

# Cours 1: Eléments de Java, Listes

Olivier Bournez

bournez@lix.polytechnique.fr  
LIX, Ecole Polytechnique

INF421-b

Bases de la programmation et de l'algorithmique

# Objectifs

- Structures de données “dynamiques”:
  - ▶ Listes, Files, Piles, Tables de Hachage, Arbres, ...
  - ▶ Avec leurs algorithmes élémentaires, et quelques applications.
- Un peu de théorie
  - ▶ Complexité de quelques algorithmes
  - ▶ Théorie des automates finis, ...
- Perfectionnement en programmation:
  - ▶ Langage support: JAVA.
  - ▶ Comprendre.
  - ▶ Programmer = Modéliser, structurer les données, concevoir des algorithmes, coder, tester.

# Organisation du cours

- 9 blocs, soit 9 vendredis.
  - ▶ Le matin, amphi, de 10h30 à 12h00.
  - ▶ L'après midi, TP.
    - Gr1 -2, Salle info 34, J.C. Filliâtre - S. Lengrand
- Page du cours  
[www.enseignement.polytechnique.fr/informatique/INF421](http://www.enseignement.polytechnique.fr/informatique/INF421)  
et vos questions à [Olivier.Bournez@polytechnique.fr](mailto:Olivier.Bournez@polytechnique.fr) et/ou  
aux enseignants de l'équipe.
- Évaluation.
  - ▶ TP noté, le sixième, **11** décembre.
  - ▶ Contrôle à la fin,
  - ▶ Note de module :  $\max(CC, \frac{3}{4}CC + \frac{1}{4}f(TP))$ , avec  $f$  proche de l'identité.

# Organisation du cours

- 9 blocs, soit 9 vendredis.
  - ▶ Le matin, amphi, de 10h30 à 12h00.
  - ▶ L'après midi, TP.
    - Gr1 -2, Salle info 34, J.C. Filliâtre - S. Lengrand
- Page du cours  
[www.enseignement.polytechnique.fr/informatique/INF421](http://www.enseignement.polytechnique.fr/informatique/INF421)  
et vos questions à [Olivier.Bournez@polytechnique.fr](mailto:Olivier.Bournez@polytechnique.fr) et/ou  
aux enseignants de l'équipe.
- Évaluation.
  - ▶ TP noté, le sixième, **11** décembre.
  - ▶ Contrôle à la fin,
  - ▶ Note de module :  $\max(CC, \frac{3}{4}CC + \frac{1}{4}f(TP))$ , avec  $f$  proche de l'identité.
- Possibilité de tutorat: les lundis de 16h30 à 18h30, Salle info 36,  
avec E. Nicolas.

# Aujourd'hui

Rappels de Java

Classes en Java

Structures Dynamiques

# Déclaration

- En Java, toutes les variables sont déclarées et typées.

```
String a;  
float z;  
int[] y;  
int x = 2;
```

- Ingrédients:
  - ▶ Un type
  - ▶ Un identificateur
  - ▶ Une éventuelle initialisation

**int** x=2; est équivalent à **int** x; x=2;

# Déclaration

- En Java, toutes les variables sont déclarées et typées.

```
String a;  
float z;  
int[] y;  
int x = 2;
```

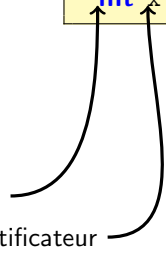
- Ingrédients:
  - ▶ Un type
  - ▶ Un identificateur
  - ▶ Une éventuelle initialisation

**int** x=2; est équivalent à **int** x; x=2;

# Déclaration

- En Java, toutes les variables sont déclarées et typées.

```
String a;  
float z;  
int[] y;  
int x = 2;
```



- Ingrédients:
  - ▶ Un type
  - ▶ Un identificateur
  - ▶ Une éventuelle initialisation

**int** x=2; est équivalent à **int** x; x=2;

# Déclaration

- En Java, toutes les variables sont déclarées et typées.

```
String a;  
float z;  
int[] y;  
int x = 2;
```

- Ingrédients:

- ▶ Un type
- ▶ Un identificateur
- ▶ Une éventuelle initialisation

**int** x=2; est équivalent à **int** x; x=2;

# Une façon de représenter la mémoire

- La mémoire est constituée de boîtes (cases), qui peuvent être nommées.

Représentation:

nom 

contenu
---------

@adresse 

contenu
---------

## Deux façons de manipuler une variable

- On utilise directement **la valeur** de la variable

v 748

- On utilise **une référence** (adresse, handle) sur l'endroit où la variable est rangée en mémoire.

v @10d448

@10d448 356

- ▶ On dit que v pointe sur sa valeur

## Deux façons de manipuler une variable

- On utilise directement **la valeur** de la variable

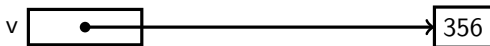
v 748

- On utilise **une référence** (adresse, handle) sur l'endroit où la variable est rangée en mémoire.

v @10d448

@10d448 356

- ▶ On dit que v pointe sur sa valeur
- ▶ Représentation graphique:



# Les types

## ■ Distinction:

- ▶ Les types primitifs = booléens, caractères, entiers, réels.
- ▶ Les autres types = objets définis à partir d'une classe, tableaux, chaînes.

## ■ Principe de base:

- ▶ Une variable de type primitif est manipulée par valeur.
- ▶ Une variable de type non-primitif est manipulée par référence.

# Types primitifs

- Les booléens
  - ▶ (**boolean**): **true** ou **false** 1 bit.
- Les caractères
  - ▶ (**char**): caractère unicode 2 octets.
- Les entiers:
  - ▶ (**byte**):  $[-128, 127]$  1 octet.
  - ▶ (**short**):  $[-32768, 32767]$  2 octets.
  - ▶ (**int**):  $[-2147483648, 2147483647]$  4 octets.
  - ▶ (**long**):  $[-9.10^{18}, 9.10^{18}]$  8 octets.
- Les réels:
  - ▶ (**float**): simple précision 4 octets.
  - ▶ (**double**): double précision 8 octets.

## Expérience: Type primitif

```
public static void main(String[] args)
{
    int x=245;
    System.out.println(x);
}
```

- Affiche: 245
- En mémoire:

x 245

# Types non-primitifs

## ■ Tableaux:

```
int[] tab;  
char[] s = new char[10];
```

## ■ Chaînes de caractères:

```
String jour = "vendredi";
```

## ■ Objets définis à partir d'une classe:

- ▶ Définis par les bibliothèques JAVA.
  - Ex: dans `System.out.println("hello");` `System.out` est une variable de la classe `PrintStreamWriter`.
- ▶ Que l'on fabrique soi-même.

```
class Personne  
{  
String nom;  
int age;  
}  
  
Personne p;
```

## Expérience: Tableau d'entiers

```
public static void main(String[] args)
{
    int[] T = {45,33,29};
    System.out.println(T);
}
```

- Affiche: *[I@e0e1c6*

## Expérience: Tableau d'entiers

```
public static void main(String[] args)
{
    int[] T = {45,33,29};
    System.out.println(T);
}
```

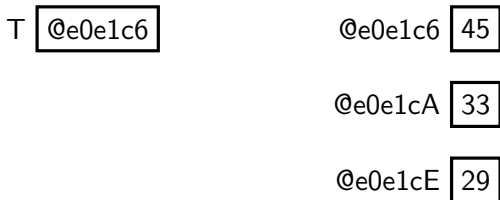
- Affiche: *[I@e0e1c6*
- Signification:
  - ▶ [: Tableau
  - ▶ I: D'entiers
  - ▶ @: Adresse en hexadécimal

## Représentation en mémoire (tableau d'entiers)

```
public static void main(String[] args)
{
    int[] T = {45,33,29};
    System.out.println(T);
}
```

Affiche: *[I@e0e1c6*

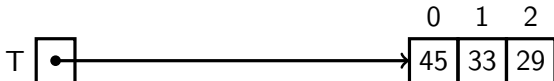
Mémoire (schématiquement):



## Représentation en mémoire (tableau d'entiers)

```
public static void main(String[] args)
{
    int[] T = {45,33,29};
    System.out.println(T);
}
```

Représentation graphique:



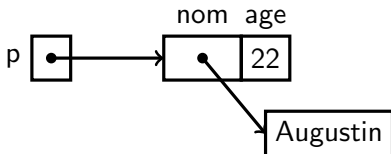
## Expérience: Objet

```
class Personne
{ String nom;
  int  age; }

public static void main(String[] args)
{
  Personne p = new Personne();
  p.nom = "Augustin";
  p.age = 22;
  System.out.println(p);
}
```

Affiche: *Personne@66848c*

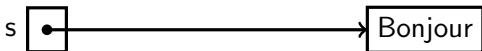
Représentation graphique:



## Expérience: Chaîne de caractères

```
String s= "Bonjour";  
System.out.println(s);
```

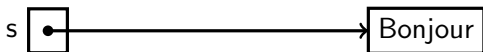
Affiche: *Bonjour*



## Expérience: Chaîne de caractères

```
String s= "Bonjour";  
System.out.println(s);
```

Affiche: *Bonjour*



Pourquoi?

