

Révisions : valeurs, variables, . . . , Listes

Luc.Maranget@inria.fr

[http://www.enseignement.polytechnique.fr/profs/
informatique/Luc.Maranget/421/](http://www.enseignement.polytechnique.fr/profs/informatique/Luc.Maranget/421/)

Objectifs

- ▶ Structures de données « dynamiques ».
 - ▷ Listes, tables de hachage, arbres...
 - ▷ Et leurs algorithmes traditionnels, (recherches par exemple).

- ▶ Perfectionnement en programmation.
 - ▷ Programmer c'est comprendre (vraiment).
 - ▷ Programmation des structures de données récursives.
 - ▷ Choix des structures de données, (exemples « realistes »)

Organisation du cours

- ▶ 9 blocs, soit 9 vendredis.
 - ▷ Le matin, amphi, de 10h30 à 12h00.
 - ▷ L'après-midi, TP.
- ▶ Évaluation.
 - ▷ TP noté, — le cinquième, 9 DEC, non le 7 DEC.
 - ▷ Contrôle classant, à la fin, 25 JAN.
 - ▷ Note de module :

$$\frac{3 * CC + \max(CC, HC - k)}{4} \rightarrow \text{lettre}$$

Quelques éléments de Java

- ▶ Qu'est-ce au juste qu'une valeur ?
 - ▷ Scalaires
 - ▷ Objets
- ▶ Qu'est-ce au juste qu'une variable ?
 - ▷ Création et initialisation.
 - ▷ Appel de méthode
 - ▷ Portée
- ▶ Allocation dynamique
 - ▷ Les tableaux.
 - ▷ Les paires.
 - ▷ Les listes.

Les valeurs

- ▶ Les scalaires.
 - ▷ Les booléens : **boolean** (**true** et **false**).
 - ▷ Les entiers : **byte** (modulo 2^8), **char** et **short** (modulo 2^{16}), **int** (modulo 2^{32}), **long** (modulo 2^{64}).
 - ▷ Les flottants : **float** (simple précision), **double** (double précision).

- ▶ Les objets : tout le reste ! Les objets appartiennent à des *classes* qui sont plus ou moins leur type.
 - ▷ Les un peu spéciaux : `String`, les tableaux...
 - ▷ Les objets de classes de la librairie : par ex. `System.out` de la classe `PrintStreamWriter`.
 - ▷ Ceux que l'on fait soi-même (en définissant leur classe et par **new**).

Déclaration: avec et sans initialisation

```
int x = 1 ;
```

- ▶ « **int x** » est la *déclaration* : une variable (qui contient un **int**) est créée.
- ▶ « = 1 » est l'initialisation : la valeur initiale de la variable est 1.

Ne pas confondre déclaration avec initialisation et *affectation*.

Même si la syntaxe est très semblable.

```
int x = 1 ; // Création avec initialisation.  
x = 2 ;    // Changer le contenu  
int y ;    // Création tout court
```

Remarque : Quelle est la valeur initiale de la variable y ?

On ne peut pas le savoir !

Variables locales

Les variables locales sont :

- ▶ Déclarées dans le corps des méthodes,
- ▶ Ou bien ce sont les paramètres des méthodes.

Le compilateur vérifie « l'initialisation » des variables locales.

```
class T {  
    static int f(int x) {  
        int y ;  
        if (x != 0) y = 1 ;  
        return x + y ;  
    }  
}
```

```
# javac T.java
```

```
T.java:5: variable y might not have been initialized
```

Les variables sont « des cases »

Deux déclarations de variable.

`int x=1, y ;` x

1

 y

?

Une affectation.

`y = x ;` x

1

 y

1

On remarque que :

- ▶ À gauche du =, une variable → une case.
- ▶ À droite du =, une variable → le contenu d'une case.

Appel de méthode

Le principe est le suivant :

- ▶ Chaque *appel* de methode crée ses propres variables.
- ▶ Les variables sont initialisés par l'appel.

```
static int f(int x) {  
    x = x + 2 ;  
    return x ;  
}
```

```
public static void main (String [] arg) {  
    int x = 2 ;  
    int r = f(x) ;  
    System.out.println("f(" + x + ") = " + r) ;  
}
```

Résultat : $f(2) = 4$

Avec des boites

```
static int f(int x) {  
    x = x + 2 ;  
    return x ;  
}
```

Au début de l'appel de méthode :

x(de m) 2 x(de f) 2

Juste avant le return :

x(de m) 2 x(de f) 4

Portée

La portée d'une variable est la zone du code où on peut l'utiliser, autrement dit, la zone de visibilité.

Dans l'exemple précédent, les portées des deux variables `x` étaient clairement limitées aux corps des méthodes `main` et `f`.

La portée est structurée selon les *blocs*, en gros délimités par `{...}`.

Quelques règles importantes :

- ▶ La portée s'étend de la déclaration de la variable à la fin du bloc de déclaration.
- ▶ Dans un même bloc, il ne peut pas y avoir deux variables de même nom.
- ▶ L'usage d'une variable fait référence à la déclaration la plus proche vers le haut du programme (liaison lexicale).

Exemple de portée I

Que penser de :

```
static int f() {  
    for (int i = 0 ; i < 10 ; i++) { ... }  
    return i ;  
}
```

Ce code est incorrect, car dans l'instruction **return** *i*, la variable *i* ne fait référence à aucune déclaration visible.

Note : On peut considérer que la boucle **for** est plus ou moins équivalente à :

```
{ int i = 0 ; // NB. un bloc est ouvert  
    while (i < 10) { ...  
        i++ ;  
    }  
}
```

Portée, II

Que se passe-t-il si on appelle `f(4)` ?

```
static int f(int x) {  
    { int x = 2 ;  
      System.out.print("x = " + x) ;  
    }  
    System.out.println(", x = " + x) ;  
}
```

Ce programme est correct. Il affiche :

`x = 2, x = 4`

Il y a en fait deux variables, `x` et `x`.

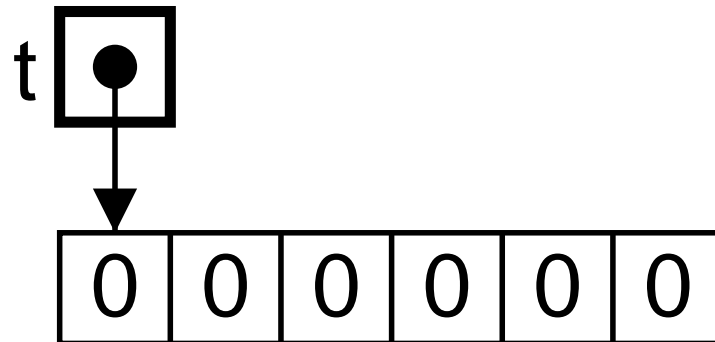
On peut très bien changer `x` (ou `x`) en `y`, sans rien changer au fond. Les variables (locales) sont un peu comme les variables muettes des mathématiques.

Un tableau avec les boites

Un tableau de taille n est (plus ou moins) une grosse case composée de n cases « normales ».

```
int [] t = new int[6] ;
```

Mais la valeur du tableau est une flèche (référence, pointeur, adresse) vers la grosse case.



Note : Les cases du tableau sont initialisés par défaut.

Valeur par défaut

Les cases des tableaux ont une valeur par défaut.

- ▶ **Scalaire** : une espèce de zéro.
 - ▷ **boolean** : c'est... **false** .
 - ▷ **Autres scalaires** : c'est bien zéro.
- ▶ **Objets** : **null**.

Notons que **null** est un objet un peu spécial. Il appartient à toutes les classes, mais on ne peut rien faire avec, sauf :

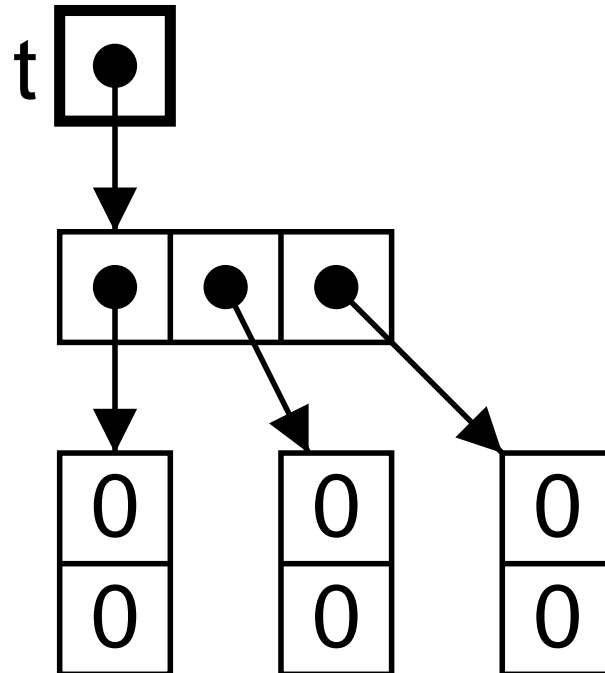
- ▶ L'affecter, par ex. `String s = null`
- ▶ Le comparer, par ex. `if (s != null)...`

La deuxième dimension

Une matrice à 3 lignes et 2 colonnes.

```
int [] [] t = new int [3] [2] ;
```

Est en fait un tableau de trois tableaux de deux entiers.

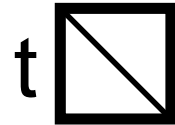


Et null dans tout ça ?

Logiquement, les tableaux sont initialisés par défaut à **null**.

```
int [] t // = null ;
```

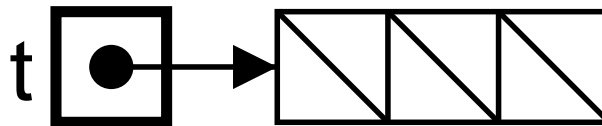
La valeur **null** remplace une flèche, mais ne pointe nulle part.



L'initialisation par défaut vaut aussi pour les cases de tableau.

```
int [][] t = new int [3] [] ; // tableau de tableaux
```

Soit : un tableau de trois **null**.



Les tableaux sont alloués « dynamiquement »

Cela veut dire que leur place est calculée/allouée lors de l'exécution.

```
static int [] f(int n, int x0) {  
    int [] r = new int [n] ;  
    for (int i = 0 ; i < n ; i++) { r[i] = x0 ; }  
    return r ;  
}
```

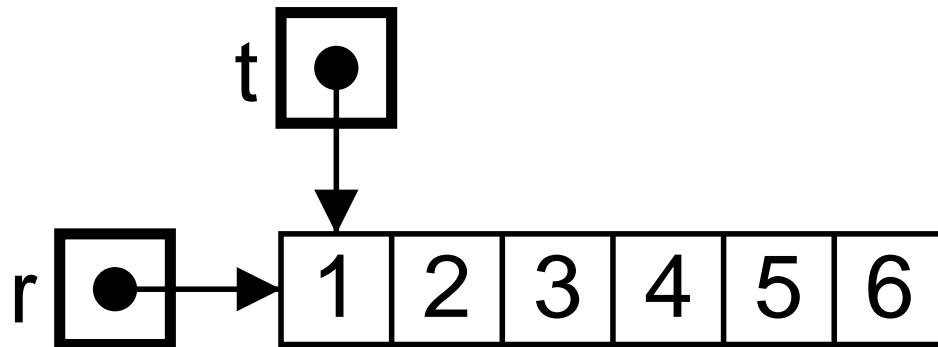
En outre, l'allocation est effectuée dans le « tas ».

