Constructors - Cont.

1. Constructors can be *overloaded*!
   must be distinguished by the number or type of their arguments.
2. When no constructor is defined, there is a *default constructor* with no arguments.
   You cannot use the default constructor once one has been defined in your program. (**bad style!**)
3. When declaring a type, it is also possible to define *default values* for each field:

```
class Point {
 double latitude = 90.0;
 double longitude = 0.0;
 double altitude = 0.0;}
```

The Semantics of Records

To cover records:

we need to add a fifth argument to the functions Σ and Θ:

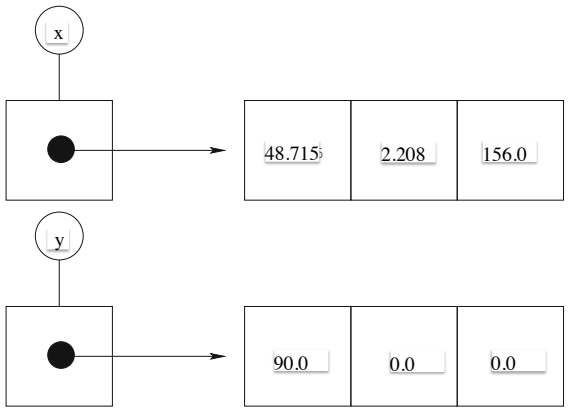
the list of constructed types,

each type being associated to an ordered pair composed:

$T \mapsto$ (list of fields, list of constructors)

Sharing

In the first case (sharing), all changes of the cell associated with x automatically change the cell associated with y, and vice versa.

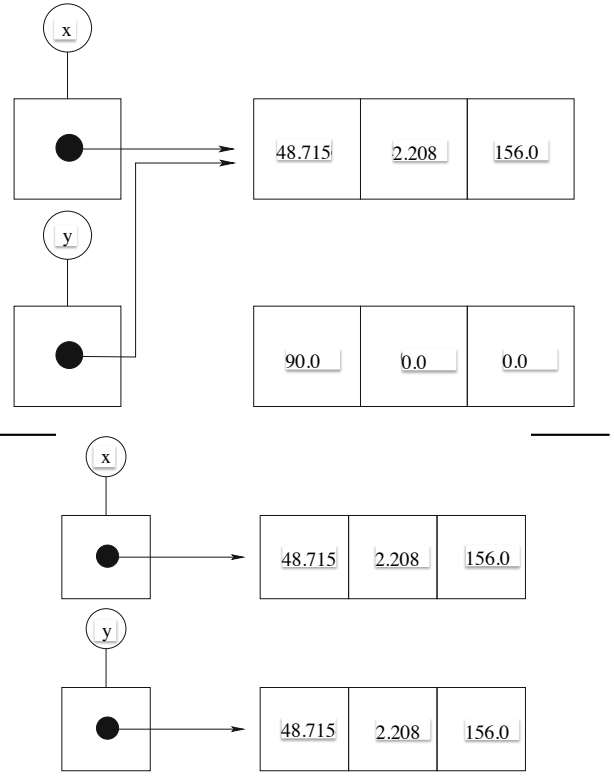


x = y (Sharing!)

y,lat = x.lat

y.long = x.long

y.alt = x.alt



Equality

```
a = new Point(1, 2, 3);
b = new Point(1, 2, 3);
```

Two types of equality:

- *Physical.* Two records of the same type are physically equivalent (a==b) only when they are identical (share the same cell).
  Hence, a == b is false.
- *Structural.* Two records of the same type are structural equivalent when their field's value are equal. Hence, a and b are structural equivalent.

Wrapper Types (from Base type to Object type)

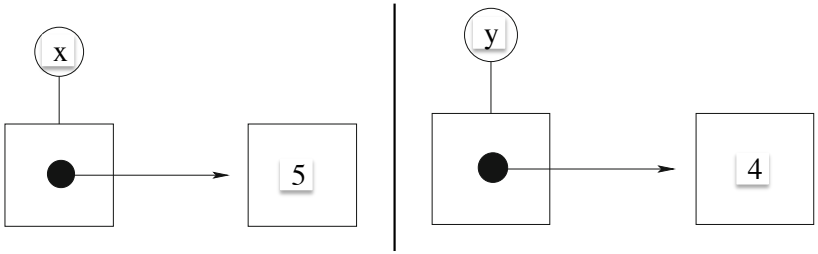
A *wrapper* is a type of record with one lone field.

```
class Integer {
 int c;
 Integer (int x) { this.c = x; } }
```

The Points (allows several variables to share a single value.):