- `System.out.println` cherche à traduire son argument sous la forme d'une chaîne de caractères.
- La traduction est le résultat de la méthode `toString`.
- Toute classe possède une méthode `toString` par défaut: elle affiche le nom de la classe suivi de `@` et l'adresse de l'objet en hexadécimal.
- La classe `String` redéfinit la méthode `toString`.

## Intérêt en pratique: toString()

```
class Personne
{ String nom;
  int age;
  public String toString() {
    return ("nom: " + nom + " age: "+ age);
  }}

public static void main(String[] args)
{
  Personne p = new Personne();
  p.nom = "Augustin"; p.age=23;
  System.out.println(p);
}
```

Affiche: *nom: Augustin age: 23*

# Initialisations

- Toutes les variables doivent être initialisées.

```
int x;  
int y=2;  
System.out.println(x+" "+y);
```

Affiche: *Variable x might not have been initialized*

# L'opérateur `new`

- L'initialisation d'une variable non-primitive passe par `new`.
- `new` crée dynamiquement un objet, et retourne une référence vers cet objet.
- Lors de la création par `new`, les champs/cases/boîtes reçoivent une valeur par défaut:
  - ▶ `false` pour un booléen.
  - ▶ 0 pour un entier ou un réel.
  - ▶ `null` pour tous les autres cas (il ne reste que des références).

Exemple:

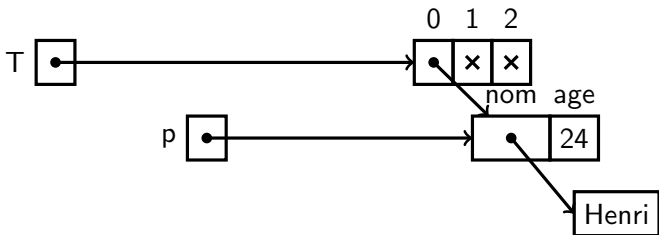
```
Personne p = new Personne();  
int[] tab = new int[10];  
System.out.println(p.nom + "/" + tab[5]);
```

Affiche: *null/0*

## Tableau de structures

```
public static void main(String[] args)
{
    Personne[] T = new Personne[3];
    Personne p = new Personne();
    p.nom = "Henri"; p.age = 24;
    T[0] = p;
    System.out.println(T);
}
```

Affiche: *[LPersonne;@10d448*

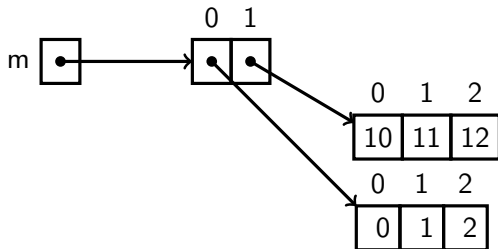


## Tableau de tableaux = Matrice

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

créé et initialise à 0 une matrice  $2 \times 3$ .

```
for (int i = 0; i < m.length; i++) {  
    for (int j=0; j < m[i].length; j++) {  
        m[i][j] = 10*i+j;  
    }  
}
```



## Le statut particulier de `null`

- `null` est une référence particulière.
- On peut affecter `null` à toute référence.
- Mais aucun champ/case/boîte/méthode n'est accessible à partir d'une variable qui vaut `null`.

```
public static void main(String[] args)
{
    Personne p =null;
    p.nom = "Augustin";
}
```

Affiche: *Exception in thread "main"*  
*java.lang.NullPointerException*

- Représentation graphique: `null` sera représenté par une croix.

# Affectation

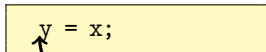
- Les rôles du membre droit et du membre gauche dans une affectation ne sont pas symétriques.

$$y = x;$$

- ▶ A gauche, une variable: une boîte
  - ▶ A droite, une variable: le contenu d'une boîte
- Règle:  $y = x$  remplace le contenu de la boîte  $y$  par le contenu de la boîte  $x$ 
    - ▶ Si  $x$  ou  $y$  sont des références, on travaille sur les références, pas sur les contenus.

# Affectation

- Les rôles du membre droit et du membre gauche dans une affectation ne sont pas symétriques.

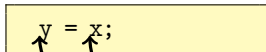


```
y = x;
```

- ▶ A gauche, une variable: une boîte
  - ▶ A droite, une variable: le contenu d'une boîte
- Règle:  $y = x$  remplace le contenu de la boîte  $y$  par le contenu de la boîte  $x$ 
    - ▶ Si  $x$  ou  $y$  sont des références, on travaille sur les références, pas sur les contenus.

# Affectation

- Les rôles du membre droit et du membre gauche dans une affectation ne sont pas symétriques.



The diagram shows a yellow rectangular box containing the code `y = x;`. Two curved arrows originate from the right side of the box. One arrow points from the variable `x` to the variable `y`. The other arrow points from the right side of the box to the text below, indicating that the value of `x` is being used to update `y`.

- ▶ A gauche, une variable: une boîte
  - ▶ A droite, une variable: le contenu d'une boîte
- Règle: `y = x` remplace le contenu de la boîte `y` par le contenu de la boîte `x`
    - ▶ Si `x` ou `y` sont des références, on travaille sur les références, pas sur les contenus.

# Affectations

```
public static void main(String[] args)
{
    int s = 1;
    int t = s;
    t = 4;

    int[] S = {1, 2, 3};
    int[] T = S;
    T[1] = 4;
    System.out.println(s + ":" + S[1]);
}
```

Affiche: 1:4

# Affectations

```
public static void main(String[] args)
{
    int s = 1;
    int t = s;
    t = 4;

    int[] S = {1, 2, 3};
    int[] T = S;
    T[1] = 4;
    System.out.println(s + ":" + S[1]);
}
```

Affiche: 1:4

s ?