- ▶ Le tableau existe encore après le retour de la méthode `f` (contrairement à la variable `r`).
- ▶ Il n'y a pas lieu de libérer explicitement la place occupée lorsque le tableau devient inutile (GC).

Les valeurs des tableaux sont des flèches

```
int [] t = {1, 2, 3, 4, 5, 6} ; // Initialisation explicite  
int [] r = t ;
```

Dans `r = t` c'est la flèche qui est copiée.



De sorte que ici `t[2]` et `r[2]` désignent la même case (la troisième).

```
t[2] = 0 ; // ranger 0 dans la case désignée  
System.out.println(r[2]) ; // Affiche '0'
```

À comparer avec

La copie du contenu de variables de type scalaire.

```
int x = 3, y = x ;
```

x **3** y **3**

```
x = 0 ; System.out.println(y) ; // Affiche '3'
```

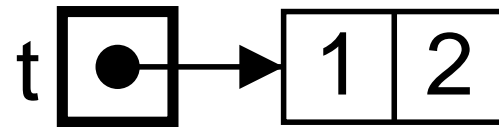
x **0** y **3**

Car ici x et y sont deux cases différentes.

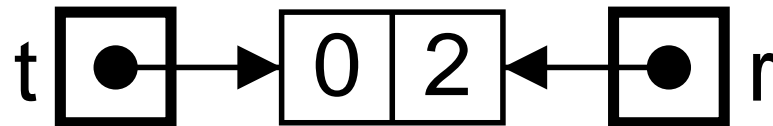
Appel de méthode

Les valeurs sont bien copiées, mais ce sont des flèches.

```
static void f(int [] r) { r[0] = 0 ; }
```



```
int [] t = { 1, 2 } ;  
System.out.println(t[0]) ; // Affiche '1'  
f(t) ;  
System.out.println(t[0]) ; // Affiche '0'
```



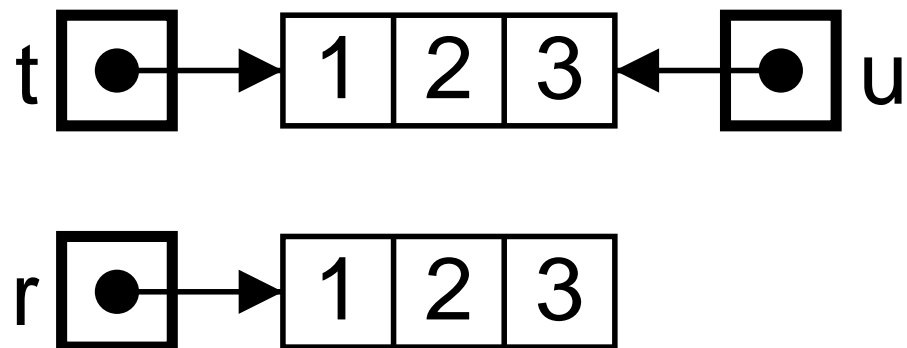
Égalité « physique »

L'égalité `==` est l'égalité des valeurs.

- ▶ Pour les scalaires aucun problème !
- ▶ Pour les objets, attention ! Les valeurs sont les flèches.

```
int [] t = {1, 2, 3} , r = {1, 2, 3} ; // Deux variables.  
int [] u = t ;  
System.out.println((t == r) + " , " + (t == u)) ;
```

Le résultat est **false** , **true**, car



Égalité « structurelle »

Pour savoir si deux tableaux sont identiques (et non plus exactement le même). On utilise `o1.equals(o2)`.

```
int [] t = {1, 2, 3} ;  
boolean b = t.equals(new int [] {1, 2, 3}) // tableau « anonyme »  
System.out.println(b) ;
```

Affiche **true**.

Résumé :

- ▶ L'égalité physique `==` expose l'implémentation.
- ▶ L'égalité structurelle est plus « mathématique ».

Les chaînes sont des objets...

Mais des objets un peu particuliers.

```
class Coucou {  
    public static void main(String [] arg) {  
        System.out.println("coucou" == "coucou") ;  
        System.out.println(arg[0] == arg[1]) ;  
    }  
}
```

La commande « java Coucou coucou coucou » affichera :

true

false

Un conseil donc, utiliser `equals`, même sur les chaînes.

Affichage

Nous savons afficher des scalaires et des chaînes.

```
System.out.println(1 + " " + true + ...)
```

Mais savons nous afficher des « objets » ?

```
int [] t = {1, 2} ;  
System.out.println(t) ;
```

Affiche : [I@a16869, c'est la flèche (l'adresse) qui est affichée !

Pour afficher un tableau, il faut écrire une boucle :

```
static void print(PrintStream out, int [] t) {  
    for (int i = 0 ; i < t.length ; i++) {  
        out.print(" " + t[i]) ;  
    }  
} // Appel : print(System.err, new [] {1, 2}) ;
```

Fabriquons nos propres structures de données

Par exemple une paire (de deux **int**):

```
class Pair {  
    int x, y ; // Variables d'instance ou champs  
  
    Pair (int x0, int y0) { // Constructeur  
        x = x0 ; y = y0 ;  
    }  
}
```

La classe ci dessus est un patron pour créer des objets, par :

```
Pair p = new Pair (1, 2) ;
```

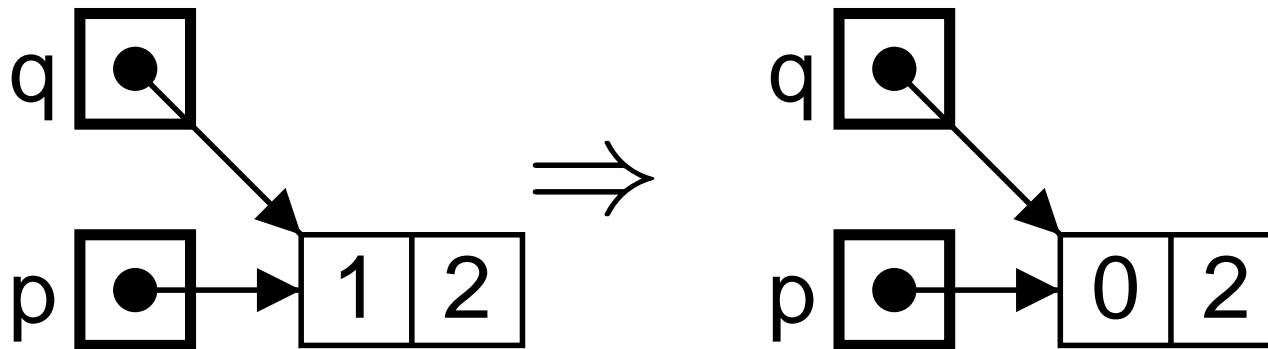
Important : L'écriture avec constructeur est meilleure que :

```
Pair p = new Pair () ; // Constructeur par défaut  
p.x = 1 ; p.y = 2 ;
```

Les valeurs des objets sont des références

Et tout ce qui s'appliquait au tableaux reste vrai.

```
Pair p = new Pair (1, 2) ;  
Pair q = p ;  
q.x = 0 ;  
System.out.println(p.x) ; // Affiche '0'
```



Comment afficher les paires

Ou début de programmation objet.

- ▶ Si `out` est un `PrintStream`, et `o` un objet, le code `out.print(o)`, revient à afficher la chaîne renvoyée par l'appel de méthode `o.toString()`.

- ▶ Par défaut, `o.toString()` affiche la valeur de `o` (une flèche).

```
out.print(new Pair(1,2)) ⇒ Pair@6f0472
```

- ▶ Mais on peut *redéfinir* la méthode `toString` (dans la classe `Pair`).

```
public String toString() { // pas de static
    return "[x=" + x + ",y=" + y + "]" ;
}
```

Dès lors,

```
out.print(new Pair(1, 2)) ⇒ [x=1,y=2]
```

Résumé sur les classes

Une classe C « pour faire des objets o » comprend :

- ▶ Un (ou des ou pas du tout) constructeur homonyme au nom de la classe. On crée un objet o par :

new $C(\dots)$

- ▶ Des (ou une ou pas du tout) variables x , que l'on réfère par :

$o.x$

- ▶ Des (ou une ou pas du tout) méthodes f , que l'on appelle par :

$o.f(\dots)$

Plus compliqué :

- ▶ Pour le moment : **static** nulle part.
- ▶ L'objet peut posséder des méthodes pré-existantes, qui peuvent être redéfinies.

Surcharge (*Overloading*)

Il est possible de définir plusieurs constructeurs, à condition que leurs *arguments* soient de types distincts.

```
class Pair {
    int x, y ;

    Pair() { // Constructeur de l'origine
        x = 0 ; y = 0 ; // Autant insister (init. par défaut)
    }

    Pair (int x0, int y0) {
        x = x0 ; y = y0 ;
    }
}
```

- ▶ Cela fonctionne aussi pour les méthodes.
- ▶ Tout ce passe comme si le type des arguments faisait partie du nom des constructeurs/méthodes.

Visibilité des variables d'instance

La règle des blocs semble rester valable :

```
class Pair {  
    int x, y ; // Déclaration  
  
    Pair (int x0, int y0) {  
        x = x0 ; y = y0 ;  
    }  
  
    public String toString() {  
        return "(" + x + ", " + y + ")" ;  
    }  
}
```

Visibilité des variables d'instance II

Mais en fait, c'est un peu différent, la portée des variables d'instance s'étend sur l'ensemble de l'objet.

```
class Pair {  
    Pair (int x0, int y0) {  
        x = x0 ; y = y0 ;  
    }  
    ...  
    int x, y ; // Déclaration  
}
```

En outre, on a aussi droit à `p.x` à peu près n'importe où... (à condition de ne pas spécifier `private int x`)

Visibilité des variables d'instance III

En fait la notation des variables d'instance comme des variables normales est une commodité.

La référence **x** dans le constructeur ou dans une méthode est une abréviation pour **this.x**.

Où **this** est l'objet construit ou dont on a appelé la méthode.

Cela permet le style :

```
class Pair {  
    int x, y ; // Déclaration  
  
    Pair (int x, int y) {  
        this.x = x ; this.y = y ;  
    }  
    ...  
}
```

La liste

Le tableau même dynamique est encore un peu rigide.