1. Uniformity
2. Sharing

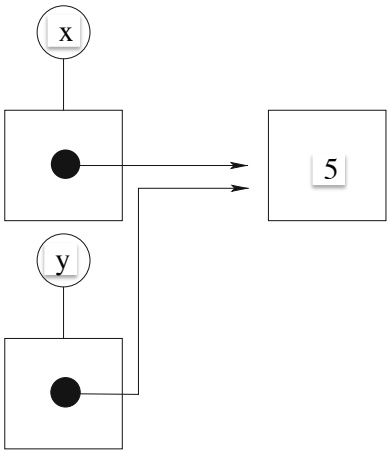
Wrapper Types - Cont.




```
Integer x = new Integer(4);
Integer y = new Integer(x.c);
x.c = 5;
System.out.println(y.c);
print 4
```

```
int x = 4;
int y = x;
x = 5;
System.out.println(y);
print 4
```

If we replace the second line with "Integer y = x" then it prints 5



Wrapper Types - Cont.

The following function swaps values of x and y.

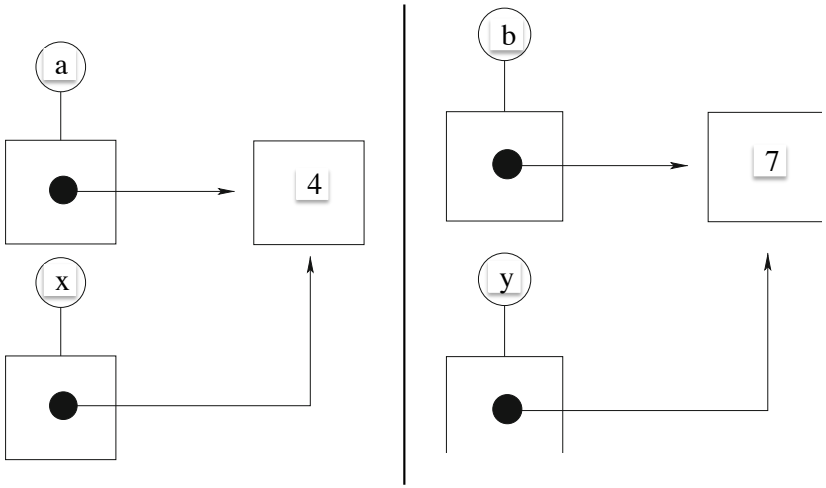
```
static void swap (Integer x, Integer y) {
 int temp = x.c;
 x.c = y.c;
 y.c = temp;}
```

In java, it is not possible to do so with arguments of type “int”. The following functions do nothing on x and y.

```
static void swp (int x, int y) {
 int temp = x;
 x = y;
 y = temp;}
```

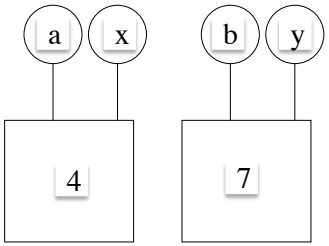
Wrapper Types - Cont.

When we call the function swap(a, b), we create the state:



and so swapping the contents of x and y will also swap those of a and b.

If a, b are constant (final), and x, y too, then:



and swap works too.

Records in Caml

1. Record Type:

```
type point = {
 latitude : double;
 longitude : double;
 altitude : double;}
```

2. Creating:

```
let x = {lat = 90.0; longi = 0.0; altitude = 0.0;} or
let y = ref{lat = 90.0; longi = 0.0; altitude = 0.0;}
```

3. Accessing:

```
 x.latitude
(!y).latitude
```

4. Assigning: all fields are constant by default!

```
type int-wrap = {mutable c: int}
let x = {c = ref 5}
x.c <- 4
```

Records in C

1. Record Type:

```
struct Point {
 double latitude;
 double longitude;
 double altitude;};
```

2. Creating:

```
struct Point x; or
struct Point x = {5.0,7.0,100.0};
```

3. Accessing:

```
x.latitude
```

4. Assigning: all fields are constant by default!

```
x.latitude = 12.0
```

   Call by copy!
   (the value of x is not a reference associated with a record in memory as it is in Java and Caml!)

Arrays

- Array Types:
  In Java, an array with elements of type T is of type T[ ].
  int [ ] t;
  This adds t, a new reference r to the environment.
  The default value is *null*.
  In java, the box r can only contain *null* or *another reference*.
- Allocation of an Array:
  To allocate, you create a box large enough:
  new int [10];
  This creates a new reference r' pointing to an n-tuple (10-tuple in this case) of default values (0 in this case). The fields are numbered from 0 to n-1.
  i.e., int [ ] t = new int [10]; $\rightsquigarrow$
  env: $[t = r]$ mem: $[r = r', r' = [\underbrace{0, \cdots, 0}_{10}]]$:

Arrays - Cont.