# Affectations

```
public static void main(String[] args)
{
    int s = 1;
    int t = s;
    t = 4;

    int[] S = {1, 2, 3};
    int[] T = S;
    T[1] = 4;
    System.out.println(s + ":" + S[1]);
}
```

Affiche: 1:4

s 1

# Affectations

```
public static void main(String[] args)
{
    int s = 1;
    int t = s;
    t = 4;

    int[] S = {1, 2, 3};
    int[] T = S;
    T[1] = 4;
    System.out.println(s + ":" + S[1]);
}
```

Affiche: 1:4

s 

1
---

t 

?
---

# Affectations

```
public static void main(String[] args)
{
    int s = 1;
    int t = s;
    t = 4;

    int[] S = {1, 2, 3};
    int[] T = S;
    T[1] = 4;
    System.out.println(s + ":" + S[1]);
}
```

Affiche: 1:4

s 

1
---

t 

1
---

# Affectations

```
public static void main(String[] args)
{
    int s = 1;
    int t = s;
    t = 4;

    int[] S = {1, 2, 3};
    int[] T = S;
    T[1] = 4;
    System.out.println(s + ":" + S[1]);
}
```

Affiche: 1:4

s 

1
---

t 

4
---

# Affectations

```
public static void main(String[] args)
{
    int s = 1;
    int t = s;
    t = 4;

    int[] S = {1, 2, 3};
    int[] T = S;
    T[1] = 4;
    System.out.println(s + ":" + S[1]);
}
```

Affiche: 1:4

s

S

t

# Affectations

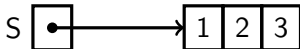
```
public static void main(String[] args)
{
    int s = 1;
    int t = s;
    t = 4;

    int[] S = {1, 2, 3};
    int[] T = S;
    T[1] = 4;
    System.out.println(s + ":" + S[1]);
}
```

Affiche: 1:4

s [ 1 ]

t [ 4 ]

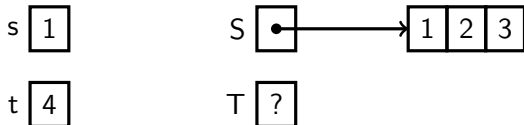


# Affectations

```
public static void main(String[] args)
{
    int s = 1;
    int t = s;
    t = 4;

    int[] S = {1, 2, 3};
    int[] T = S;
    T[1] = 4;
    System.out.println(s + ":" + S[1]);
}
```

Affiche: 1:4

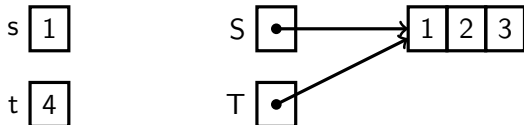


# Affectations

```
public static void main(String[] args)
{
    int s = 1;
    int t = s;
    t = 4;

    int[] S = {1, 2, 3};
    int[] T = S;
    T[1] = 4;
    System.out.println(s + ":" + S[1]);
}
```

Affiche: 1:4

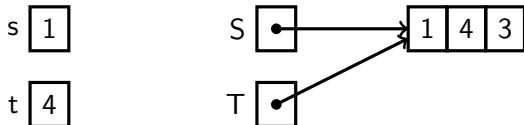


# Affectations

```
public static void main(String[] args)
{
    int s = 1;
    int t = s;
    t = 4;

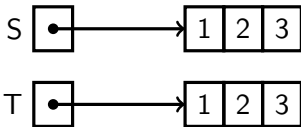
    int[] S = {1, 2, 3};
    int[] T = S;
    T[1] = 4;
    System.out.println(s + ":" + S[1]);
}
```

Affiche: 1:4



## Si on voulait copier...

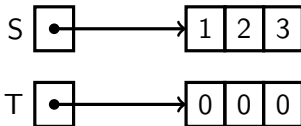
```
public static void main(String[] args)
{
    int[] S = {1, 2, 3};
    int[] T = new int[3];
    for (int i = 0; i < S.length; i++)
        T[i] = S[i];
}
```



## Si on voulait copier...

```
public static void main(String[] args)
{
    int[] S = {1, 2, 3};
    int[] T = new int[3];
    for (int i = 0; i < S.length; i++)
        T[i] = S[i];
}
```

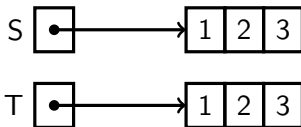
Avant le **for**



## Si on voulait copier...

```
public static void main(String[] args)
{
    int[] S = {1, 2, 3};
    int[] T = new int[3];
    for (int i = 0; i < S.length; i++)
        T[i] = S[i];
}
```

Après le **for**



## Tester l'égalité

- Attention: == teste l'égalité des contenus des boîtes.
  - ▶ si x et y ne sont pas des références, pas de problème.
  - ▶ sinon, attention.

```
int [] S={1,2,3};  
int [] T={1,2,3};  
int [] U=T;  
System.out.println((S==T) + "," + (T==U));
```

Affiche: *false,true*

- Pour tester l'égalité des contenus pointés, utiliser la méthode equals.

```
int [] T = {1, 2, 3} ;  
boolean b = T.equals(new int [] {1, 2, 3});  
System.out.println(b);
```

Affiche: *true*

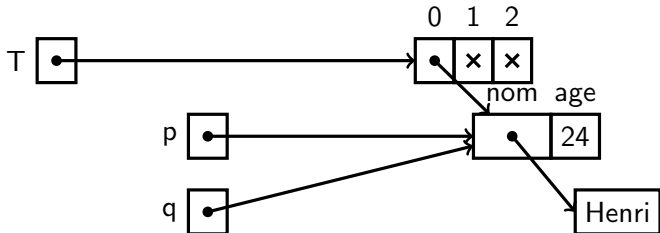
## Exercise

```
Personne[] T = new Personne[3];  
Personne p = new Personne();  
p.nom = "Henri"; p.age = 22;  
T[0] = p;  
System.out.println(T[0].age);  
Personne q = T[0];  
q.age = 24;  
System.out.println(T[0].age);
```

Affichage?

22

24



# Appels de fonctions et paramètres

## ■ Principe:

- ▶ Chaque appel de méthode crée ses propres variables.
- ▶ Les variables correspondant aux paramètres sont initialisées par l'appel.

## ■ Exemple:

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

Affiche:  $f(2)=4$


# Explication

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

# Explication

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

Appel de main():



# Explication

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

Appel de main():

x ?

# Explication

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

Appel de main():

x 2

# Explication

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

Appel de main():

x

r

# Explication

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

Appel de main():

x

r

Appel de f(2):

# Explication

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

Appel de main():

x

r

Appel de f(2):

x

# Explication

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

Appel de main():

x

r

Appel de f(2):

x

# Explication

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

Appel de main():

x 2

r ?

x 4

Retour de  $f(2)$  avec la valeur 4

# Explication

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

Appel de main():

x

r

# Explication

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

Appel de main():

x

r

Affiche:  $f(2)=4$

# Explication

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

Affiche:  $f(2)=4$

## Exercice (pas très joli)

Et si l'on voulait modifier le  $x$  de main?

Solution: ne pas passer  $x$ , mais une référence.

```
class MonInt {int val;}

static int f(MonInt x) {
    x.val = x.val + 2;
    return x.val;
}

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

Affiche:  $f(4)=4$

Les variables MonInt de f et de Main sont toujours distinctes, mais elles pointent sur le même entier qui est modifié.

# Aujourd'hui

Rappels de Java

**Classes en Java**

Structures Dynamiques

# Classes JAVA

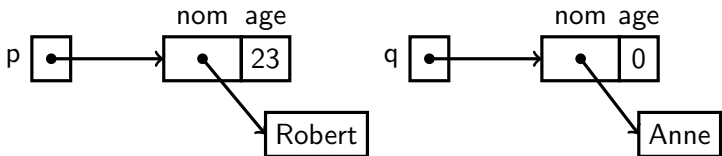
- Une classe JAVA permet de définir un nouveau type (non-primitif) à partir de types connus, ou déjà construits.
- Une classe peut contenir:
  - ▶ des attributs (variables),
  - ▶ des méthodes,
  - ▶ des constructeurs,
  - ▶ d'autres classes.

# Constructeurs

- Un constructeur d'une classe est une méthode, de même nom que la classe, sans type de retour.
- Un constructeur par défaut (sans argument) est fourni à toute classe. Il initialise tous les champs à leur valeur par défaut.
- On peut définir plusieurs constructeurs, s'ils n'ont pas la même signature (même nombre d'arguments avec mêmes types).
- Attention: si l'on définit un constructeur, le constructeur par défaut n'est plus défini.
- Les constructeurs sont appelés par **new**.