- ▶ Typiquement, il faut connaître la taille du tableau à l'avance.

Mais considérons une « liste » de courses :

- ▶ Typiquement, nous ajoutons les articles à acheter, un par un, sans savoir combien il y en aura.

Tant qu'il y a du papier...

- ▶ Lorsque nous faisons les courses, pour ne rien oublier, nous pouvons acheter les articles dans l'ordre de la liste.

Une analogie plus exacte, est un « carnet de courses » : un article par page !

Soyons plus précis

Une liste est :

- ▶ Soit vide,
- ▶ Soit un premier article et le reste de la liste.

Le plus simple est de procéder ainsi :

- ▶ La liste vide est **null**.
- ▶ La cellule de liste est un objet de la classe :

```
class List {  
    String val ;  
    List next ;  
}
```

```
List moi = new List("patates", new List("thon", null)) ;  
List elle = new List("laitue", new List("boulgour", null)) ;
```

La liste dans la mémoire

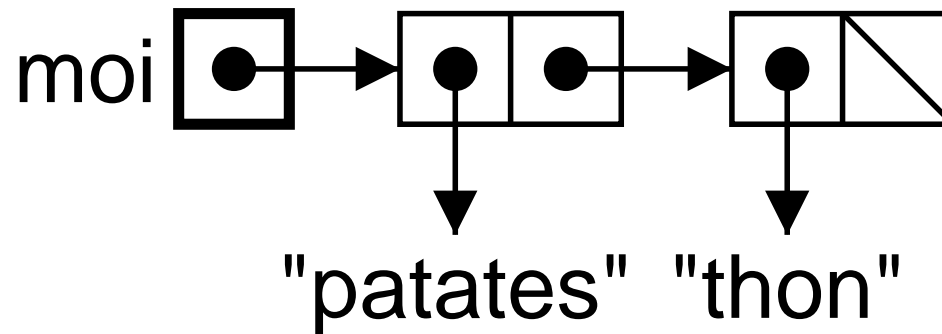
Une valeur de type `List` est

- ▶ Une référence vers une cellule de `List` (pointeur, adresse).
- ▶ Ou bien la référence `null` qui ne pointe nulle part.

Une cellule de liste comporte :

- ▶ L'élément en tête de liste (champ `val`).
- ▶ Et un le reste de la liste (champ `next` de type `List`).

```
List moi = new List("patates", new List("thon", null)) ;
```



Intermède

On veut afficher les listes : peut-on y arriver en redéfinissant `toString` ?

Non ! Pourquoi ?

Parce que `null` est une liste valide, et que `null.toString()` ne fonctionne pas.

```
List pasFaim = null ;  
System.out.println(pasFaim) ;
```

Affiche `null`, car la méthode `println` est du genre.

```
void println(... p) {  
    if (p == null) {  
        printString("null") ;  
    } else {  
        printString(p.toString()) ;  
    }  
}
```

Afficher une liste

Il faut employer une méthode statique.

```
static void print(List p) {  
    for (List q = p ; q != null ; q = q.next) {  
        System.out.println(q.val) ;  
    }  
}
```

Remarques Ce genre de boucle **for** est la façon « idiomatique » de parcourir une liste.

Ici, la variable *q* n'est pas utile :

```
static void print(List p) {  
    for ( ; p != null ; p = p.next) {  
        System.out.println(p.val) ;  
    }  
}
```

(Car on modifie *p*, variable locale, pas la liste).

Quoi de neuf ?

C'est comme la paire, sauf...

```
class List {  
    ...  
    List next ;  
}
```

... que la liste est une structure de donnée récursive.

- ▶ Les listes de courses sont solution de l'équation :

$$L = \mathbf{null} \uplus (\mathbf{String} \times L)$$

- ▶ La programmation sur les listes est « naturellement » récursive.

Programmation naturellement récursive

Fabriquer une nouvelle liste de courses avec deux listes de courses.

```
List courses = List.append(moi, elle) ;
```

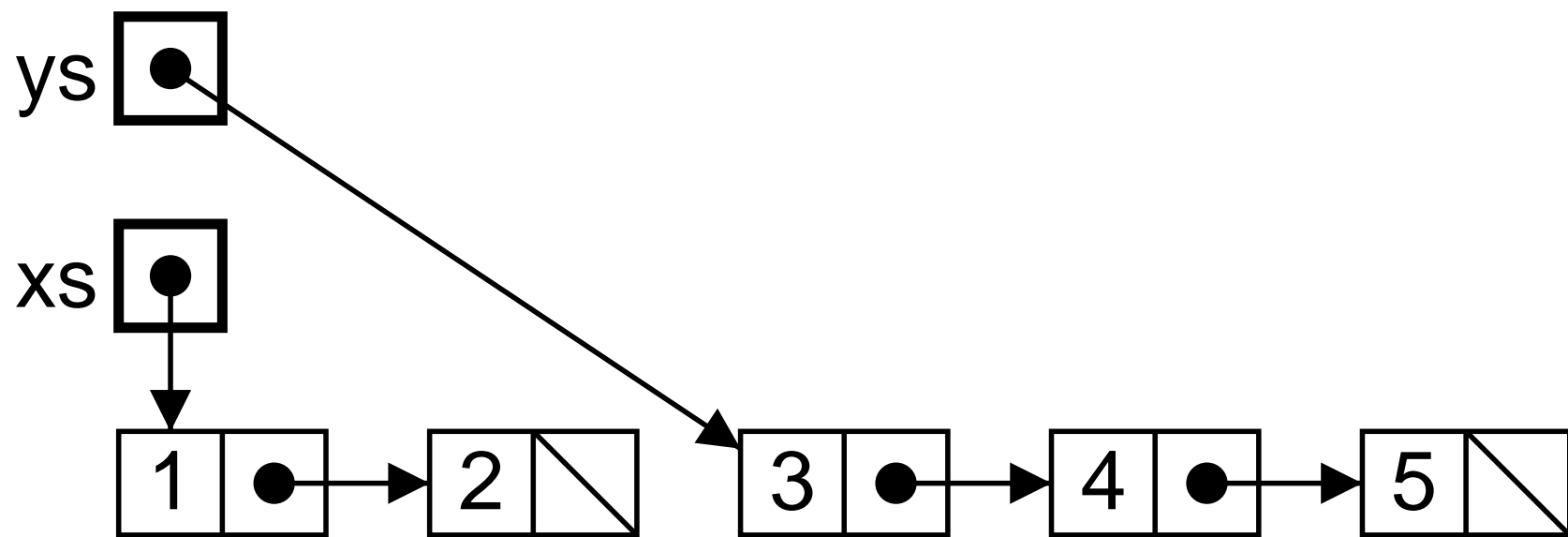
On peut exprimer `append` par des équations (récursives), en notant \emptyset la liste vide et $(a; L)$ une liste non-vide.

$$A(\emptyset, M) = M$$

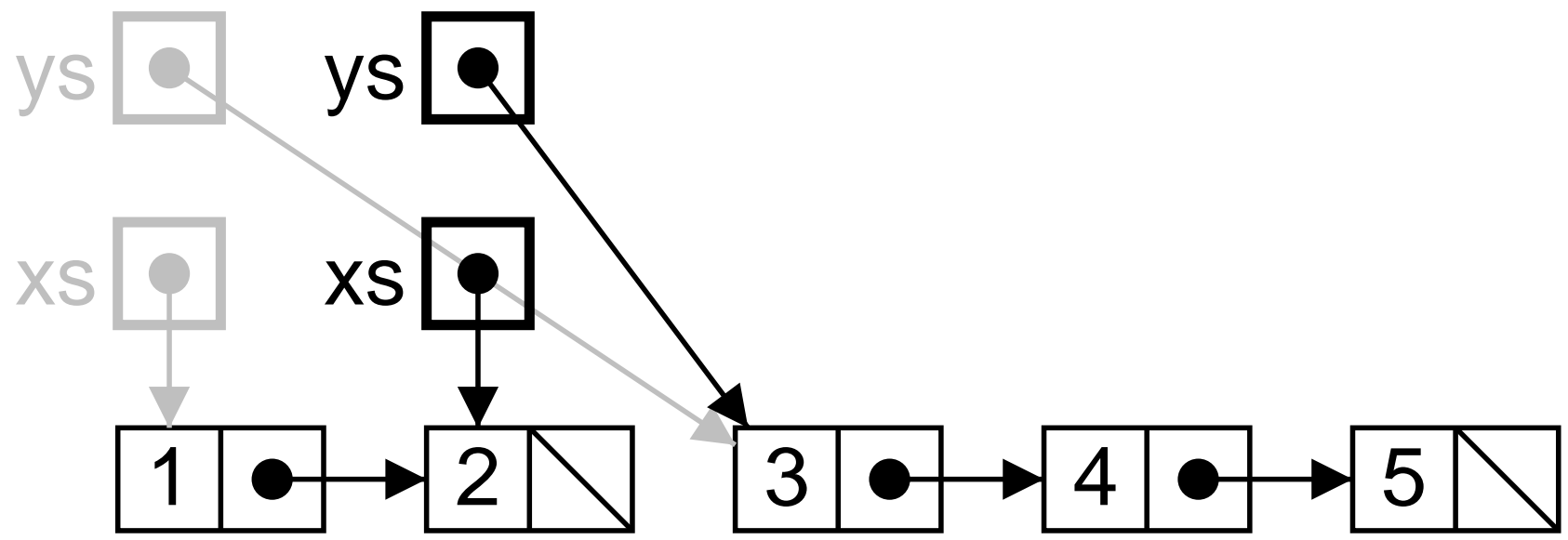
$$A((a; L), M) = a; A(L, M)$$

```
static List append(List p, List q) {  
    if (p == null) {  
        return q ;  
    } else {  
        return new List(p.val, append(p.next, q)) ;  
    }  
}
```

