

INF421-B

Bases de la programmation et de l'algorithmique

(Bloc 1/ 9)

Philippe Baptiste

CNRS LIX, École Polytechnique

4 novembre 2010

Objectifs généraux (1) : Algorithmique

- ▶ Structures dynamiques (listes, files, piles, tables de hachage, arbres, graphes)
- ▶ Algorithmes élémentaires sur ces structures (création, parcours, recherche, *etc.*)
- ▶ Des applications
 - ▶ évaluation d'une expression arithmétique,
 - ▶ gestion d'un dictionnaire,
 - ▶ compression,
 - ▶ calcul d'un arbre couvrant,
 - ▶ calcul de chemins, *etc.*

Organisation du cours

- ▶ 9 blocs, soit 9 vendredis : Amphi de 10h30 à 12h00 et l'après-midi, TP
 - ▶ Enseignement par groupes : 13h30 – 15h30 puis 15h45 – 17h45
 - ▶ Philippe Baptiste (Amphi 10h30 à 12h)
 - ▶ Jean-Christophe Filliâtre et Romain Lebreton
- ▶ La page du cours www.dix.polytechnique.fr/INF421 et vos questions à Philippe.Baptiste@polytechnique.fr et/ou aux enseignants de l'équipe.
- ▶ L'évaluation
 - ▶ Un CC
 - ▶ Un TP noté (le 5ème) avec une application du cours
 - ▶ $\max(CC, \frac{3}{4}CC + \frac{1}{4}TP)$

Objectifs généraux (2) : Programmation et modélisation

Un langage support "Java 1.5" pour quelques concepts fondamentaux de programmation

- ▶ Les **références**
- ▶ Les types récurifs
- ▶ Une programmation structurée (\rightarrow POO)

Programmer c'est modéliser un problème, structurer les données, concevoir des algorithmes, coder et tester.

Méthode : Des révisions et un rythme soutenu mais raisonnable

Aujourd'hui

Types, variables, références

Classes

Structures dynamiques : les listes

Types Primitifs

Types **primitifs** :

- ▶ Les booléens (**boolean**) codés sur 1 bit : **true** ou **false**
- ▶ Les entiers (**byte**) codés sur 1 octet : $[-128, 127]$
- ▶ Les entiers (**short**) codés sur 2 octets : $[-32768, 32767]$
- ▶ les caractères **char** codés sur 2 octets : unicode
- ▶ Les entiers (**int**) codés sur 4 octets : $[-2147483648, 2147483647]$
- ▶ Les entiers (**long**) codés sur 8 octets : $[-910^{18}, 910^{18}]$
- ▶ Les flottants (**float**) codés sur 4 octets
- ▶ Les flottants (**double**) codés sur 8 octets

Les données de type primitif sont manipulées par **valeur**

Les types

- ▶ Toujours déclarer et typer les variables
- ▶ La création peut se faire au même moment ou plus tard
- ▶ Sauf pour les variables “primitives”, la création passe par **new**
- ▶ Java utilise un garbage collector (ramasse miettes) qui gère la mémoire

Distinction

- ▶ Types primitifs
- ▶ les autres

Les types qui ne sont pas primitifs

Les autres types sont des **objets**, des **tableaux** (en java, les tableaux sont aussi des objets) ou des chaînes de caractères.

```
class Personne {
    String nom;
    int age;
    int[] x;
    String nom;
}
```

Pour ces types, la valeur des variables est une référence, *i.e.*, l'adresse mémoire où est logée la donnée accompagnée du type de la donnée.

Les types qui ne sont pas primitifs

```
double val = 0.987654321 ;
Personne robert = new Personne();
byte[] tabByte = {7, 9, 8};
long[] tabLong = {7, 9, 8};
System.out.println("val = " + val);
System.out.println("robert = " + robert);
System.out.println("tabByte = " + tabByte);
System.out.println("tabLong = " + tabLong);
```

Sur mon PC

```
val = 0.987654321
robert = Personne@10b62c9
tabByte = [B@82ba41
tabLong = [J@923e30
```

Sur ma station

```
val = 0.987654321
robert = Personne@10148730
tabByte = [B@10124e70
tabLong = [J@1015eae0
```

Les types qui ne sont pas primitifs

“Personne@10148730” kesaco ?

- ▶ `println` cherche une “représentation” des données sous la forme d’une chaîne de caractères.
- ▶ Fournie par la méthode `toString` (définie pour tout objet)
- ▶ Par défaut, nom de la classe suivi de ‘@’ et de l’écriture hexadécimale du code de hachage de l’objet (des détails plus tard)
- ▶ On peut redéfinir la méthode `toString` comme dans la classe `String` :

Le code suivant

```
String s = "Hello there";
System.out.println("s = " + s);
```

nous donne `s = Hello there`.

Les types qui ne sont pas primitifs

```
byte[] tabByte = {7, 9, 8};
System.out.println("tabByte = " + tabByte);
```

“tabByte = [B@10124e70” kesako ?

