Vrije Universiteit Brussel
Faculty of Sciences
Computer Science Department

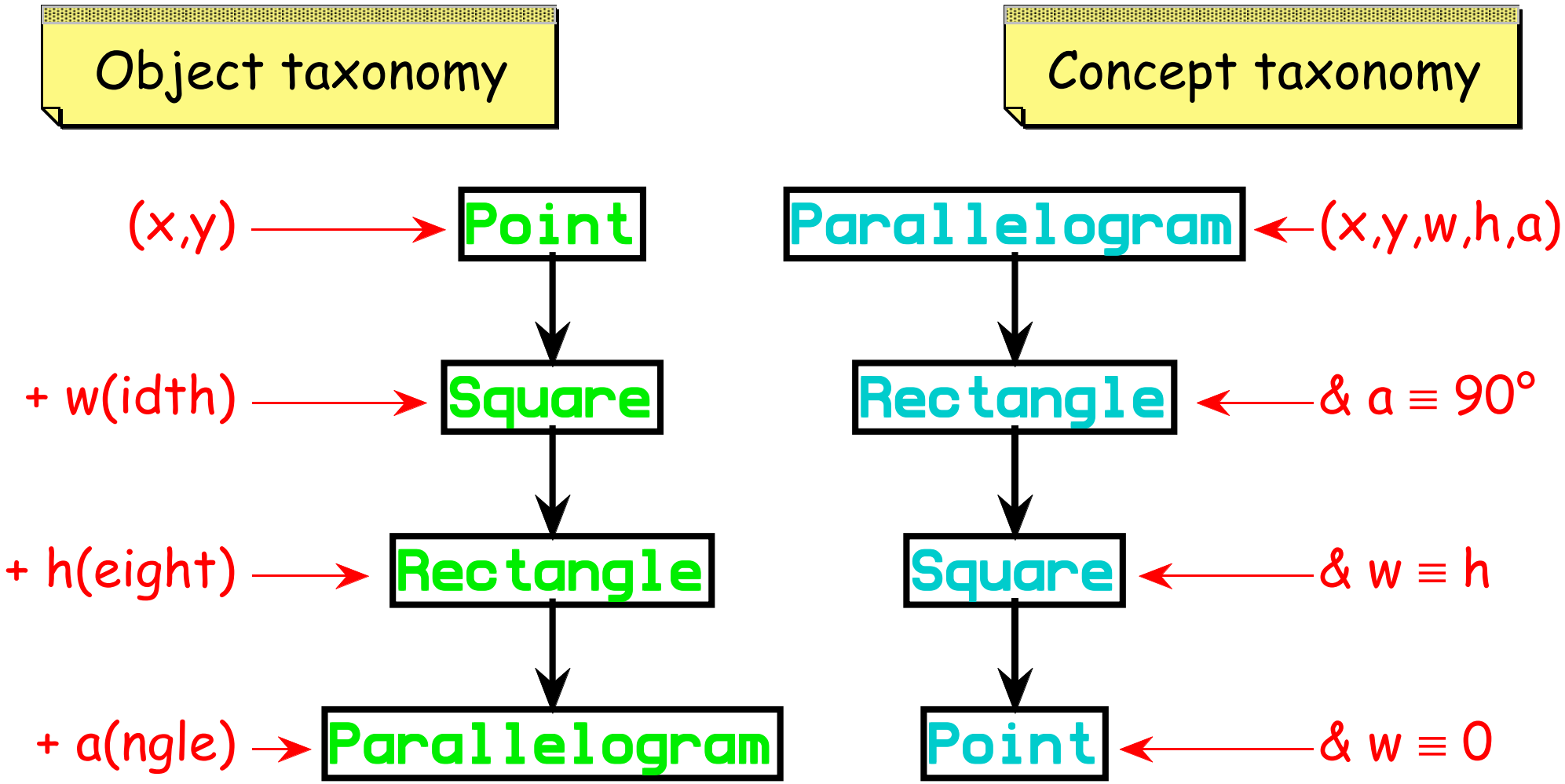
A framework approach to inheritance

Theo D'Hondt
Programming Technology Lab
[http://progwww.vub.ac.be]