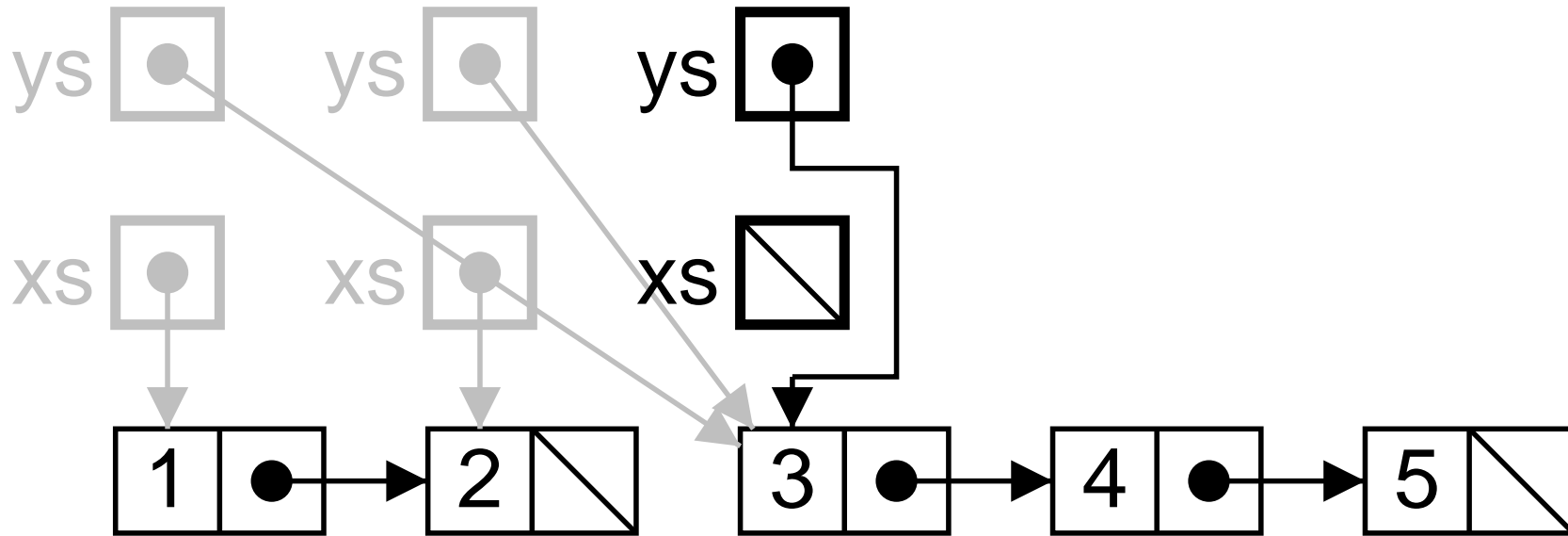

Premier appel



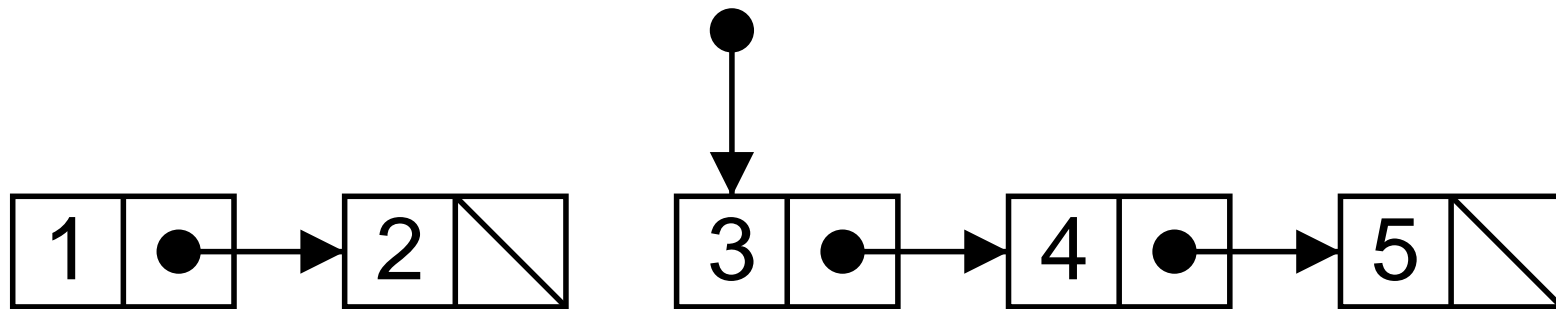
Deuxième appel



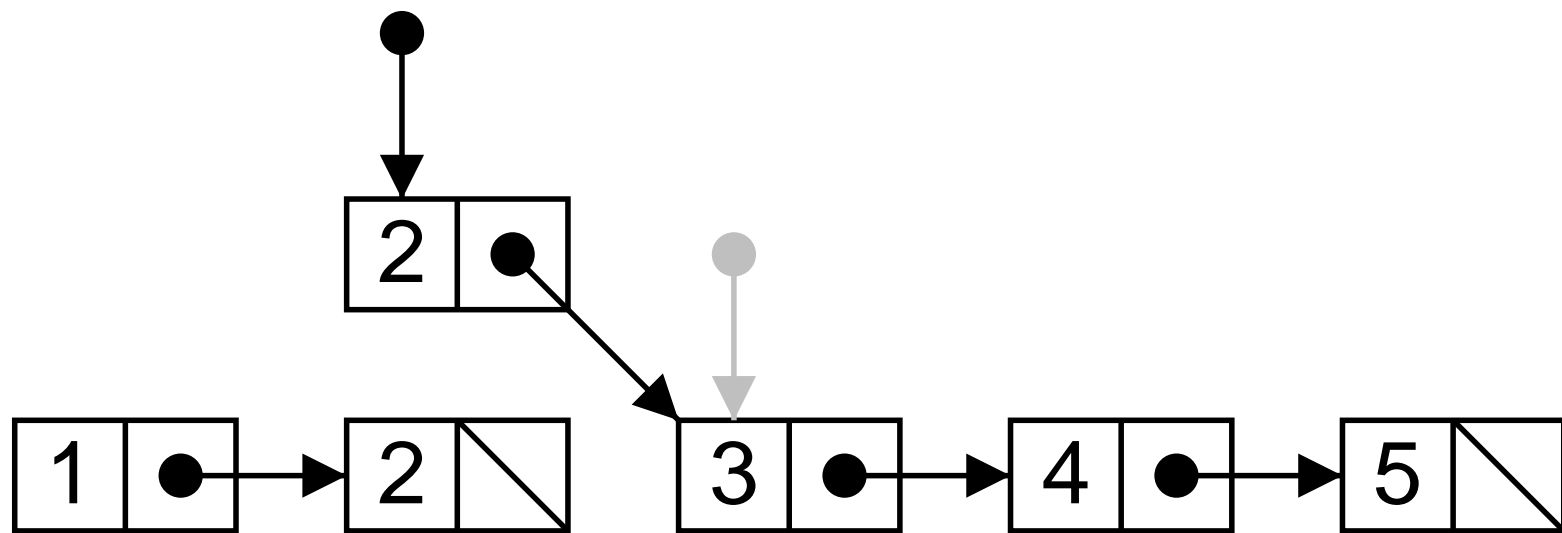
Troisième appel



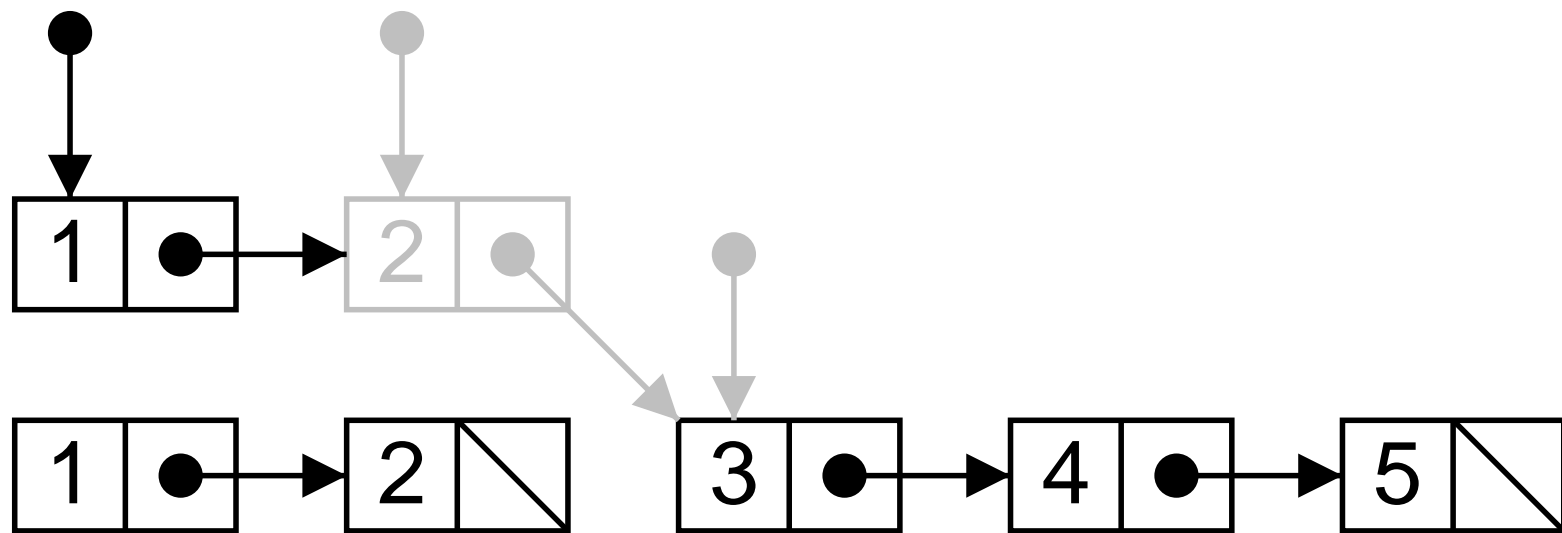
Retour du troisième appel



Retour du deuxième appel



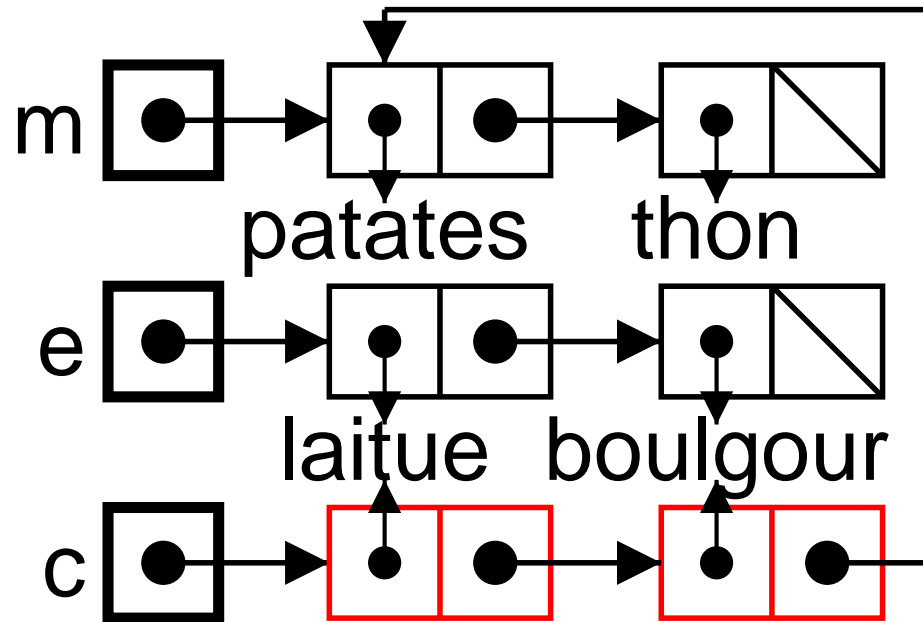
Retour du premier appel



Situation finale

```
List courses = List.append(elle, moi) ;
```

La liste courses est "laitue"; "blougour"; "patates"; "thon"