```
class Personne
{ String nom;
  int age;
  Personne(String s, int k)
  { nom = s;
    age = k;  }
  Personne(String s)
  { nom = s;}
}
```

```
Personne p = new Personne("Robert",23);
Personne q = new Personne("Anne");
```



# Variable **static**

- Une variable déclarée **static** est
  - ▶ une variable de classe: commune à toutes les instances de la classe.
  - ▶ utilisable sous la forme `NomDeClasse.NomDeVariable`.
    - Exemple: `Math.Pi`, `Integer.MAX_VALUE`
- Une variable non déclarée **static** (le défaut)
  - ▶ chaque objet instance de la classe en possède sa propre version.
    - Exemple: `nom`, `age` de la classe `Personne`.
  - ▶ Elle est utilisable
    - sous la forme `i.nom` où `i` est une instance de la classe, Exemple: `p.nom`.
    - ou sous la forme `nom` comme synonyme de `this.nom` (donc à l'intérieur d'une méthode non **static**).

# Méthode **static**

- Une méthode **static** est
  - ▶ une méthode de classe: elle peut être utilisée sans référence à une instance particulière de la classe.
  - ▶ utilisable sous la forme `NomDeClasse.nomDeMethode()`.
    - Exemple: `Math.abs()`.
  - ▶ Elle n'a donc pas accès aux variables non **static**.
- Une méthode non **static** (le défaut)
  - ▶ est une méthode d'objet: elle est toujours utilisée en référence à une instance de la classe.
  - ▶ On peut y utiliser dans sa définition le mot clé **this** pour désigner l'objet courant.
  - ▶ utilisable sous la forme `i.nomDeMethode()`, où `i` est une instance de la classe,
    - Exemple: `p.age()`, où `age()` est une méthode de la classe `Personne`.

# Méthode `static`

- Une méthode `static` est
  - ▶ une méthode de classe: elle peut être utilisée sans référence à une instance particulière de la classe.
  - ▶ utilisable sous la forme `NomDeClasse.nomDeMethode()`.
    - Exemple: `Math.abs()`.
  - ▶ Elle n'a donc pas accès aux variables non `static`.
- Une méthode non `static` (le défaut)
  - ▶ est une méthode d'objet: elle est toujours utilisée en référence à une instance de la classe.
  - ▶ On peut y utiliser dans sa définition le mot clé `this` pour désigner l'objet courant.
  - ▶ utilisable sous la forme `i.nomDeMethode()`, où `i` est une instance de la classe,
    - Exemple: `p.age()`, où `age()` est une méthode de la classe `Personne`.
- On peut écrire `NomDeMethode()` pour `this.NomDeMethode()` lorsque cela a un sens (donc à l'intérieur d'une méthode non `static`).

## Exemple

```
class Point {  
    static float ox;  
    float x;  
    Point (float a)    { x=a;}  
    static void FixeOrigine(float x)    {ox=x;}  
    float DistanceOrigine() {return x-ox;}  
    void Ajoute (float dx)    {x = x + dx;}  
    static Point AjouteS (Point p, float dx)  
        {return new Point(p.x+dx);}  
}
```

```
Point p = new Point(1);  
Point q = new Point(3);  
Point.FixeOrigine(5);  
p.Ajoute(1);  
q = Point.AjouteS(q,1);  
System.out.println(p.ox+" "+p.x+" "+q.ox+" "+q.x);
```

Affiche: 5.0 2.0 5.0 4.0

# Conséquence

	Peut accéder variable	
	static?	non static?
Méthode static	oui	non
Méthode non static	oui	oui

```
class Point {  
    static float ox;  
    float x;  
    static void Affichex() {  
        System.out.println(x);  
    }  
}
```

Affiche: *non-static variable x cannot be referenced from a static context*

# Aujourd'hui

Rappels de Java

Classes en Java

Structures Dynamiques

# Pourquoi des structures dynamiques

Quel est le problème avec les tableaux?

- Les tableaux, même s'ils sont dynamiques en JAVA, sont un peu rigides.
- On doit connaître à l'avance la taille du tableau.

Exemple: On veut gérer une liste de mots. On veut pouvoir

- afficher les mots
- ajouter des mots
- rechercher si un mot est dans liste
- supprimer des mots

## Avec un tableau?

```
class ListeDeMots {  
    // On commence avec au plus 2 mots  
    String[] mots = new String[2];  
    // nb de mots dans la liste  
    int n = 0;  
  
    boolean ajouter(String mot) { ... }  
    void enlever(String mot) { ... }  
    boolean contient(String mot){ ... }  
    void afficher() { ... }  
  
    // renvoie la position du mot dans mots (-1 si inconnu)  
    int index(String mot) { ... }  
    // double la taille de la liste  
    void doubler() { ... }  
}
```

```
int index(String mot) {
    for (int i = 0; i < n; i++)
        if (mots[i].equals(mot))
            return i;
    return -1; // mot inconnu
}

boolean contient(String mot) {
    return (index(mot) ≠ -1);
}

void afficher() {
    for (int i = 0; i < n; i++)
        System.out.print(mots[i] + " ");
}
```

```
boolean ajouter(String mot) {  
    if (n == mots.length) doubler();  
    mots[n++] = mot;  
    return true;  
}  
void enlever(String mot) {  
    int i = index(mot);  
    if (i ≥ 0) {  
        for (int j=i+1; j < n; j++)  
            mots[j-1] = mots[j];  
        n--; } }
```

```
void doubler() {  
    String[] mots2 =  
        new String[2*mots.length];  
    for (int i=0; i < mots.length; i++)  
        mots2[i] = mots[i];  
    mots = mots2;  
}
```

Défauts de cette approche?

# Liste chaînée

- Une liste chaînée est une suite de couples formés d'un élément et de la référence (adresse) vers l'élément suivant.
- Opérations usuelles sur les listes
  - ▶ *Créer une liste vide.*
  - ▶ *Tester si une liste est vide.*
  - ▶ *Afficher une liste.*
  - ▶ *Ajouter un élément (en tête de liste).*
  - ▶ *Rechercher un élément.*
  - ▶ *Supprimer un élément.*
  - ▶ *Afficher, retourner, concaténer, dupliquer, ...*

# Implémentation en Java

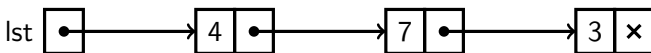
```
class Liste {  
    int contenu;  
    Liste suivant;  
    Liste (int x, Liste a) {  
        contenu = x;  
        suivant = a;  
    }  
}
```

```
Liste lst;
```

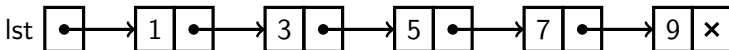
- Pour une liste vide: `lst` vaut **null**.
- Pour une liste non vide:
  - ▶ `lst.contenu` est le premier élément (la tête de la liste).
  - ▶ `lst.suivant` est la liste privée de son premier élément (la queue de la liste).

## En mémoire.

```
Liste lst = new Liste(4, new Liste (7, new Liste (3, null)));
```



```
Liste lst = new Liste(1, new Liste (3,  
    new Liste (5, new Liste (7, new Liste (9, null))));
```



# Vide, Tête, Queue

```
static Liste faireListeVide()  
{  
    return null;  
}
```

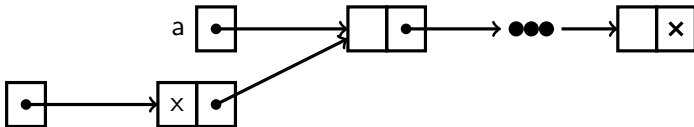
```
static boolean estVide(Liste a)  
{  
    return (a==null);  
}
```

```
static int tete(Liste a) {  
    return a.contenu;  
}
```

```
static Liste queue(Liste a) {  
    return a.suivant;  
}
```

## Ajouter un élément en tête

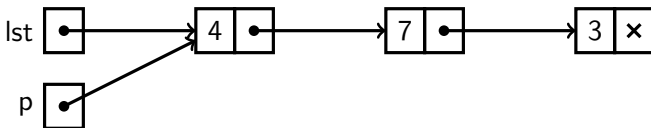
```
static Liste ajouter (int x, Liste a) {  
    return new Liste (x, a);  
}
```



## Parcourir une liste

```
for (Liste p = ...; p ≠ null; p = p.suivant)
{
// traiter p.contenu
}
```

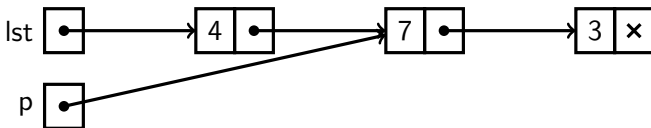
```
Liste p = ...;
while (p ≠ null) {
// traiter p.contenu
p = p.suivant;
}
```



## Parcourir une liste