—Contents—

- Origins
- Modular inheritance
- How does it work?
- What is it good for?
- Performance issues
- Current relevance

—Origins—


Object taxonomy
Concept taxonomy
(x,y)
Point
+ w(idth)
Square
+ h(eight)
Rectangle
+ a(ngle)
Parallelogram
Parallelogram
(x,y,w,h,a)
Rectangle
& a ≡ 90°
Square
& w ≡ h
Point
& w ≡ 0

—Origins—

```
public class Point
  { private int x = 0;
    private int y = 0;
    public void move(int X, int Y) { x = X; y = Y; }
    public double area() { return 0; }}

public class Square extends Point
  { protected int w = 0;
    public void widen(int W) { w = W; }
    public double area() { return w*w; }}

public class Rectangle extends Square
  { protected int h = 0;
    public void heighten(int H) { h = H; }
    public double area() { return w*h; }}

public class Parallelogram extends Rectangle
  { protected double a = 0;
    public void skew(double A) { a = A; }
    public double area() { return w*h*sin(a); }}
```

the conventional approach

Origins

```
public class Parallelogram
  { protected int    x = 0;
    protected int    y = 0;
    protected int    w = 0;
    protected int    h = 0;
    protected double a = 0;
    public void move(int X, int Y) { x = X; y = Y; }
    public void widen(int W) { w = W; }
    public void heighten(int H) { h = H; }
    public void skew(double A) { a = A; }
    public double area() { return w*h*sin(a); }}

public class Rectangle extends Parallelogram
  { public void skew(double A)
      { System.out.println( "cannot skew figure" ); }
    public double area() { return w*h; }}

public class Square extends Rectangle
  { public void heighten(int H)
      { System.out.println( "cannot heighten figure" ); }
    public double area() { return w*w; }}

public class Point extends Square
  { public void widen(int W)
      { System.out.println( "cannot widen figure" ); }
    public double area() { return 0; }}
```

the bad solution

—Origins—

```
[ Self makePoint
   μmethod:
    [ x μvar: 0;
      y μvar: 0;
      self setX: newX
       μmethod: [ x μ<- newX; self ];
      self setY: newY
       μmethod: [ y μ<- newY; self ];
      Self addSide
       μmethod:
        [ s μvar: 0;
          self setS: newS
           μmethod: [ s μ<- newS; self ];
          Self addSecondSide
           μmethod:
            [ ss μvar: 0;
              self setSS: newSS
               μmethod: [ ss μ<- newSS; self ];
              μself ];
          Self addAngle
           μmethod:
            [ a μvar: 0;
              self setA: newA
               μmethod: [ a μ<- newA; self ];
              μself ];
          μself ];
      μself ]
```

Agora:
mixin methods

—Modular inheritance—

tweaking semantics ...

```
Stack(n):
   { T[n]: void;
     t: 0;
     empty()::
       t = 0;
     full()::
       t = n;
     push(x)::
       { T[t:= t+1]:= x;
         self };
     pop()::
       { x: T[t];
         t:= t-1;
         x };
     clone() }
:<closure Stack>
S: Stack(10)
:<dictionary>
S.push(1)
:<dictionary>
T: S.clone()
:<dictionary>
T.push(2)
:<dictionary>
S.pop()
:1
```