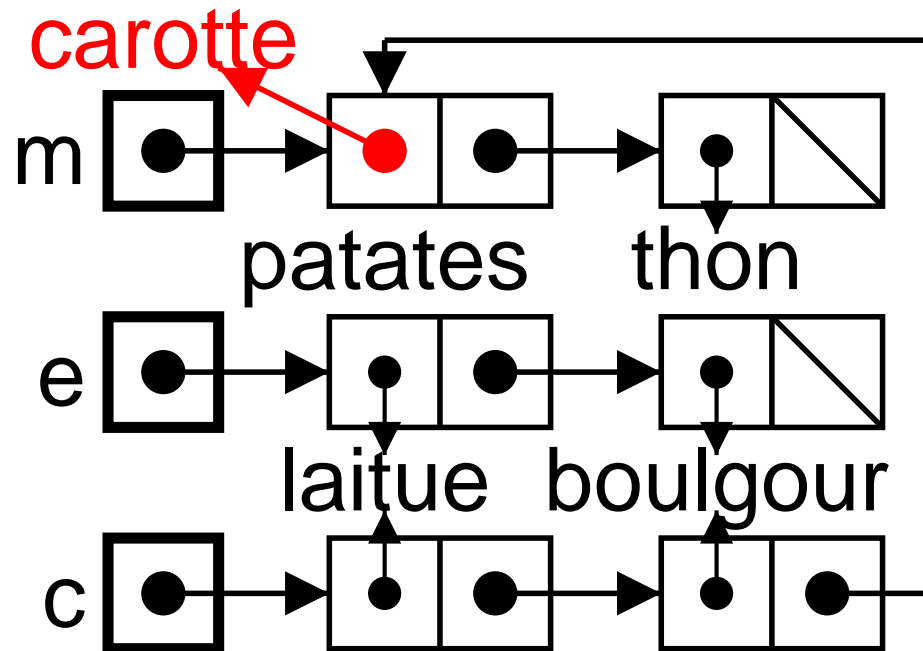
Important : La liste *elle* n'a pas changé. Structure *persistante*.

Moins important : Les cellules de *elle* sont copiées.

Et si je change d'avis

```
moi.val = "carottes" ;
```

Situation :

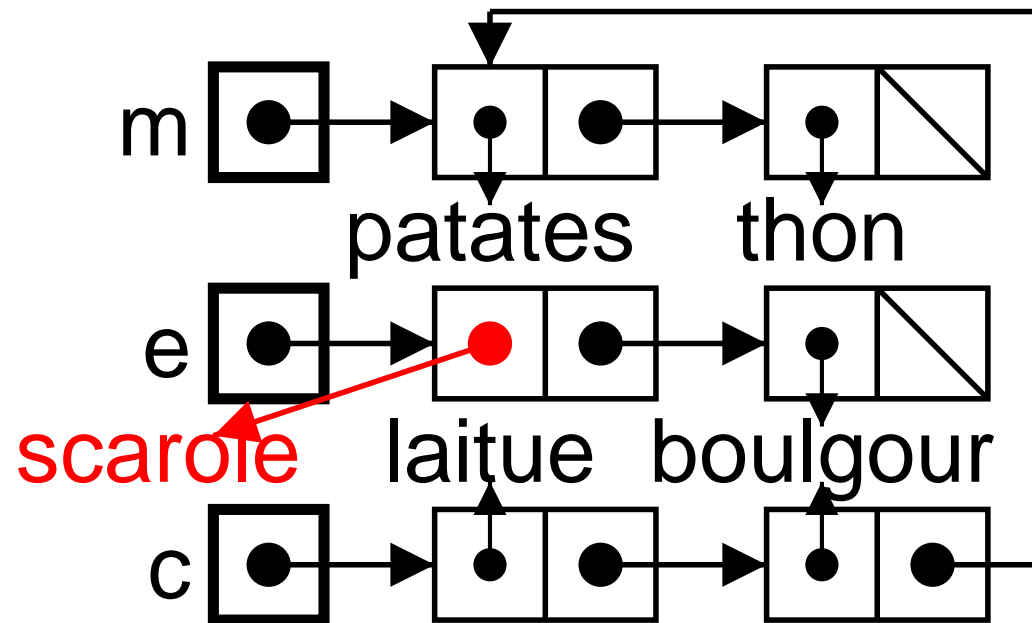


Conclusion : moi devient *"carotte"*; *"thon"* et courses devient *"laitue"*; *"boulgour"*; *"carotte"*; *"thon"*.

Et si elle change d'avis ?

```
elle.val = "scarole" ;
```

Situation :

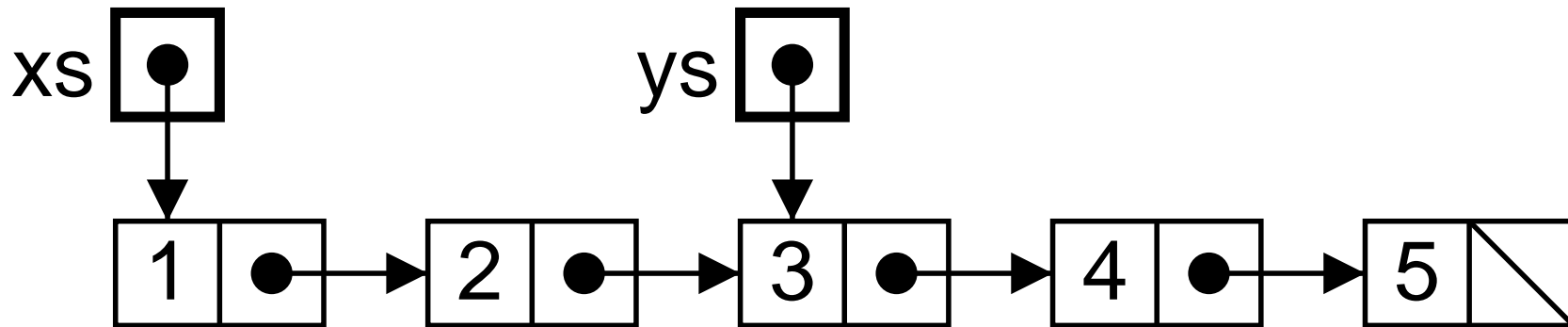
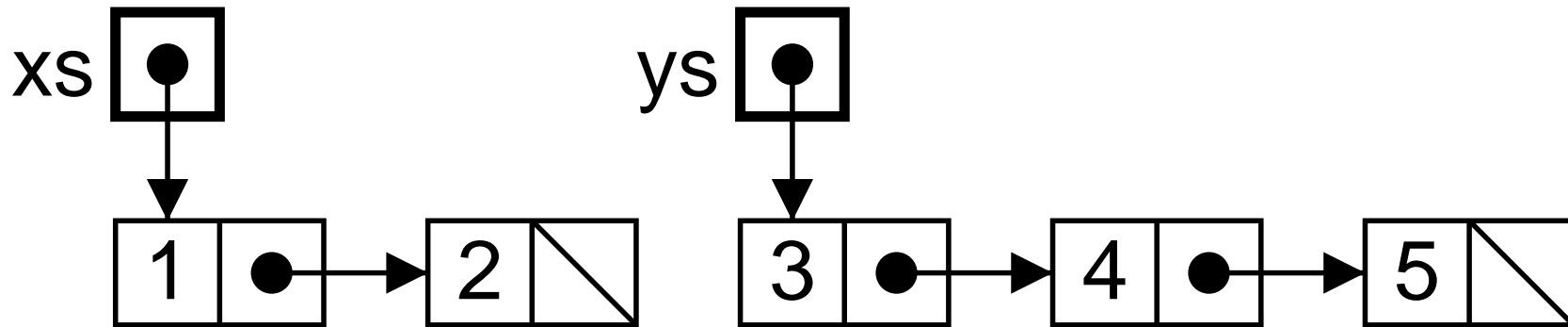


Conclusion : courses ne change pas.

Conclusion

La mutation est incompatible avec les structure persistantes.

J'aime les chose délicates



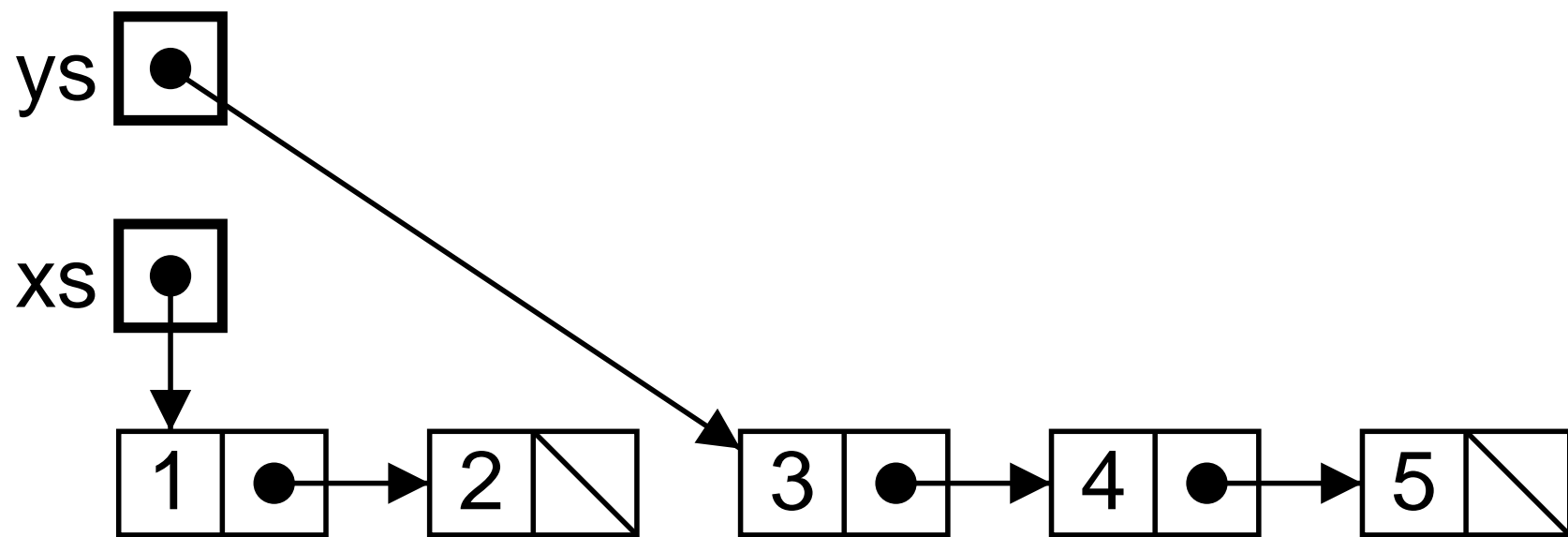
Concaténer en place

La fonction `nappend` concatène les listes « en place »