- Accessing:
  t[k]
  gives the k^{th} field.
- Creation:

```
int[] t1;
t1 = new int[]{10,11,12,13,14,15,16,17};
int[] t2 = new int[]{20,21,22,23,24,25,26,27};
char[] ca = {'J','a','v','a'};
```

  t1[5] $\rightsquigarrow$ 15, t2[5] $\rightsquigarrow$ 25, ca[0] $\rightsquigarrow$ 'J'

- Assigning:

```
ca[0] = 'j';
```

(Arrays in C and Caml are essentially the same)

Arrays of Arrays

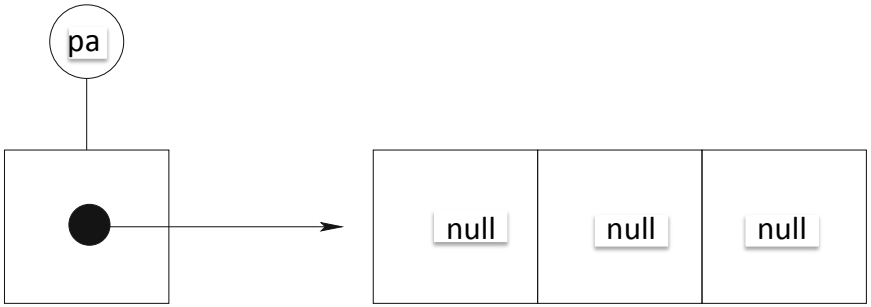
(T [ ])[ ] is an *array of arrays*.

```
int[][] t = new int[20][15];
```

$t[\underbrace{i}_{0\cdots19}][\underbrace{j}_{0\cdots14}]$

Arrays of Objects

```
Point[] pa = new Point[4];
```



Usual looping construct:

```
for(int i=0; i<pa.length; i++) { ... (refer to pa[i])}
```

Better!

```
for(Point p: pa) { ... (refer to p) }
```

Dynamic Data Types - Recursive Records

Example:

```
class List {
 int hd;
 List tl;}
```

Note: $int \times List$ is only non-empty lists

but $List \simeq \{\text{empty list}\} \uplus (int \times List)$

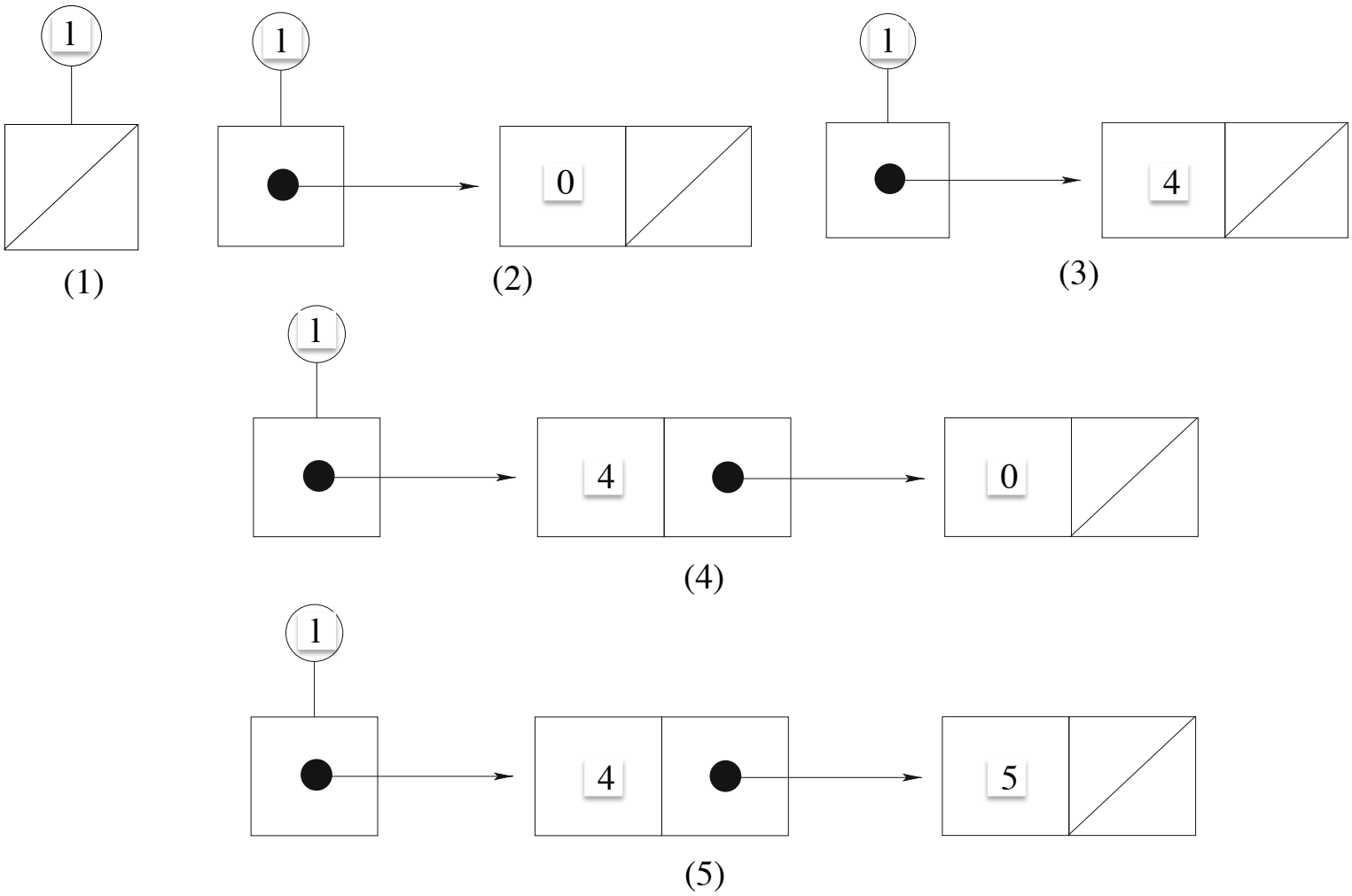
Observe: List is a record and its value is a reference, a reference can be null.