... : ... = variable

... :: ... = constant

clone() = clone lexical environment

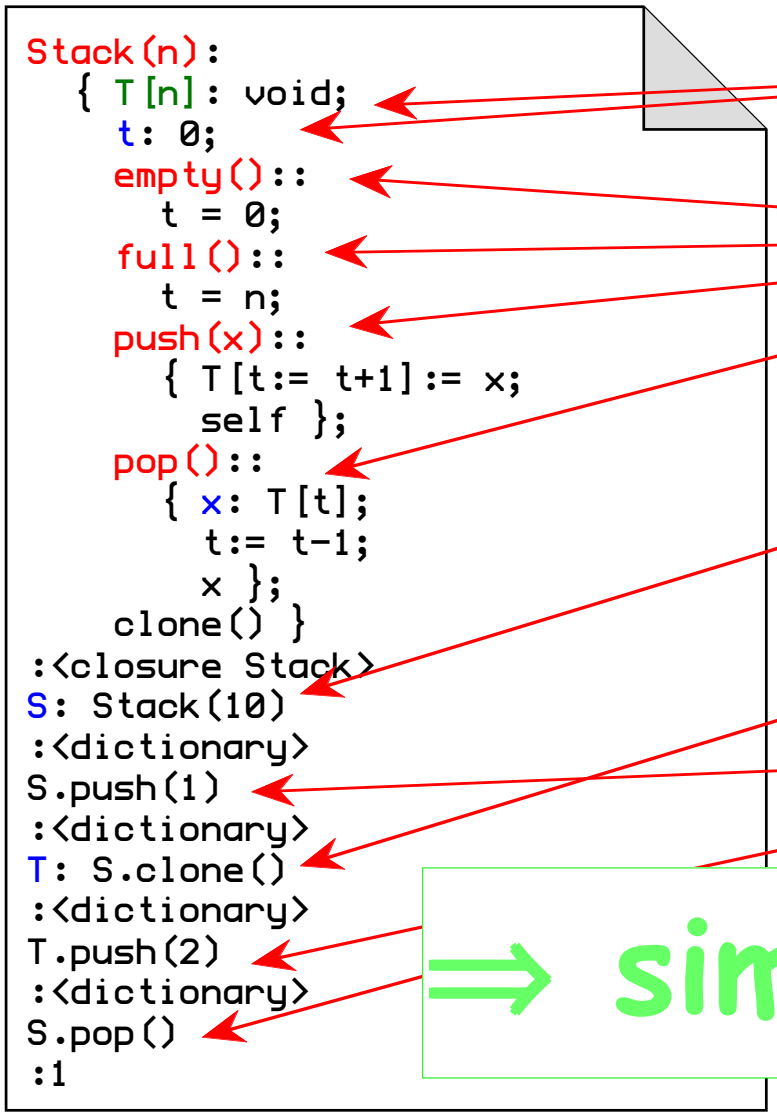
... . = evaluate within clone

red = function declaration

green = table declaration

blue = variable declaration

—Modular inheritance—



```
Stack(n):
  { T[n]: void;
    t: 0;
    empty()::
      t = 0;
    full()::
      t = n;
    push(x)::
      { T[t:= t+1]:= x;
        self };
    pop()::
      { x: T[t];
        t:= t-1;
        x };
    clone() }
:<closure Stack>
S: Stack(10)
:<dictionary>
S.push(1)
:<dictionary>
T: S.clone()
:<dictionary>
T.push(2)
:<dictionary>
S.pop()
:1
```

- instance variables
- instance methods
- prototype ceation
- object instantiation
- message passing

⇒ simple object model

—Modular inheritance—

```
Stack(n):
  { T[n]: void;
    t: 0;
    empty()::
      t = 0;
    full()::
      t = n;
    push(x)::
      { T[t:= t+1]:= x;
        self };
    pop()::
      { x: T[t];
        t:= t-1;
        x };
    makeProtected()::
      { push(x)::
          if(full(),
             error('overflow'),
             super.push(x));
        pop()::
          if(empty(),
             error('underflow'),
             super.pop());
        clone() };
    clone() }
```