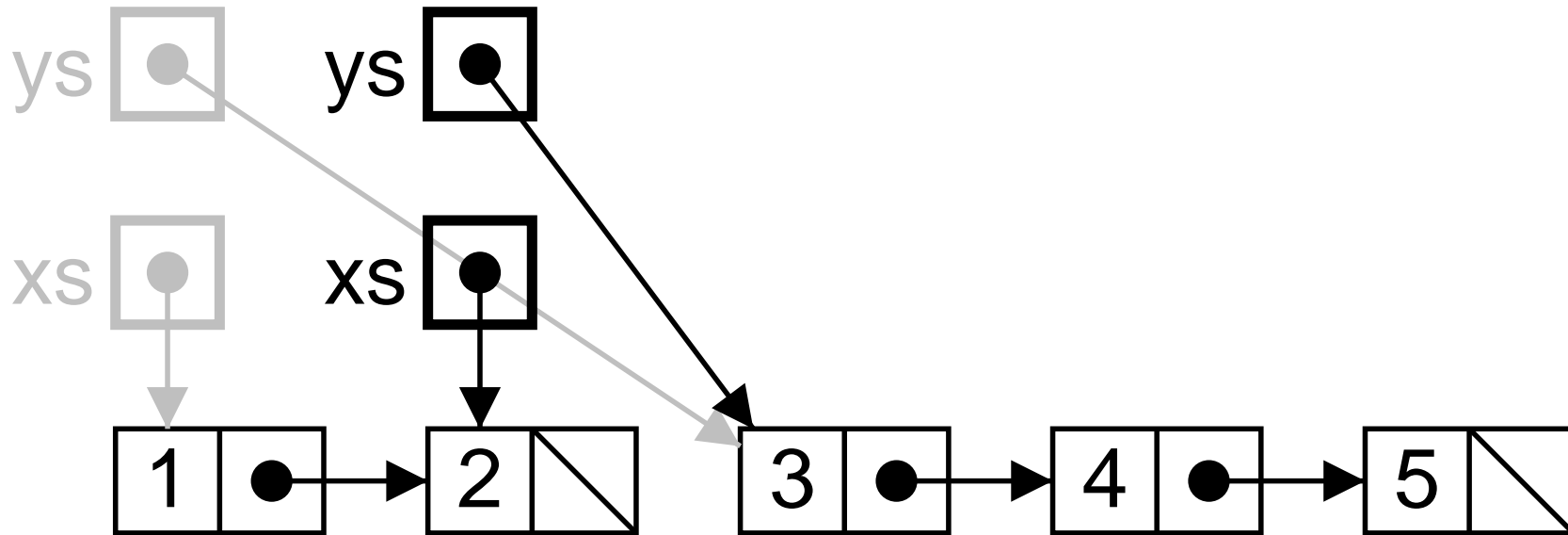
Elle s'écrit facilement, en remplaçant les allocations de cellules de `append` par des mutations du champ `next`.

```
static List nappend(List xs, List ys) {  
    if (xs == null) {  
        return ys ;  
    } else {  
        xs.next = nappend(xs.next, ys) ;  
        return xs ;  
    }  
}
```