- ▶ [→ Tableau
- ▶ B → de Bytes
- ▶ @ → Adresse, suivi de l’adresse en hexadécimal

Schématiquement,

- ▶ la valeur 7 (`tabByte[0]`) est stockée à 10124e70
- ▶ la valeur 9 (`tabByte[1]`) est stockée à 10124e71
- ▶ la valeur 8 (`tabByte[2]`) est stockée à 10124e72

La méthode “toString” (digression)

```
class Personne {
    String nom;
    int age;
    public String toString() {
        return "Mon nom est " + nom + " j'ai " + age + " ans";
    }}
```

```
class Test {
    public static void main(String[] args) {
        Personne robert = new Personne();
        System.out.println("robert = " + robert);
    }}
```

Et nous obtenons alors

```
robert = Mon nom est null j'ai 0 ans
```

Pourquoi “`null`” et pourquoi “0” ?

Une représentation des variables

Pour un type **primitif** : n 2

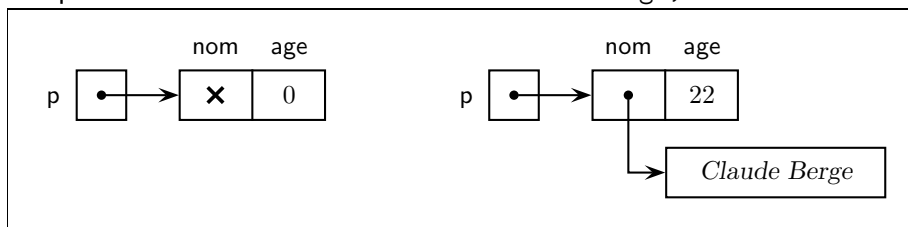
Pour un type **non-primitif** :

Quand la valeur d'une variable est une référence, c'est-à-dire l'adresse d'une donnée, la valeur numérique de cette adresse est remplacée par une flèche dirigée vers l'emplacement de la donnée

Une représentation des variables (objet)

```
class Personne { String nom; int age; }
class Test {
    public static void main(String[] args) {
        Personne p = new Personne();
        p.nom = "Claude Berge"; p.age = 22;
        System.out.print(p); // toString pas encore surchargé
        System.out.println(" " + p.nom + ", " + p.age + " ans"); }
}
```

Ce qui nous donne Personne@10b62c9 Claude Berge, 22 ans

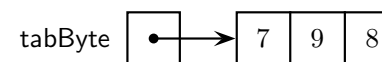


Une représentation des variables (Tableau)

```
byte[] tabByte = {7, 9, 8};
```



(a)



(b)

- ▶ tabByte vaut [B@10124e70 et à l'adresse @10124e70 nous avons un tableau de trois entiers (a)
- ▶ En résumé, tabByte pointe sur le tableau (b)

Déclaration et initialisation

- ▶ Avec l'instruction "**int** x;" on crée un entier (non-initialisé)
- ▶ Avec "**int** x = 98;" on crée un entier et on l'initialise

Pour les types simples, une variable non initialisée vaut 0 (ou false) pour un **boolean**.

Et pour les autres types ?

les champs d'un objet sont initialisés à une valeur par défaut.

- ▶ 0 pour les types primitifs numériques, et **false** pour les types booléen,
- ▶ et **null** pour tout champ objet.

Création des tableaux

- ▶ L'instruction "`int [] t;`" crée un tableau initialisé par `null`
- ▶ L'allocation des tableaux est dynamique, *i.e.*, la mémoire utilisée est allouée au moment de l'exécution.

- ▶ Plus difficile, les tableaux de tableaux

```
int [][] m = new int[4][6];
```

crée et initialise à 0 une matrice d'entier 4 * 6

Le `null`

- ▶ `null` est une valeur de référence particulière.
- ▶ Ce n'est la référence d'aucun objet.
- ▶ On peut affecter `null` à toute variable référence, mais aucun champ et aucune méthode d'objet n'est accessible par une variable qui contient cette valeur spéciale.

Erreur de manipulation : levée d'une exception de la classe `NullPointerException`. Il faut utiliser l'opérateur `new` pour la création d'un nouvel objet.

Rq. Le `null` est indiqué par une croix dans les figures.

Exercice

Que contient `mat` ?

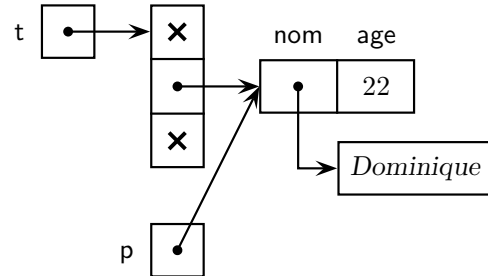
```
int [][] mat = new int[4][ ];
for (int i = 0; i < mat.length; i++) {
    mat[i] = new int[i];
    for (int j = 0; j < mat[i].length; j++)
        mat[i][j] = 1;
}
```

Exercice : Que fait le code suivant ?

```
class Personne {
    String nom;
    int age;
}
class Test {
    public static void main(String[] args) {
        Personne[] t = new Personne[3];
        Personne p = new Personne();
        p.nom = "Dominique";
        p.age = 22;
        t[1] = p;
        for (int i = 0; i < t.length; i++)
            System.out.print(t[i] + " ");
    }
}
```

Exercice : Que fait le code suivant ?