mixin method

S: Stack(10)
prototype creation

T: S.clone()
object instantiation

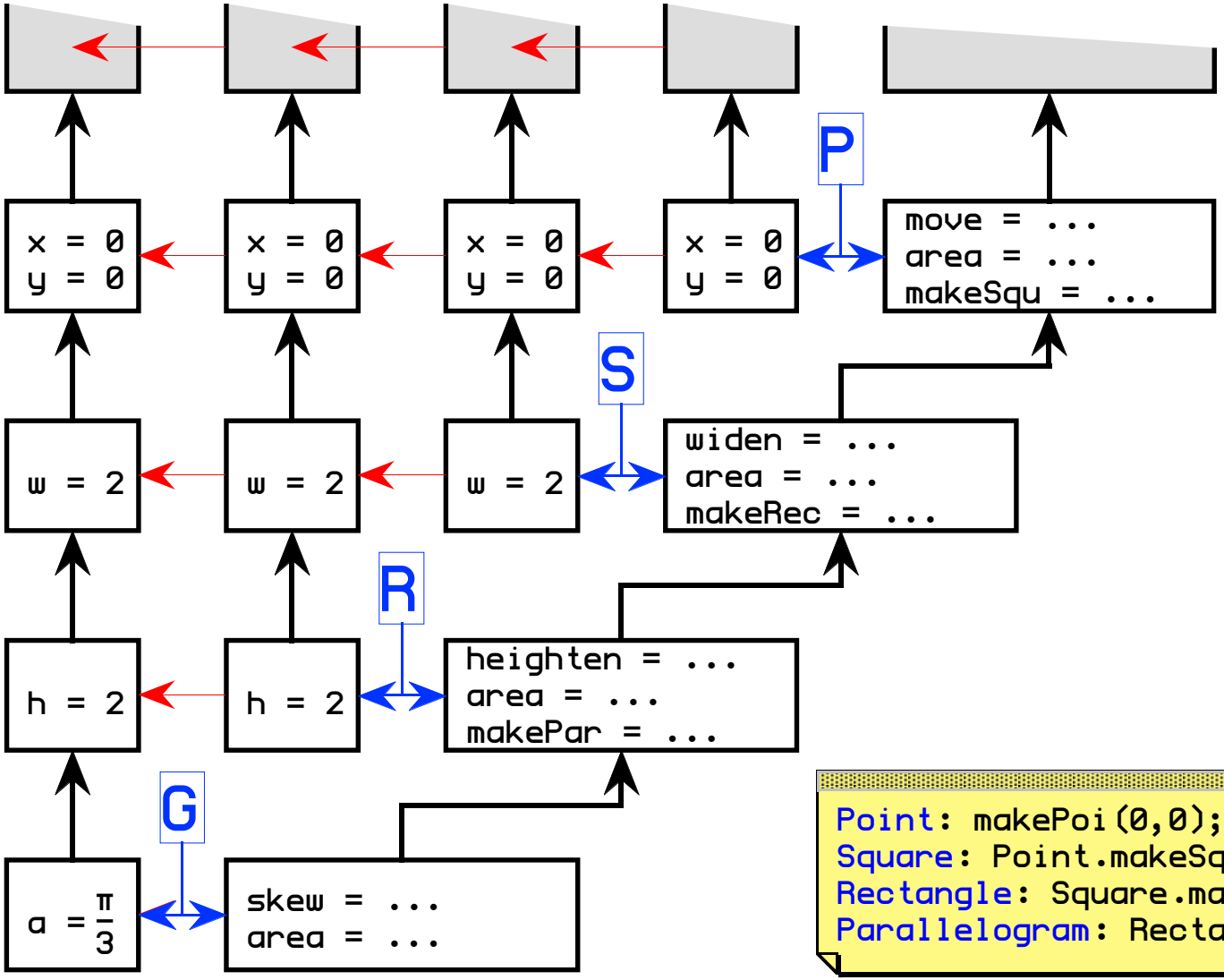
R: S.makeProtected()
"subclassing"