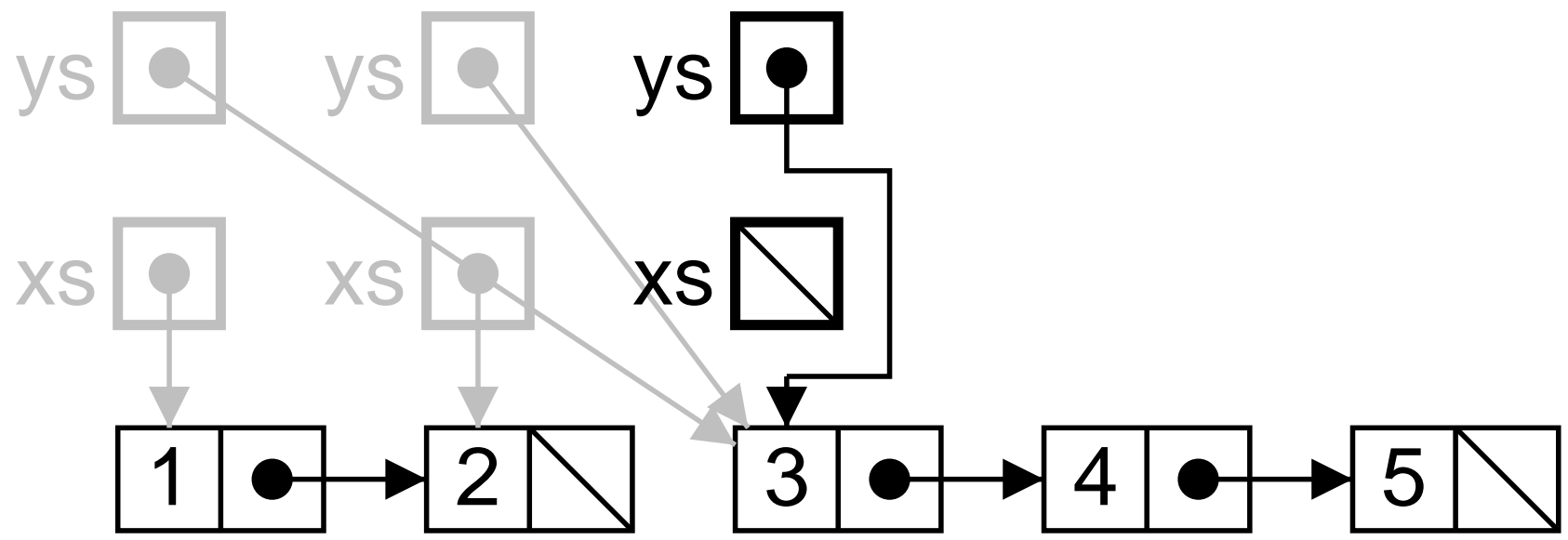
Premier appel



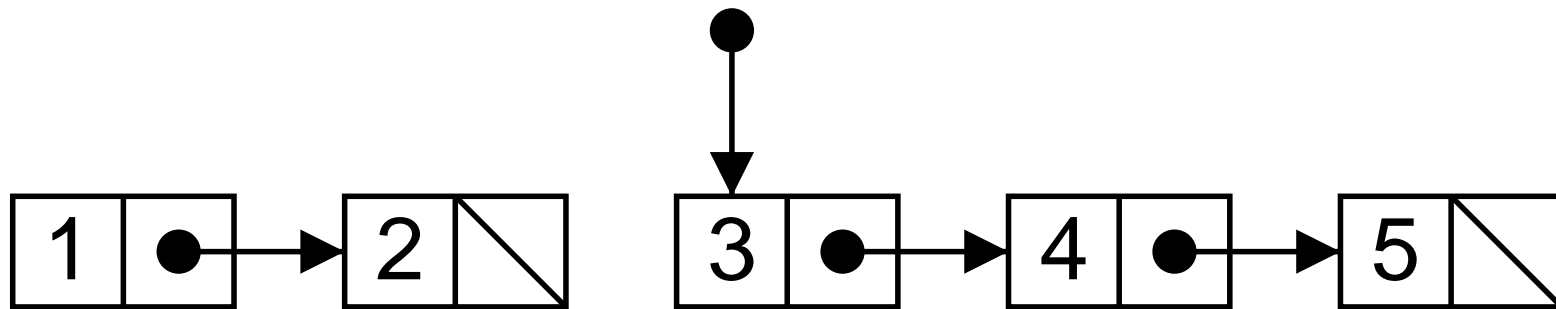
Deuxième appel



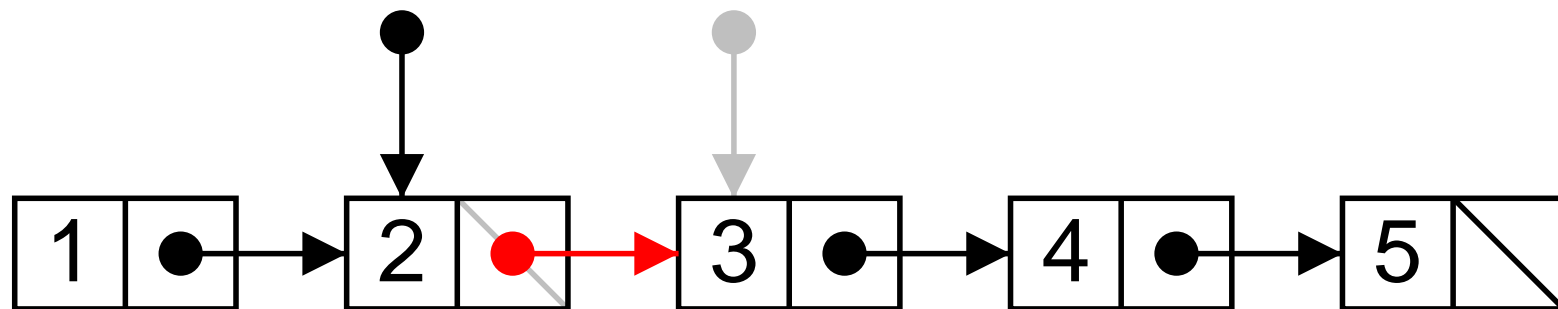
Troisième appel



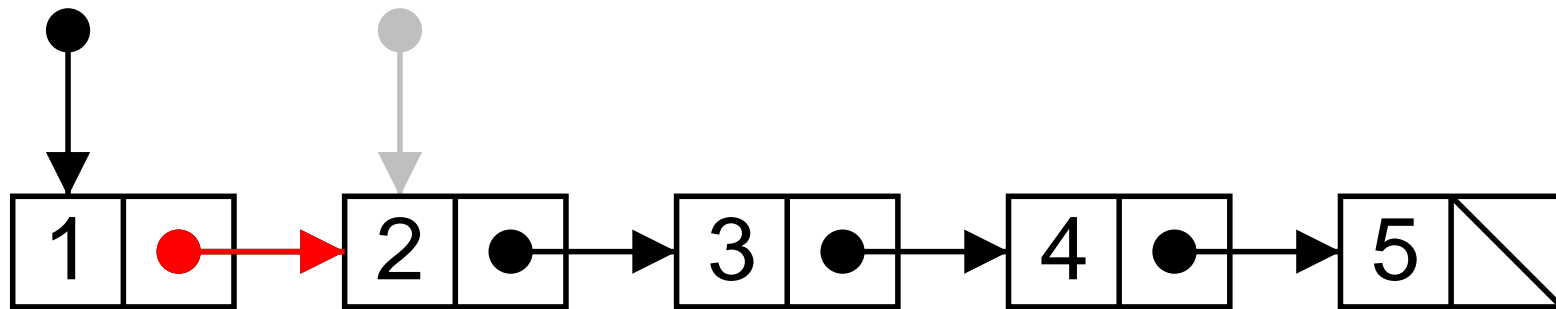
Retour du troisième appel



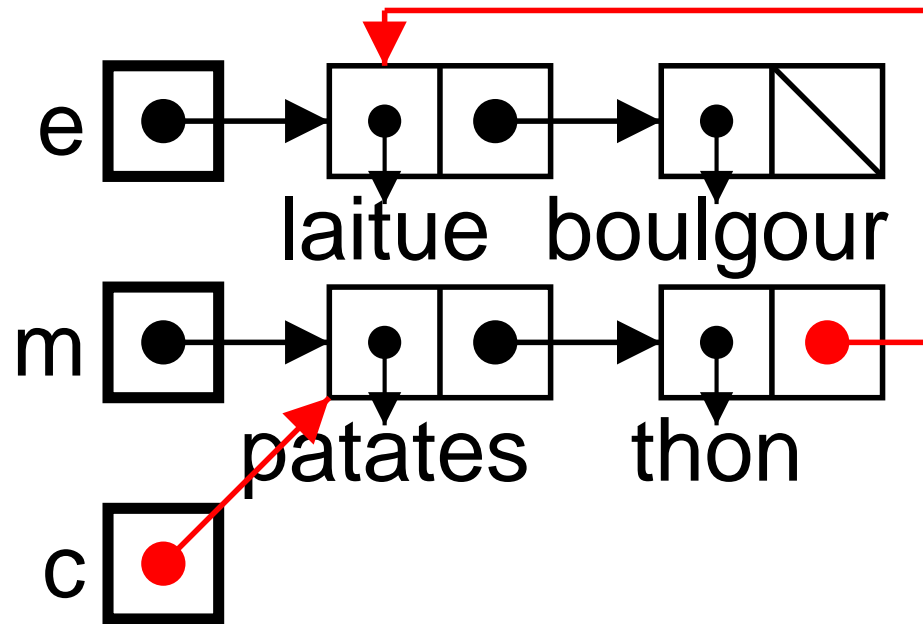
Retour du deuxième appel



Retour du premier appel



Résultat (des courses en place)

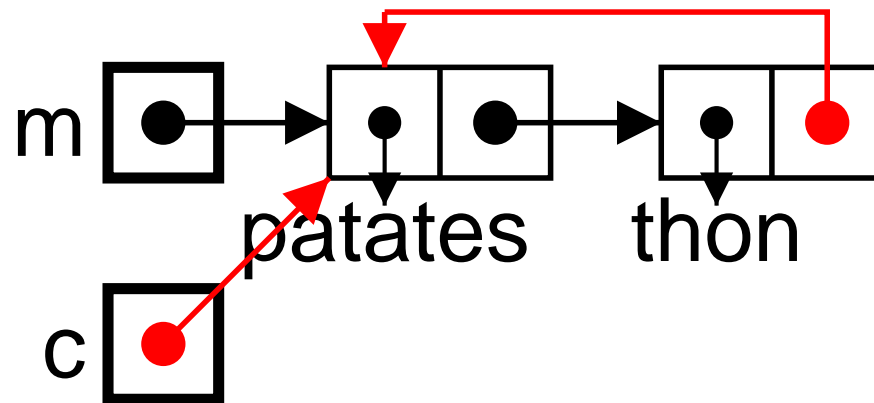


Danger principal La liste pointée par moi a « changé ». Car une de ses cellules a été « mutée ».

Un second problème

Écrivons

```
List courses = List.nappend(moi, moi) ;
```



- ▶ Une liste circulaire, l'interprétation récursive devient douteuse (plus de **null**).
- ▶ Les structures mutables sont délicates.

Programation itérative

Deux façons d'aborder la programmation.

- ▶ « Qu'est-ce que je veux calculer ? ».
 - ▷ Conception plus simple, et plus sûre.
 - ▷ On néglige le coût en temps calcul à l'exécution (pas toujours un problème).

- ▶ « Comment je calcule ? ».
 - ▷ Conception plus délicate, car forcément plus détaillée.
 - ▷ Maîtrise du coût.

En gros, programmation récursive vs. programmation avec des boucles.

Programmation itérative

Comment nappend fonctionne-t-elle ?

- ▶ Trouver la dernière cellule de `xs`.
- ▶ Faire pointer son champ `next` vers `ys`.
- ▶ Ne pas oublier la liste vide.

```
static List nappend2(List xs, List ys) {  
    if (xs == null) return ys ;  
    List zs = xs ;  
    for ( ; zs.next != null ; zs = zs.next) ;  
    zs.next = ys ;  
    return xs ;  
}
```

Et sans modifier la liste d'origine ?

Comment obtenir une concaténation itérative et non-destructive simplement :

```
static List append2(List xs, List ys) {  
    List zs = copy(xs) ; // Copier xs qui va être modifiée  
    return nappend2(zs, ys) ;  
}
```

(En pratique, on ne devrait presque jamais écrire un truc pareil. Si le besoin s'en fait sentir, penser aux listes persistentes.)

En trichant

Il est facile d'écrire un `copy` récursif.

```
List copy(List xs) {  
    if (xs == null) {  
        return null ;  
    } else {  
        return new List (xs.val, copy(xs.next)) ;  
    }  
}
```

Ne pas tricher c'est écrire `copy` en style itératif.

Qu'est-ce que ça fait ?

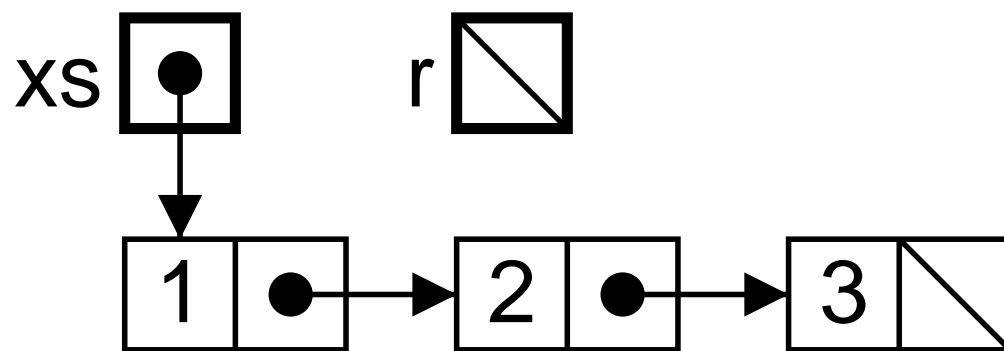
- ▶ Parcourir la liste `xs`,
- ▶ En copiant ses cellules dans une nouvelle liste.

Premier essai

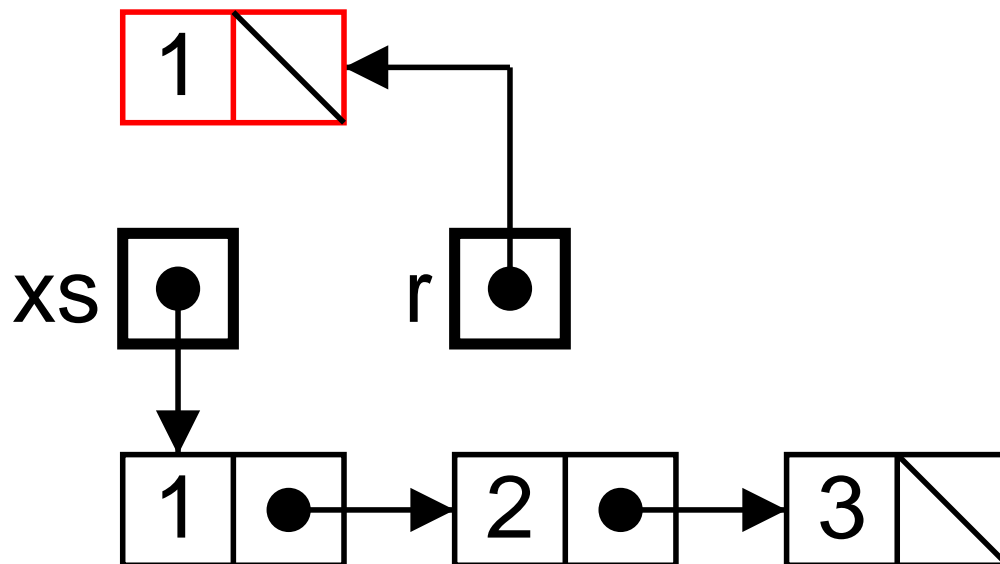
- ▶ Parcourir la liste `xs`
- ▶ En la recopiant.
- ▶ Ne pas oublier la liste vide (c'est bon !).

```
static List copy2(List xs) {  
    List r = null ;  
    for ( ; xs != null ; xs = xs.next) {  
        r = new List (xs.val, r) ;  
    }  
    return r ;  
}
```