```
for (Liste p = ...; p ≠ null; p = p.suivant)
{
// traiter p.contenu
}
```

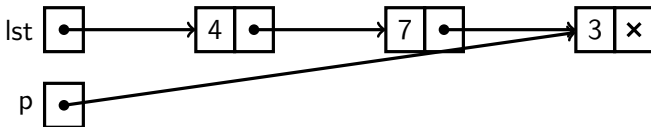
```
Liste p = ...;
while (p ≠ null) {
// traiter p.contenu
p = p.suivant;
}
```



## Parcourir une liste

```
for (Liste p = ...; p ≠ null; p = p.suivant)
{
// traiter p.contenu
}
```

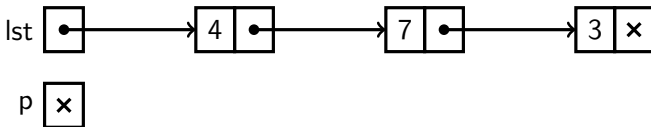
```
Liste p = ...;
while (p ≠ null) {
// traiter p.contenu
p = p.suivant;
}
```



## Parcourir une liste

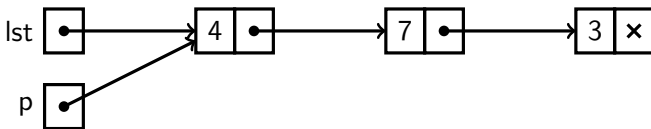
```
for (Liste p = ...; p ≠ null; p = p.suivant)
{
  // traiter p.contenu
}
```

```
Liste p = ...;
while (p ≠ null) {
  // traiter p.contenu
  p = p.suivant;
}
```



## Afficher une liste (itératif)

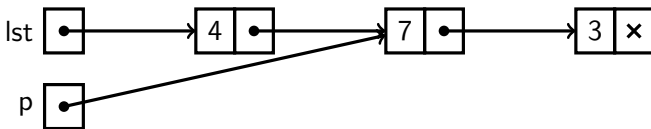
```
static void Afficher(Liste p) {  
    while (p ≠ null) {  
        System.out.print(p.contenu);  
        if (p.suivant ≠ null) System.out.print(",");  
        p = p.suivant;  
    }  
    System.out.println();  
}
```



Affiche: 4,7,3

## Afficher une liste (itératif)

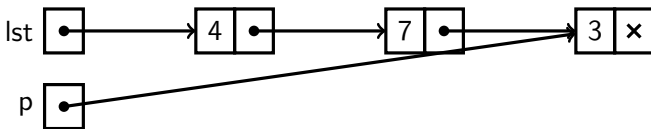
```
static void Afficher(Liste p) {  
    while (p ≠ null) {  
        System.out.print(p.contenu);  
        if (p.suivant ≠ null) System.out.print(",");  
        p = p.suivant;  
    }  
    System.out.println();  
}
```



Affiche: 4,7,3

## Afficher une liste (itératif)

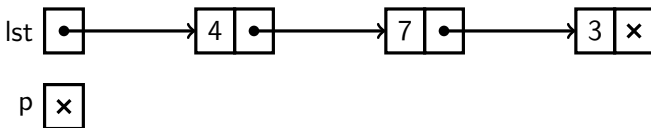
```
static void Afficher(Liste p) {  
    while (p ≠ null) {  
        System.out.print(p.contenu);  
        if (p.suivant ≠ null) System.out.print(",");  
        p = p.suivant;  
    }  
    System.out.println();  
}
```



Affiche: 4,7,3

## Afficher une liste (itératif)

```
static void Afficher(Liste p) {  
    while (p ≠ null) {  
        System.out.print(p.contenu);  
        if (p.suivant ≠ null) System.out.print(",");  
        p = p.suivant;  
    }  
    System.out.println();  
}
```



Affiche: 4,7,3

## Afficher une liste (récursif)

Une liste d'entiers est solution de l'équation

$$L = \text{null} \uplus (\text{int} \times L)$$

```
static void Afficher(Liste p) {  
    if (p== null)  
        System.out.println();  
    else {  
        System.out.print(p.contenu);  
        if (p.suivant != null) System.out.print(",");  
        Afficher(p.suivant);  
    }  
}
```

## Longueur d'une liste (itératif)

```
static int longueur(Liste l) {  
    int n = 0;  
    for (; l ≠ null; l=l.suivant)  
        n = n+1;  
    return n;  
}
```

## Longueur d'une liste (récursif)

Equations récursives, en notant  $\emptyset$  la liste vide, et en notant  $(x, L)$  une liste non-vide.

$$\begin{aligned} \text{longueur}(\emptyset) &= 0 \\ \text{longueur}((x, L)) &= 1 + \text{longueur}(L) \end{aligned}$$

```
static int longueur(Liste l) {  
    if (l == null)  
        return 0;  
    return 1 + longueur(l.suivant);  
}
```

# Tester l'appartenance

## ■ Version itérative:

```
static boolean Dans(int x, Liste a) {  
    while (a ≠ null) {  
        if (a.contenu == x)  
            return true;  
        a = a.suivant;  
    }  
    return false;  
}
```

## ■ Version récursive:

```
static boolean Dans(int x, Liste a) {  
    if (a == null)  
        return false;  
    if (a.contenu == x)  
        return true;  
    return Dans(x, a.suivant);  
}
```

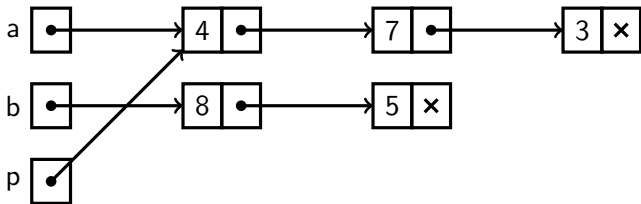
# Concaténer deux listes (version itérative, mutable)

- Objectif:

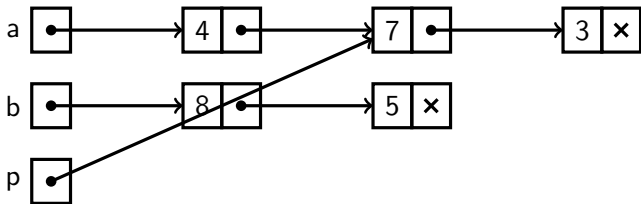
- ▶ ajouter les éléments de la liste b à la fin de ceux de la liste a.