—Modular inheritance—

```
{ makePoi(x,y)::
    { move(X,Y)::
        { x := X; y := Y };
      area():: 0;
      makeSqu(w)::
        { widen(W)::
            w := W;
          area():: w^2;
          makeRec(h)::
            { heighten(H)::
                h := H;
              area():: w*h;
              makePar(a)::
                { skew(A)::
                    a := A;
                  area():: w*h*sin(a);
                  clone() };
              clone() };
          clone() };
      clone() };
  Point: makePoi(0,0);
  Square: Point.makeSqu(2);
  Rectangle: Square.makeRec(3);
  Parallelogram: Rectangle.makePar(3.1415926/3);
  Parallelogram.area() }
:5.19615
```

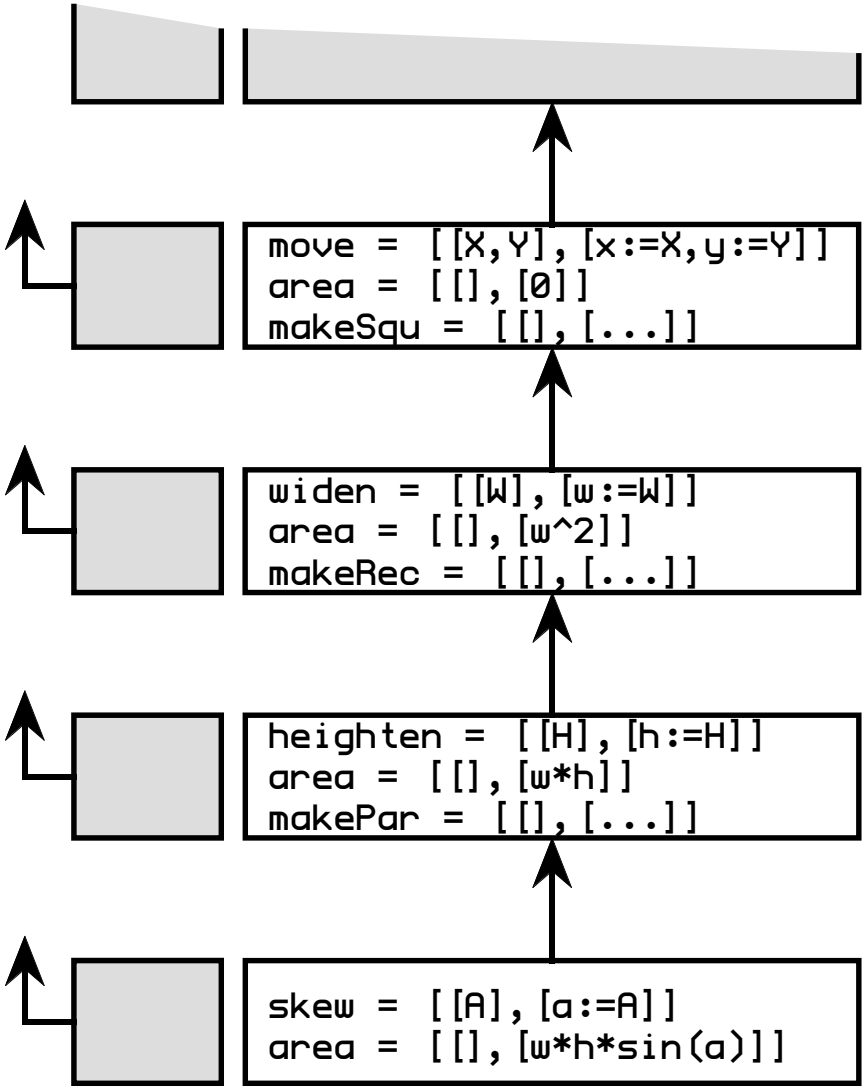
emulating class-based inheritance ...