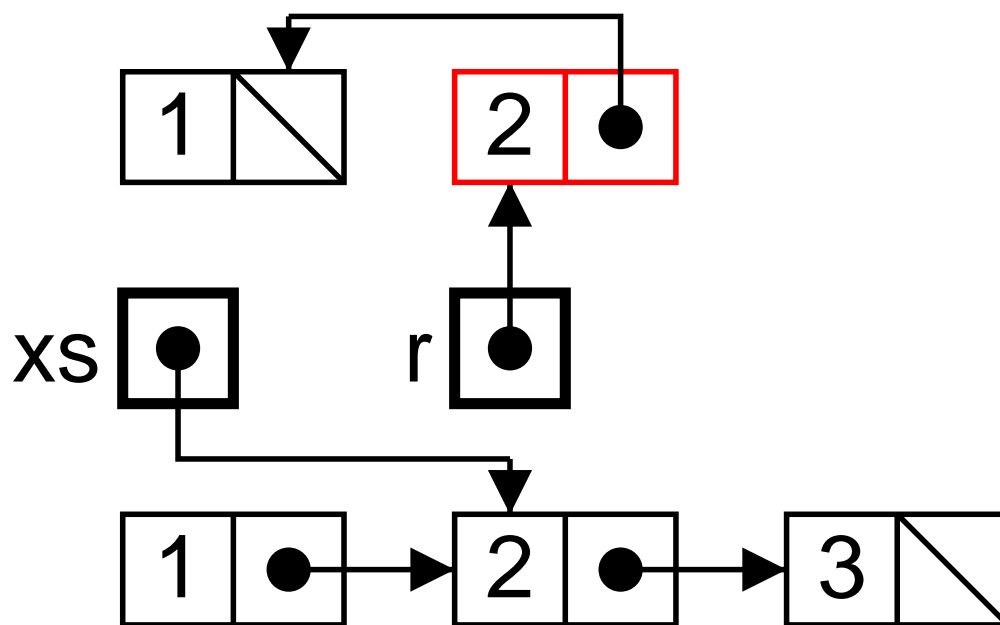

Première itération



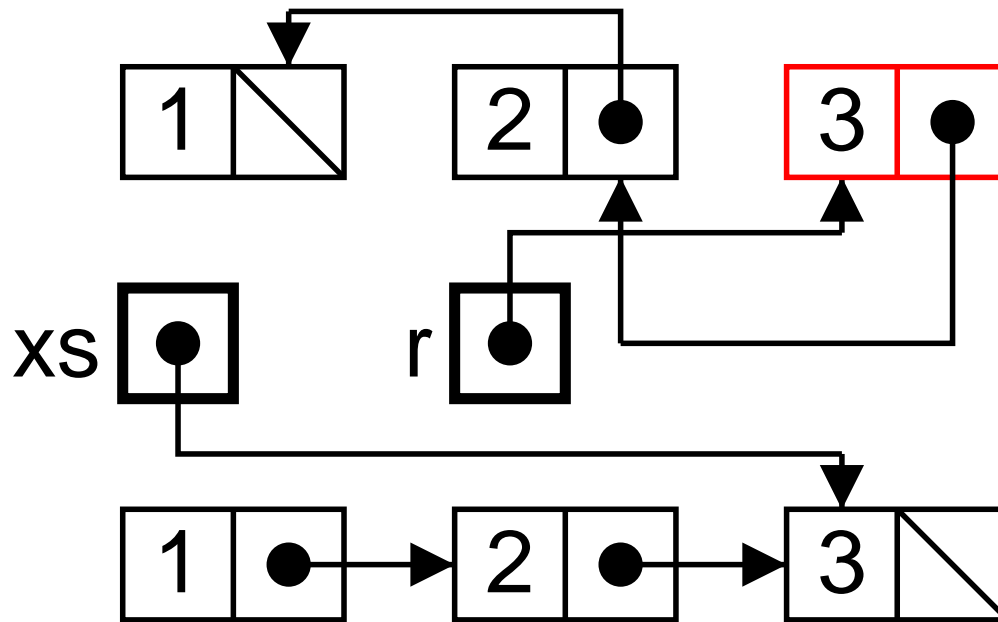
Exécution du corps



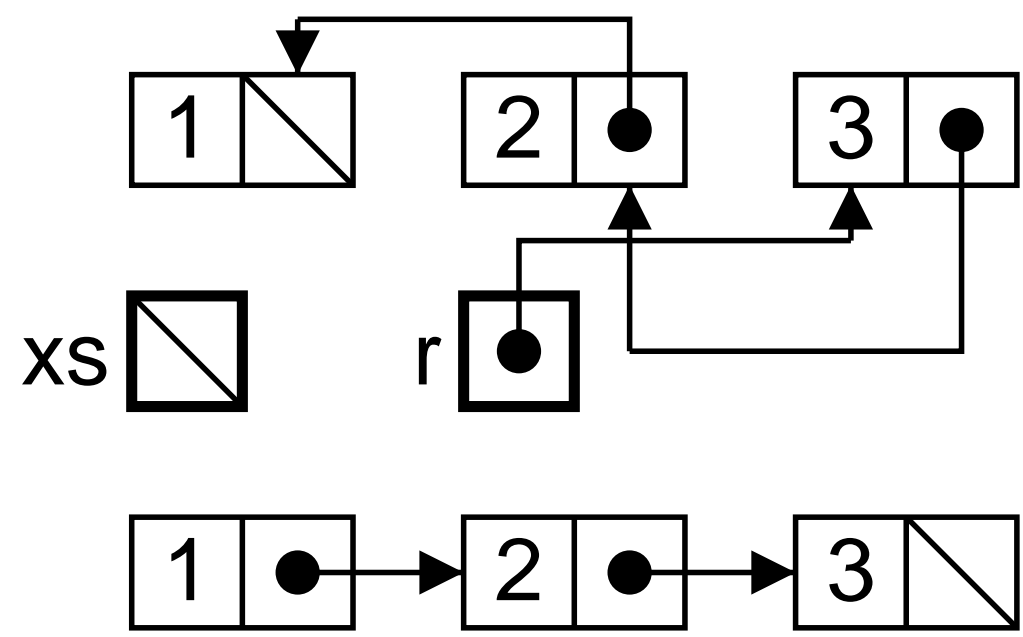
Deuxième itération (fin)



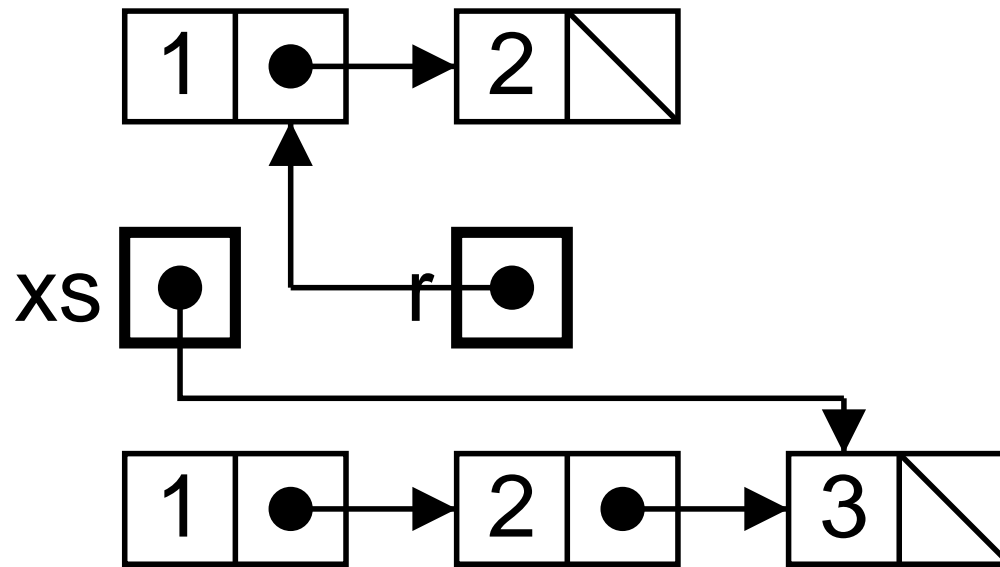
Troisième itération (fin)



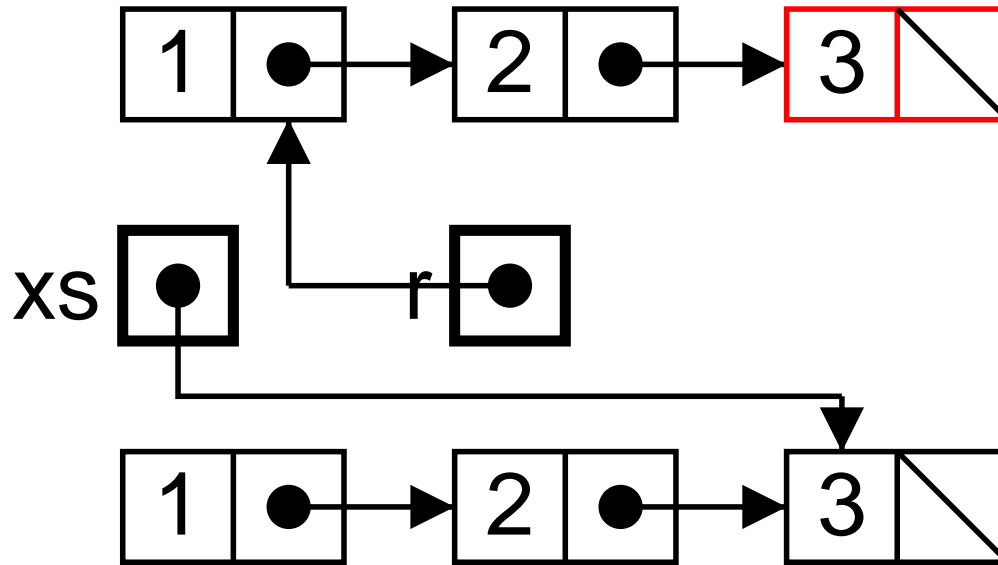
Et c'est fini



Copier dans l'ordre



Copier dans l'ordre



Programmation itérative de copy

- ▶ Il faut ajouter la nouvelle cellule à *la fin* de la liste résultat (variable `lst` — dernière cellule).
- ▶ Et il faut que cela soit vrai tout le temps, y compris au début.
- ▶ Et attention à la liste vide

```
static List copy2(List xs) {  
    if (xs == null) return ys ;  
    List r = new List (xs.val,null) ;  
    List lst = r ;  
    for ( xs = xs.next ; xs != null ; xs = xs.next) {  
        lst.next = new List (xs.val, null) ;  
        lst = lst.next ;  
    }  
    return r ;  
}
```


Programmation itérative de append

Comment append fonctionne-elle ?

- ▶ Copier (la liste référencée par) `xs` dans l'ordre.
- ▶ En remplaçant le **null** de la fin par `ys`.

```
static List append2(List xs, List ys) {
    if (xs == null) return ys ;
    List r = new List (xs.val,null) ;
    List lst = r ;
    for ( xs = xs.next ; xs.next != null ; xs = xs.next) {
        lst.next = new List (xs.val, null) ;
        lst = lst.next ;
    }
    lst.next = ys ;
    return r ;
}
```

Les structures de données dynamiques

Nature des structures de données :

- ▶ Persistentes (ou fonctionelles, ou non-mutable).
- ▶ Mutables (ou impératives, ou en place).

Le choix persistant/mutable est un choix de fonctionnalité. Il doit être conscient et influence l'ensemble du programme.

Style de programmation :

- ▶ Récursif.
- ▶ Itératif (avec des boucles).

Le choix récursif/itératif est un choix d'efficacité, compromis facilité d'écriture/efficacité du programme.

Résumé

On peut définir *quatre* catégories. Selon la nature des structures de données et le style de programmation.

	Persistent	Mutable
Récurif	append	nappend
Itératif	append2	append2

- ▶ Simplicité de conception : récursif et persistent.
- ▶ Itératif souvent plus efficace (mais est-ce bien utile ?), parfois naturel (inverser une liste).
- ▶ Structures mutables : parfois plus efficace (dépend de l'efficacité de GC), parfois voulu.