So, $List \simeq \{\text{null}\} \uplus (int \times List)$

Recursive Records - Example

```
List l;
l = new List();
l.hd = 4;
l.tl = new List();
l.tl.hd = 5;
l.tl.tl = null;
```

Recursive Records - Example - Cont.



Recursive Records - Constructor

```
List (final int x, final List y) {this.hd = x; this.tl = y;}
List l = new List(4,new List(5,null));
```

Recursive Definitions and Fixed Point Equations

$$\text{“}List \simeq \{\text{null}\} \uplus int \times List\text{”}$$

is not a definition but an equation to be solved!

How? To construct a solution we proceed by successive approximations:

Let i be the number step of approximation and L_i the solution at the step i.

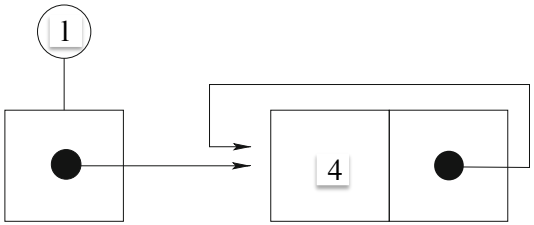
$$L_0 = \emptyset$$
$$L_1 = \{null\} \uplus (int \times L_0) = \{null\}$$
$$L_2 = \{null\} \uplus (int \times L_1) = \{null\} \uplus (int \times \{null\})$$
$$L_3 = \{null\} \uplus (int \times L_2) = \{null\} \uplus (int \times \{null\}) \uplus (int \times (int \times \{null\}))$$
$$\vdots$$
$$List := \lim_{i \to \infty} L_i$$

Infinite Values

The type List contains values that cannot be constructed in a finite number of steps. For example:

```
List l = new List();
l.hd = 4;
l.tl = l;
```

Creates the following list:



The following program terminates properly if you apply it to finite lists, but enters an infinite loop if you try to apply it to infinite lists.

```
static int sum (final List l) {
 if (l == null) return 0;
 return l.hd + sum(l.tl);}
```

Type Algebra

0: empty
1: unit
$+$: disjoint some
$*$: product
X: variable, "some types"
$+$: disjoint some
$\mu X.F(x)$: least-fixed point, solve $x \simeq F(x)$

Bool $:=$ false $\uplus$ true $= 1 + 1 \simeq 2$
$A + 0 \simeq 0 + A \simeq A$
$A.1 \simeq 1.A \simeq A$
$A + B \simeq B + A$
$A.B \simeq B.A$
$A.(B + C) \simeq A.B + A.C$
$A + (B + C) \simeq (A + B) + C$

Type Calculus

$\delta_x A$

$\delta_x 1 = 0$

$\delta_x(A + B) = \delta_x A + \delta_x B$

$\delta_x(A \,.\, B) = \delta_x A \,.\, \delta_x B$

$\delta_x A =$ type of one context of A.

$\delta_x L = L \,.\, L.$

$\delta_x(\frac{1}{1-x}) = (\frac{1}{1-x})^2$

Solve $\delta_x Y(X) = Y(X)$? $\quad Y(X) = \mathrm{Bag}(X)$