—How does it work?—




```
Point: makePoi(0,0);
Square: Point.makeSqu(2);
Rectangle: Square.makeRec(3);
Parallelogram: Rectangle.makePar(3.1415926/3);
```

—How does it work?—



retrieving a method:

m: R.area

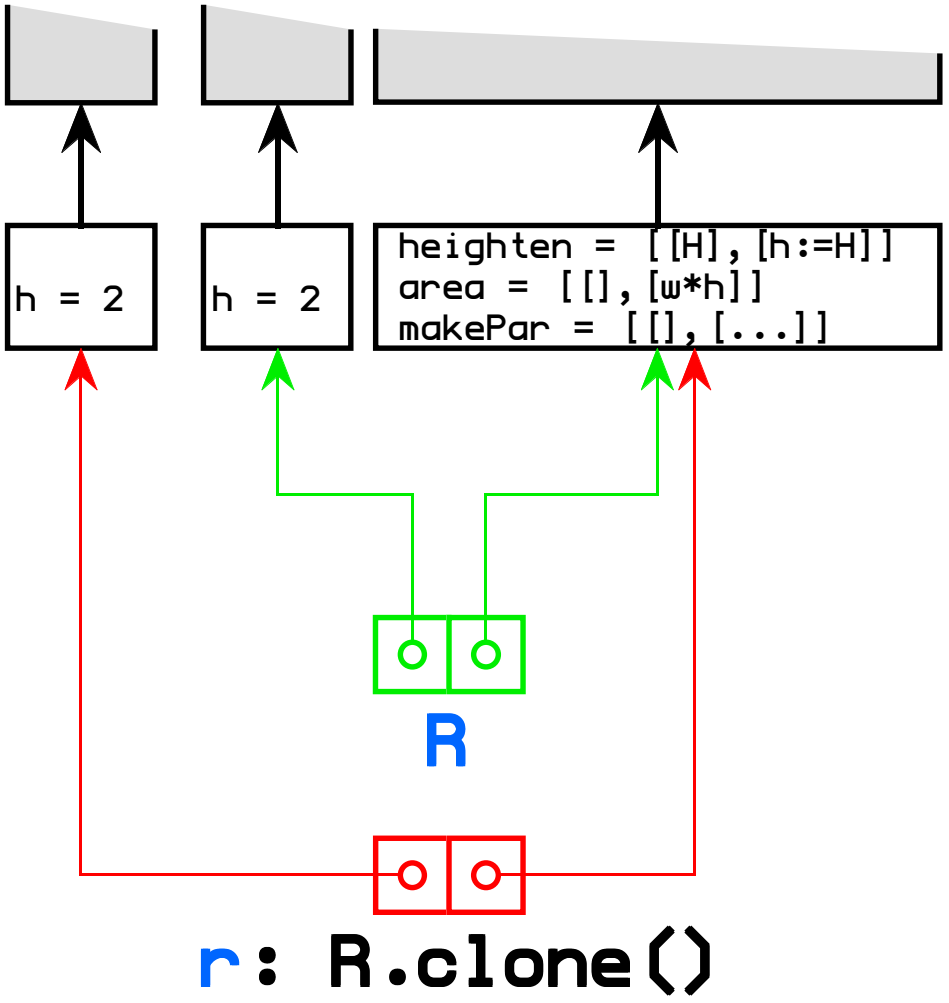
converts a function to a closure:

m = [[],[w*h],R]

⇒ first class methods!

controlled dynamic scope ...

—How does it work?—



cloning an object:

r: R.clone()

performs

- a shallow copy of the constant part
- a deep copy of the variable part

—What is it good for?—

```
{ makePoi(x,y)::
    { move(X,Y)::
        { x := X; y := Y };
      area():: 0;
      addW(w)::
        { widen(W)::
            w := W;
          area():: w^2;
          clone() };
      addH(h)::
        { heighten(H)::
            h := H;
          area():: w*h;
          clone() };
      addA(a)::
        { skew(A)::
            a := A;
          area():: w*h*sin(a);
          clone() };
      clone() };
  Point: makePoi(0,0);
  Square: Point.addW(2);
  Rectangle: Square.addH(3);
  Parallelogram: Rectangle.addA(3.1415926/3);
  Parallelogram.area() }
:5.19615
```

