // object s1
  Wrapper w1 = new Wrapper(subj); // object o1
  Wrapper w2 = new Wrapper(subj); // object o2
  Subj alias = w2.ref;
  ```
- subj and alias not same!
- alias == o2->$_{ref}$s1


  ```

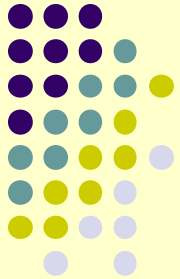
When Do These Change?



- Access paths are copied on every reference assignment, built up on field write
- ```
  Subj subj1 = new Subj();          // object s1
  Subj subj2 = new Subj();          // object s2
  Wrapper w1 = new Wrapper(subj1); // object o1
  Wrapper w2 = new Wrapper(subj2); // object o2
  Subj aliasForS2 = w2.ref;
  w1.ref = aliasForS2;
  ```
- *One way* to view: only keep last object of assigned ref's access path, append to lhs of assignment
  - `w1.ref == aliasForS2 == o2->refS2`


  ```

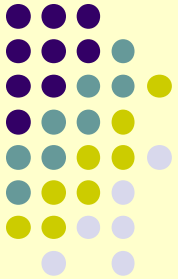
Another Way to View



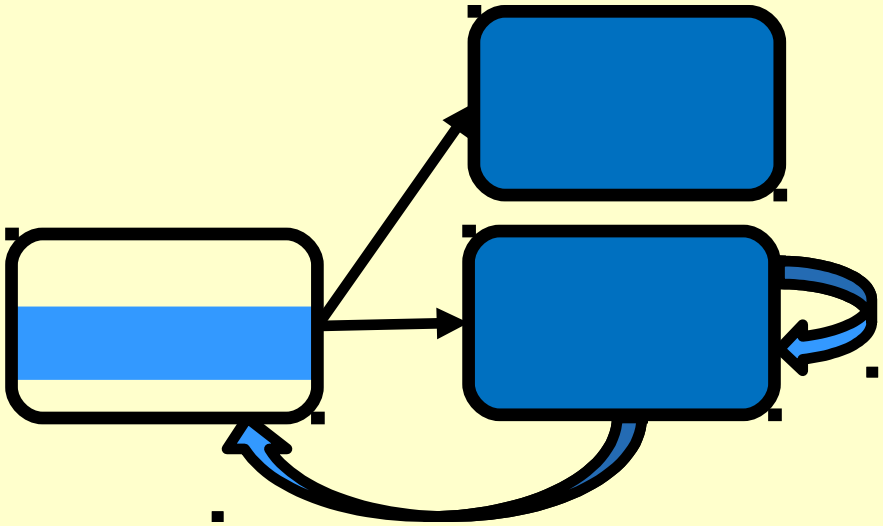
- Every *stack* reference represents a full access path but *heap* references do not
- ```
  Subj subj1 = new Subj();          // object s1
  Subj subj2 = new Subj();          // object s2
  Wrapper w1 = new Wrapper(subj1); // object o1
  Wrapper w2 = new Wrapper(subj2); // object o2
  Subj aliasForS2 = w2.ref;
  w1.ref = aliasForS2;
  ```
- Access paths built up on field read
  - `w1.ref == w2.ref == s2`
  - `aliasForS2 == o2->ref s2`


  ```

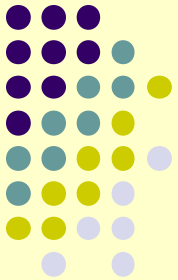
Also Prevents Surprises with Mutable References



- Since we have per-reference access path: `this` does not change by mere reassignment of wrapper fields



To Summarize



- Morphing can emulate inheritance and address its shortcomings
  - automation but with control
    - no all-or-nothing reuse
    - no conflicts when reusing from multiple sources
    - real reuse: single pattern for many methods
  - all with modular type safety
  - everything works with generic/unknown field types

Caveats

- But need deep delegation
- Subtle, complex consequences of per-field late binding
  - aliasing of subobjects seems inevitable
  - mutability of subobject references a design choice
- Is this a reasonable programming model?
- Can it be implemented efficiently?
  - a reference becomes an entire data structure!