```
static Liste nappend2(Liste a, Liste b) {  
    if (a==null)  
        return b;  
    Liste p;  
    for (p=a; p.suivant !=null; p=p.suivant)  
        {}  
    p.suivant = b;  
    return a;  
}
```

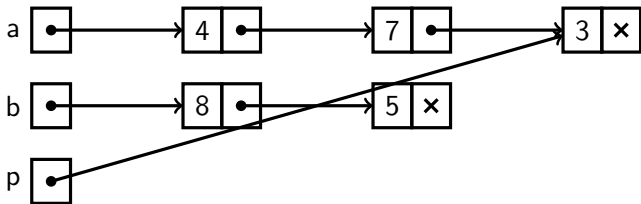
## Concaténer deux listes (version itérative, mutable)



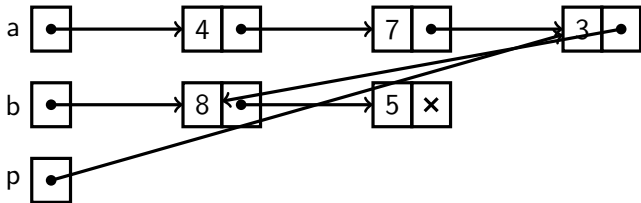
# Concaténer deux listes (version itérative, mutable)



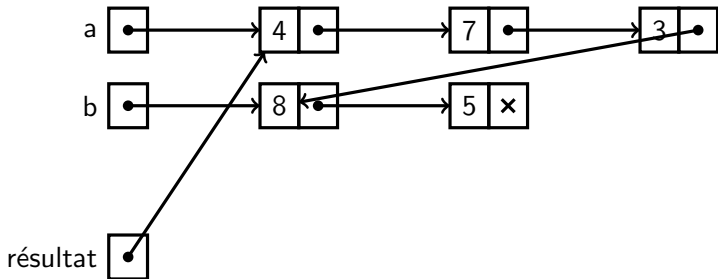
# Concaténer deux listes (version itérative, mutable)



# Concaténer deux listes (version itérative, mutable)



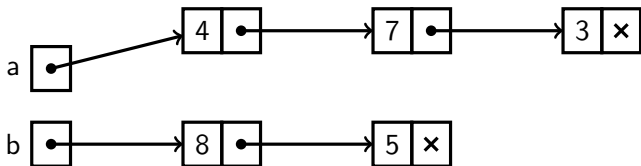
# Concaténer deux listes (version itérative, mutable)



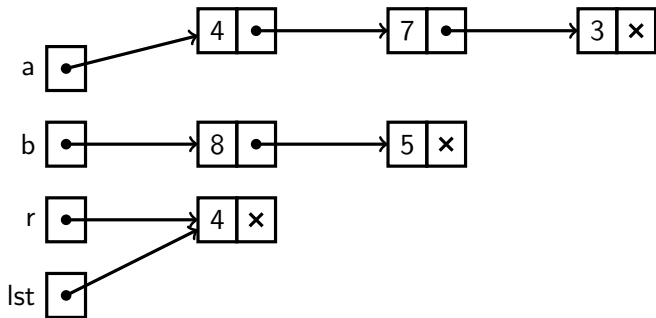
## Concaténer deux listes (version itérative, persistante)

```
static Liste append2(Liste a, Liste b) {  
    if (a== null) return b;  
    Liste r = new Liste(a.contenu, null) ;  
    Liste lst = r;  
    for (a= a.suivant; a ≠ null; a = a.suivant) {  
        lst.suivant = new Liste(a.contenu, null);  
        lst = lst.suivant;  
    }  
    lst.suivant = b;  
    return r;  
}
```

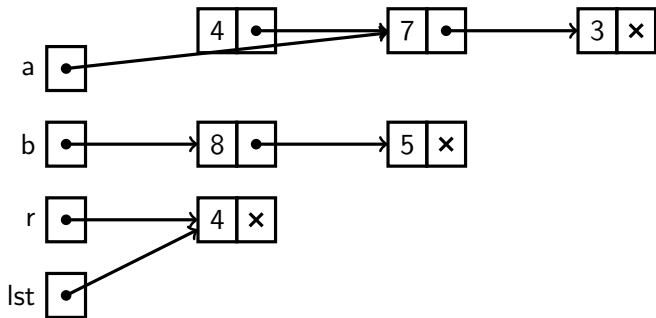
## Concaténer deux listes (version itérative, persistante)



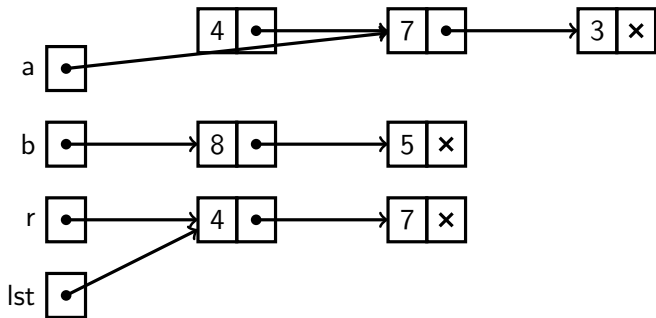
# Concaténer deux listes (version itérative, persistante)



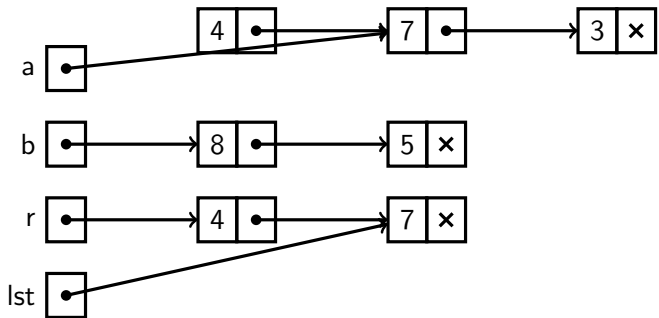
# Concaténer deux listes (version itérative, persistante)



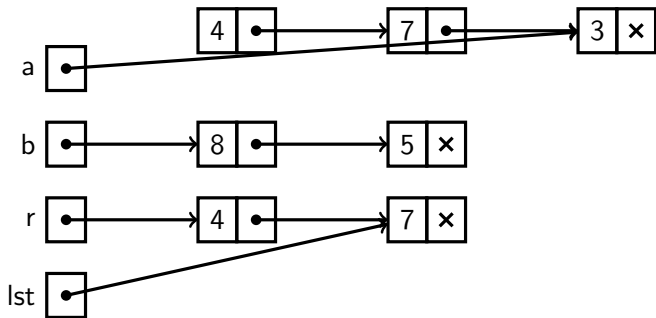
# Concaténer deux listes (version itérative, persistante)



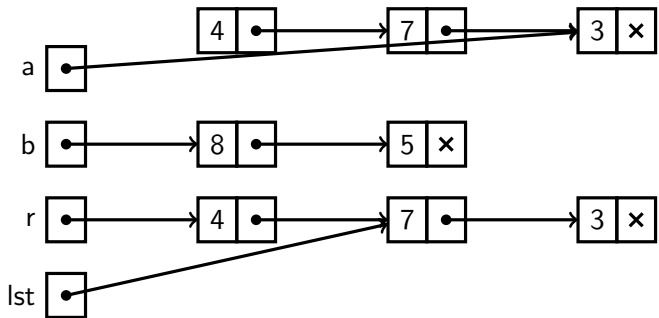
# Concaténer deux listes (version itérative, persistante)



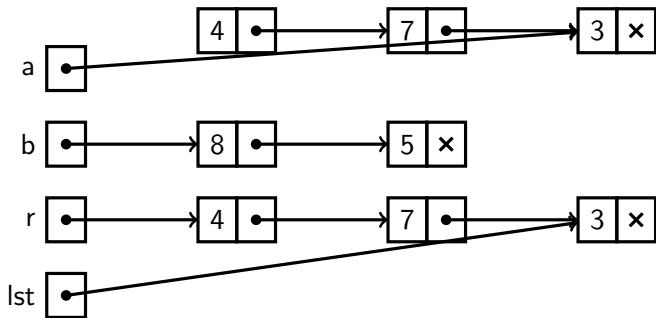
# Concaténer deux listes (version itérative, persistante)



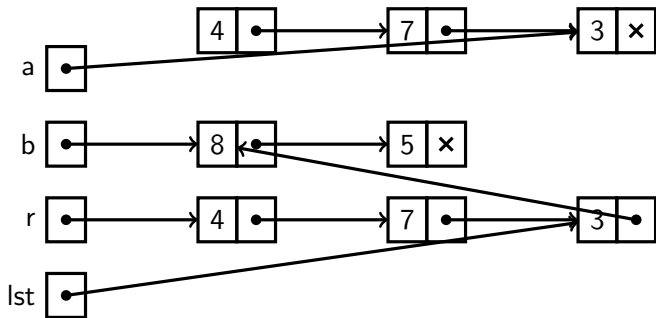
# Concaténer deux listes (version itérative, persistante)



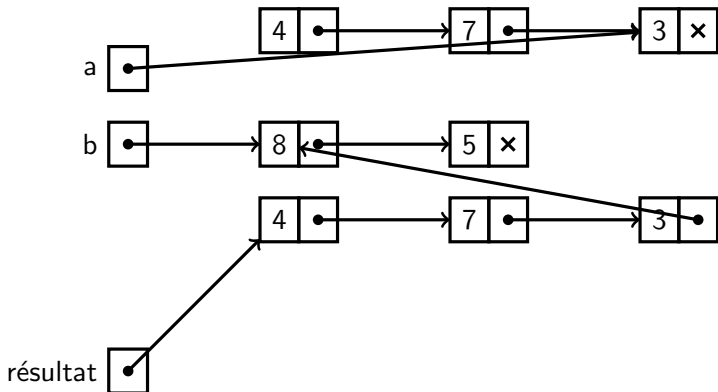
# Concaténer deux listes (version itérative, persistante)



# Concaténer deux listes (version itérative, persistante)



# Concaténer deux listes (version itérative, persistante)



# Concaténer deux listes (version récursive, persistante)

- Une liste d'entiers est solution de l'équation

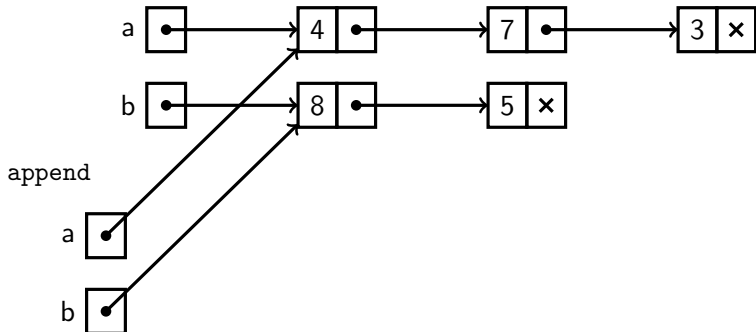
$$L = \text{null} \uplus (\text{int} \times L)$$

- Equations récursives

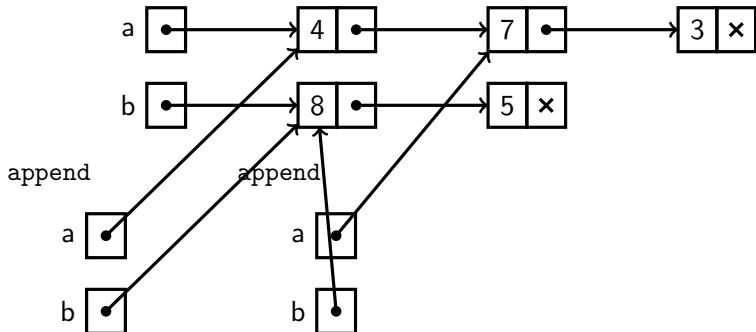
$$\begin{aligned} \text{append}(\emptyset, b) &= b \\ \text{append}((x, L), b) &= (x, \text{append}(L, b)) \end{aligned}$$