composing inheritance operators

```
{ makePoi(x,y)::
    { move(X,Y)::
        { x := X; y := Y };
      area():: 0;
      addW(w)::
        { widen(W)::
            w := W;
          area():: w^2;
          clone() };
      addH(h, skewed)::
        { heighten(H)::
            h := H;
          if(skewed,
             void,
             area():: w*h);
          clone() };
      addA(a)::
        { skew(A)::
            a := A;
          area():: w*h*sin(a);
          clone() };
      clone() };
  Point: makePoi(0,0);
  Square: Point.addW(2);
  Diamond: Square.addA(3.1415926/3);
  Rectangle: Square.addH(3, false);
  Parallelogram1: Rectangle.addA(3.1415926/3);
  Parallelogram2: Diamond.addH(3, true);
   [ Parallelogram1.area(), Parallelogram2.area() ] }
:[5.19615, 5.19615]
```

d for?—

conditional overriding

multiple inheritance?

—What is

basic mixins

generator for Rectangles

generator for Squares

overriding mixins

generic generator

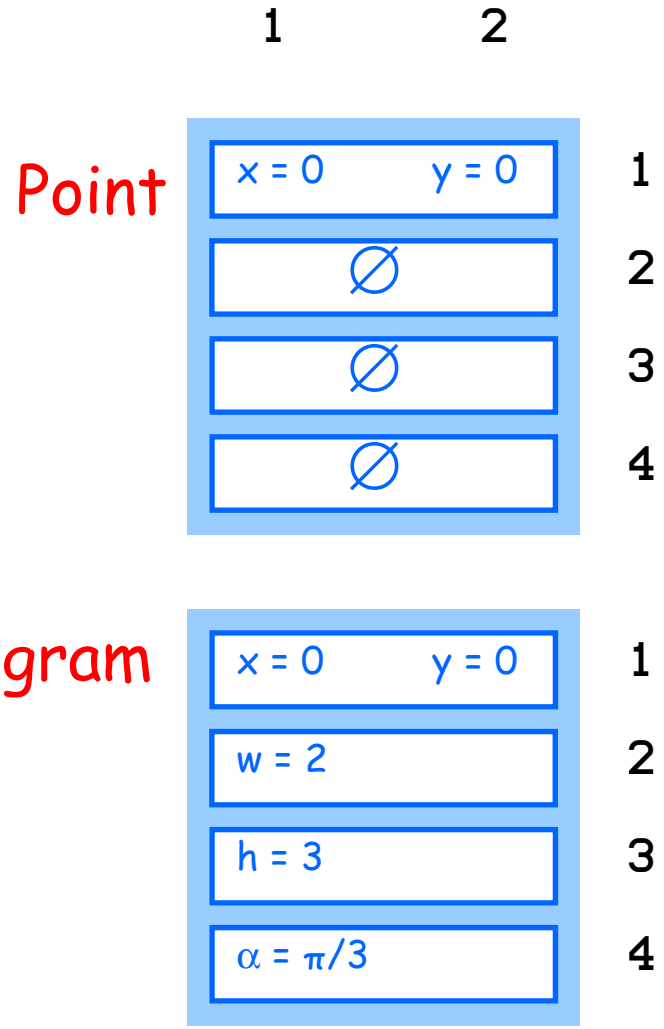
```
{ makePro():
    { addWidth()::
        { w: 0;
          widen(W):: w:= W;
          width():: w;
          clone() };
      addHeight()::
        { h: 0;
          heighten(H):: h:= H;
          height():: h;
          clone() };
      makeRec()::
        { R:: clone();
          R.make() };
      makeSqu()::
        { addHeight()::
            { heighten(H)::
                error("cannot heighten figure");
              height():: w;
              clone() };
          S::clone();
          S.make() };
      make()::
        { x:: self.addWidth();
          x.addHeight() };
      clone() };
  P: makePro();
  R: P.makeRec();
  S: P.makeSqu();
  S.widen(3);
  [ S.height(), S.width() ] }
:[3, 3]
```

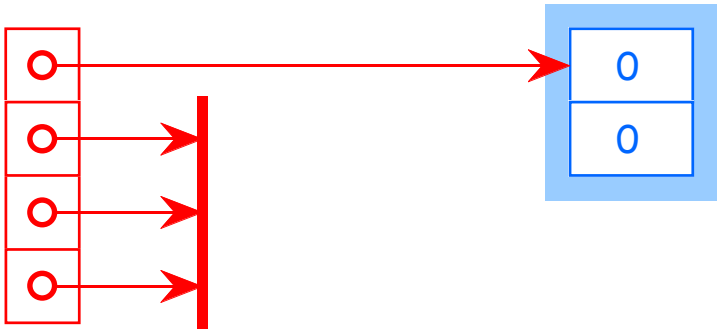
mixin framework

—What is it good for?—

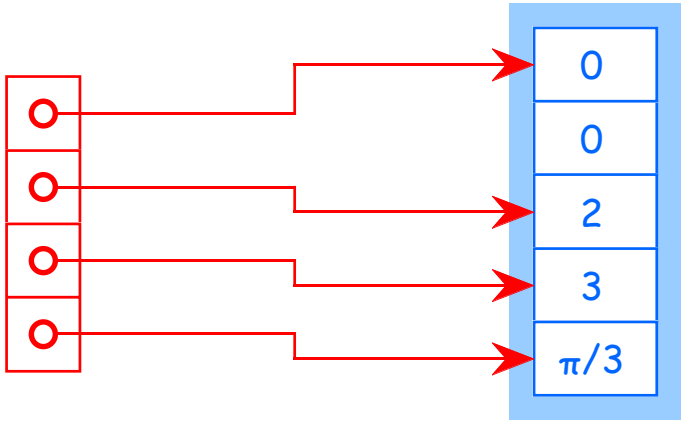
Modular inheritance
can be used to build inheritance frameworks that require inheritance operators to be composed of mixin methods (according to possibly complex computational rules) that may selectively be overriden ...

—Performance issues—





lexical addressing using displays



—Current relevance—

Reuse contracts

- decomposition (or: intentional composition) of component structure
- managed evolution of same

on the one hand

—Current relevance—

separation between
- class hierarchy
- type hierarchy

in practice available in Java?

on the other hand

—Current relevance—

```
interface Parallelogram
  { public void move(int X, int Y);
    public void widen(int W);
    public void heighten(int H);
    public void skew(double A);
    public double area(); }

interface Rectangle extends Parallelogram { }

interface Square extends Rectangle { }

interface Point extends Square { }
```

```
Parallelogram P = new cPoint();
Parallelogram S = new cSquare();
Parallelogram R = new cRectangle();
Parallelogram G = new cParallelogram();
```

the good approach

Current relevance

```
public class cPoint implements Point
  { private int x = 0;
    private int y = 0;
    public void move(int X, int Y) { x = X; y = Y; }
    public void widen(int W)
      { System.out.println( "cannot widen figure" ); }
    public void heighten(int H)
      { System.out.println( "cannot heighten figure" ); }
    public void skew(double A)
      { System.out.println( "cannot skew figure" ); }
    public double area() { return 0; }}

public class cSquare extends cPoint implements Square
  { protected int w = 0;
    public void widen(int W) { w = W; }
    public double area() { return w*w; }}

public class cRectangle extends cSquare implements Rectangle
  { protected int h = 0;
    public void heighten(int H) { h = H; }
    public double area() { return w*h; }}

public class cParallelogram extends cRectangle implements Parallelogram
  { protected double a = 0;
    public void skew(double A) { a = A; }
    public double area() { return w*h*sin(a); }}
```

the good approach - cont'd

—Current relevance—

Modular inheritance:

can be used as supporting technology to maintain a causal link between the implementation (e.g. class hierarchy) and the concept (e.g. interface hierarchy)

Reuse contracts might be used as a methodology to define the boundaries of inheritance (de)composition

—Current relevance—

Modular inheritance can be supported:

- as a methodology
- as a preprocessor
- as a language feature

—Conclusion—