```
static Liste append(Liste a, Liste b) {  
    if (a== null)  
        return b;  
    else  
        return new Liste(a.contenu, append(a.suivant,b));  
}
```

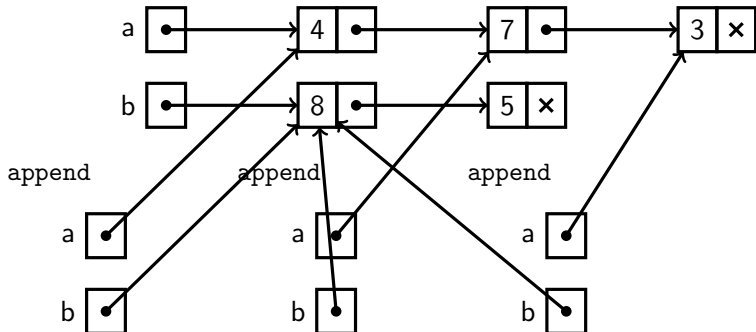
## Concaténer deux listes (version récursive, persistante)



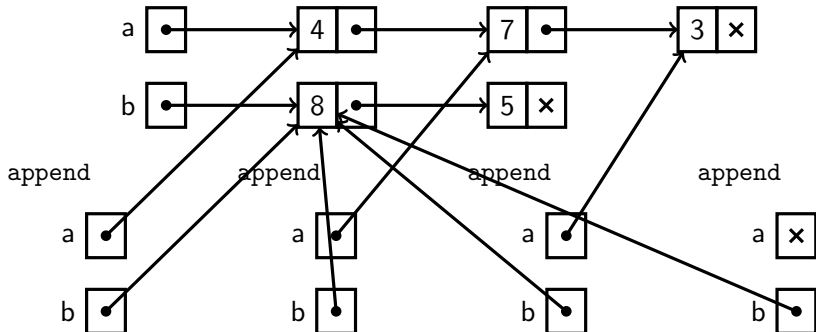
# Concaténer deux listes (version récursive, persistante)



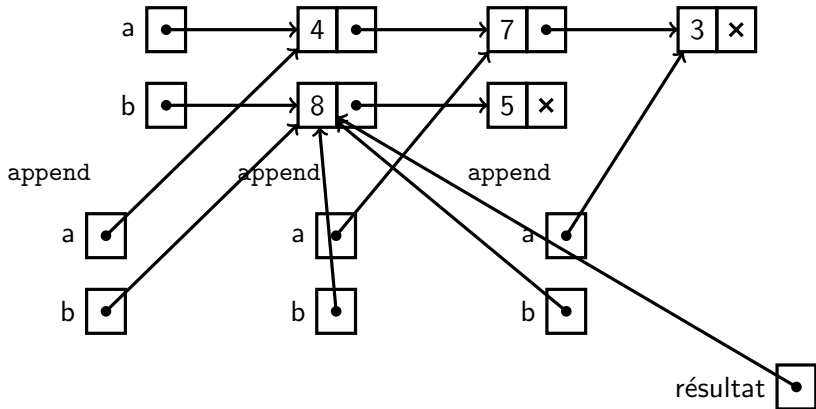
# Concaténer deux listes (version récursive, persistante)



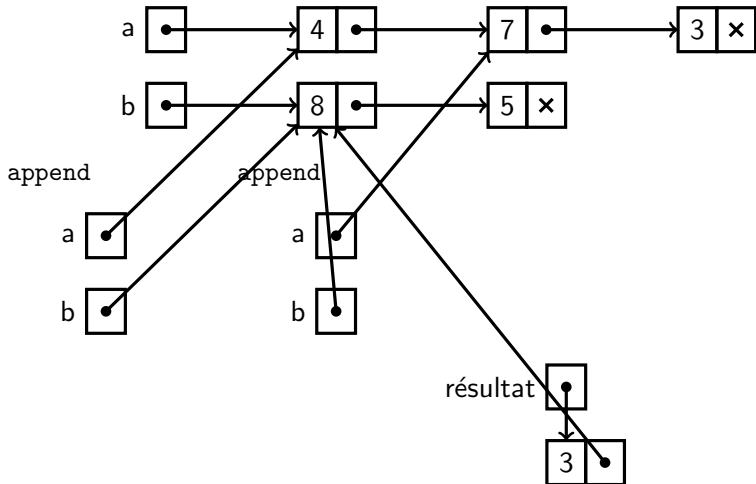
# Concaténer deux listes (version récursive, persistante)



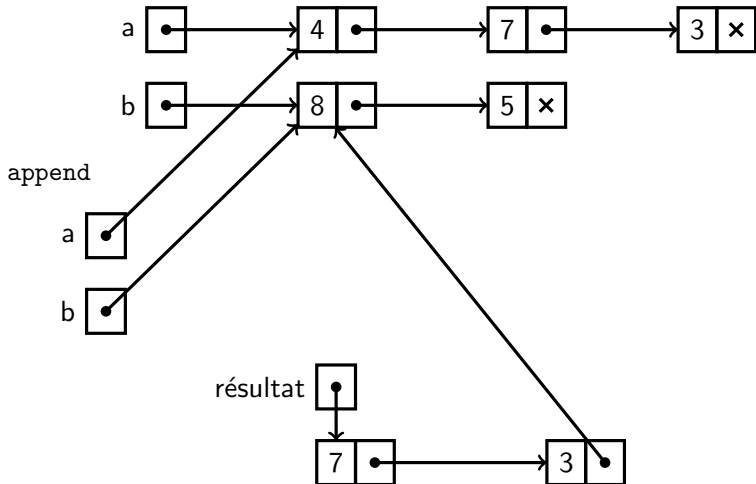
# Concaténer deux listes (version récursive, persistante)



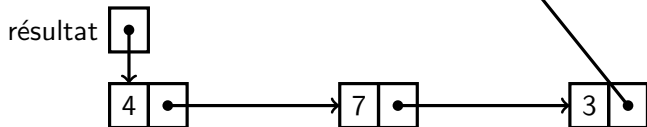
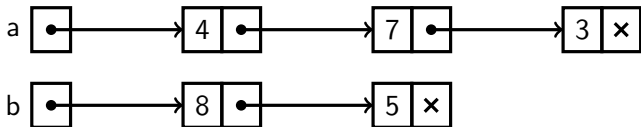
# Concaténer deux listes (version récursive, persistante)



# Concaténer deux listes (version récursive, persistante)



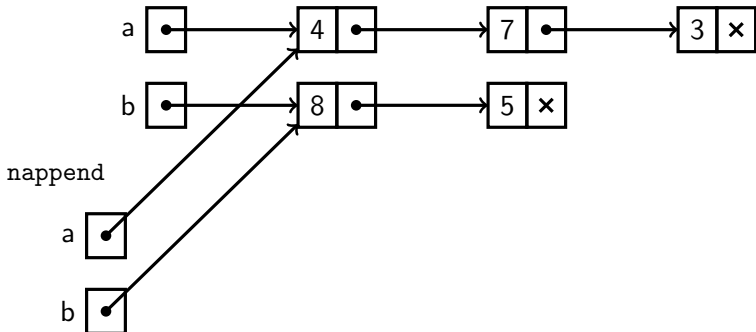
## Concaténer deux listes (version récursive, persistante)



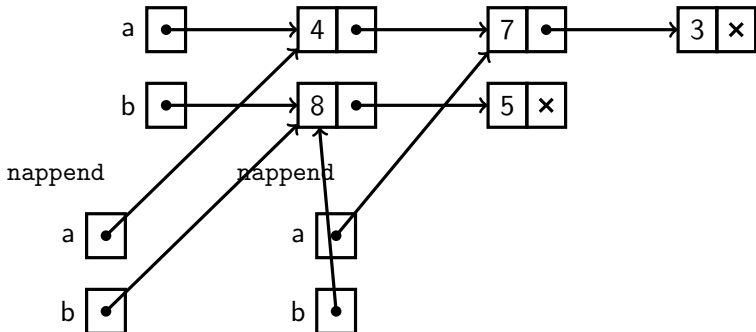
## Concaténer deux listes (version récursive, mutable)

```
static Liste nappend(Liste a,Liste b) {  
    if (a==null)  
        return b;  
    else {  
        a.suivant = nappend(a.suivant,b);  
        return a;  
    }  
}
```

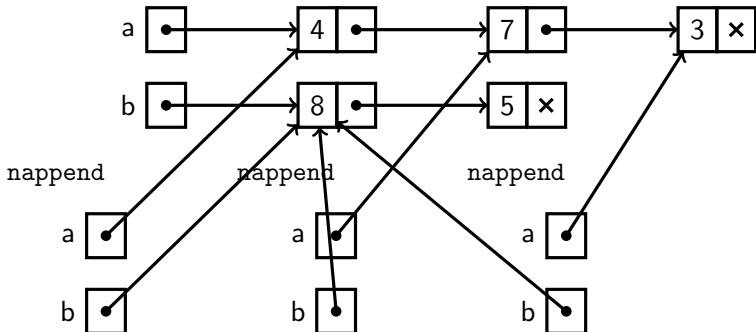
## Concaténer deux listes (version récursive, mutable)



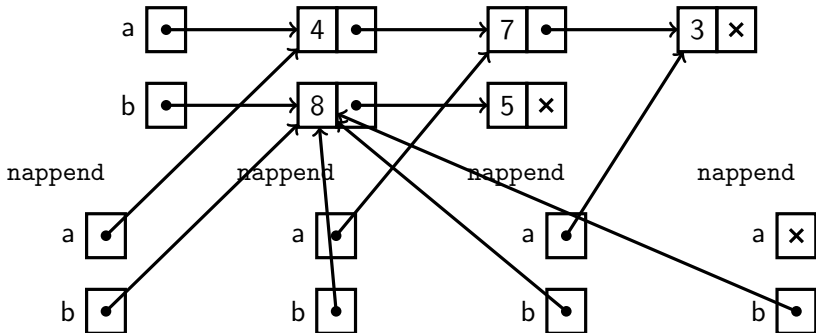
## Concaténer deux listes (version récursive, mutable)



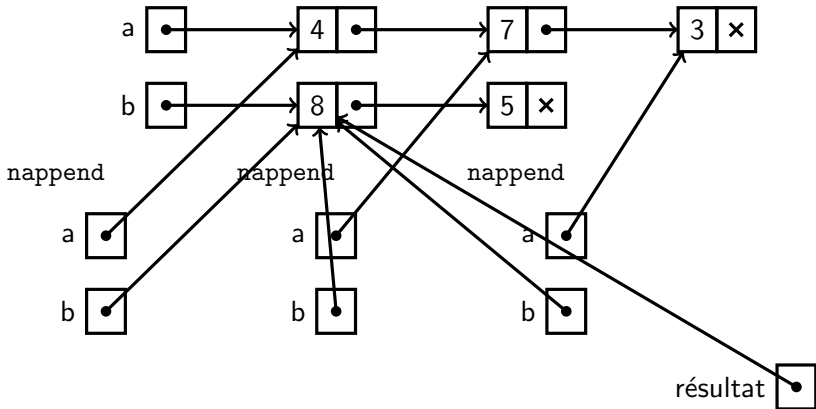
# Concaténer deux listes (version récursive, mutable)



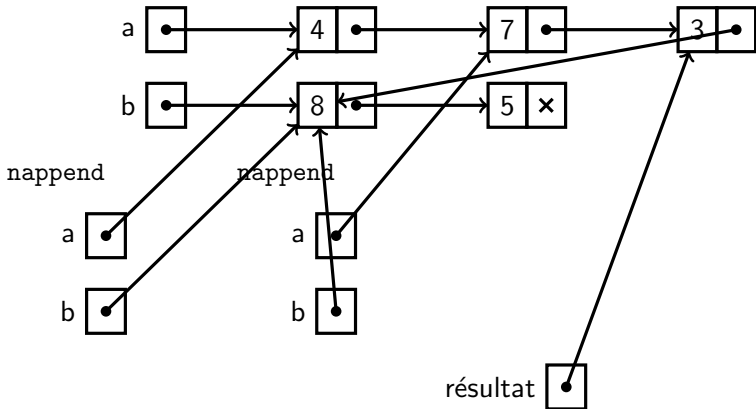
# Concaténer deux listes (version récursive, mutable)



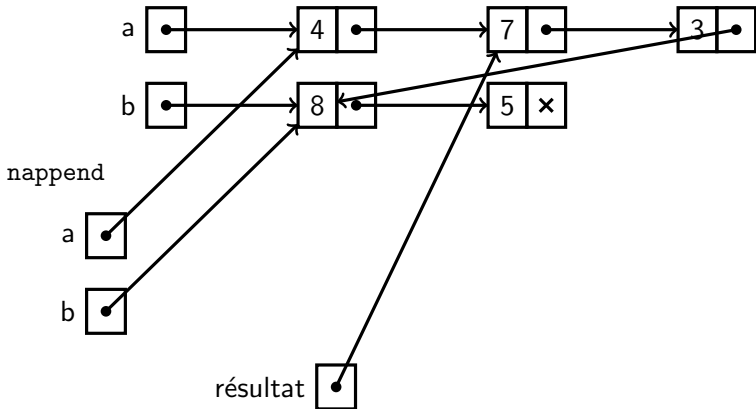
# Concaténer deux listes (version récursive, mutable)



# Concaténer deux listes (version récursive, mutable)



## Concaténer deux listes (version récursive, mutable)



## Concaténer deux listes (version récursive, mutable)