On obtient `null` `Personne@10b62c9` `null`. Graphiquement.



Piège 1 : les tests

“==” teste l’égalité des valeurs. **Attention**

- ▶ Pour les types simples aucun problème !
- ▶ Pour les autres les valeurs sont des références.

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

Et le résultat est **false** , **true** !

Solution : Utiliser `equals`

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

Et le résultat est **true**.

Piège 2 : les passages de paramètres

A chaque appel, une méthode crée ses propres variables (initialisées aux valeurs d’appel). Ainsi,

```
class Test {
    static int f(int x) {
        x = 2 * x;
        return x;
    }
    public static void main (String [] arg) {
        int x = 1;
        System.out.println("f = " + f(x) + ", x = " + x);
    }
}
```

retourne `f = 2`, `x = 1`. **x n’est pas modifié**. Comment faire pour “conserver” la modification ?

Piège 2 : les passages de paramètres (suite)

```
class MonInt { int val; }
class Test {
    static int f(MonInt z) {
        z.val = 2 * z.val;
        return z.val; }
    public static void main (String [] arg) {
        MonInt x = new MonInt();
        x.val = 1;
        System.out.println("f = " + f(x) + " x = " + x.val) ; }}}
```

Le `MonInt z` de `f` et la variable d’appel `x` restent distinctes mais elles contiennent des références qui sont égales.

Et donc, on a `f = 2` `x.val = 2`.

Aujourd'hui

Types, variables, références

Classes

Structures dynamiques : les listes

Constructeur

- ▶ Un constructeur d'une classe est une méthode, de même nom que la classe, sans type de retour.
- ▶ Les constructeurs servent à créer des objets. Ils sont appelés par **new**.
- ▶ Un constructeur **par défaut ou implicite** (pas d'arg) est fourni à toute classe. Il initialise tous les champs à leur valeur par défaut.
- ▶ Le constructeur par défaut n'est plus défini si un autre constructeur explicite a été défini.
- ▶ Plusieurs constructeurs peuvent coexister (signature \neq)

Une classe JAVA

Classe = bloc d'un programme qui définit un nouveau type non primitif. Une classe contient

- ▶ des attributs (ou variables),
- ▶ des méthodes,
- ▶ des initialisations, des constructeurs,
- ▶ d'autres classes (classes internes ou imbriquées).

Les méthodes et les variables sont

- ▶ de classe, ou **static**
- ▶ d'instance.

Exercice : Que fait le code suivant ?

```
class Complexe {
    double r; double i;
    Complexe() {r = 0.; i = 0.; }
    Complexe(int r1, int im) {r = ((double)r1); i = ((double)im); }
    Complexe(double r, double i) {
        this.r = r; this.i = i;}
    Complexe(double rayon, double omega) {
        r = rayon * Math.cos(omega); i = rayon * Math.sin(omega);}
    public String toString() {
        return r + " + i " + i;}
}
class Test { public static void main(String[] args) {
    Complexe zero = new Complexe();
    Complexe z1 = new Complexe(1, 1);
    Complexe z2 = new Complexe(0.8, 1.6);
    System.out.print(zero+"\n"+z1+"\n"+z2+"\n");}}
```

Variable **static**

- ▶ Une variable **static** est commune à tous les objets de la classe, c'est une **variable de classe**
- ▶ Une variable **static** est utilisable ainsi
NomdeClasse.nomDeVariable, e.g., Math.Pi,
Integer.MAX_VALUE
- ▶ Une variable non **static** a une instance par objet (e.g., le nom et l'âge dans la classe Personne)

Méthode **static**

- ▶ Une méthode **static** peut être utilisée sans référence à un objet particulier, *i.e.*, c'est une **méthode de classe**
- ▶ Une méth. **static** n'a pas accès aux variables non **static**
- ▶ On peut appeler une méthode **static** par
NomdeClasse.nomDeMethode(), e.g., Math.abs()
- ▶ Une méthode non **static** est toujours utilisée **en référence à un objet**. C'est une **méthode d'objet**.
- ▶ Le mot clé **this** est utilisé dans les méthodes **d'instance** pour désigner l'objet courant.

Exemple : `System.out.println("Robert Tarjan")`
La classe `System` contient une variable de classe de nom `out` et l'objet `System.out` appelle une méthode d'objet de nom `println()`.

Exercice : Que fait le code suivant ?

```
class Paire {
    static int[] x;
    int y; }
class Test {
    public static void main(String[] args) {
        Paire.x = new int[1];
        Paire.x[0] = 7;
        System.out.println("Paire.x = " + Paire.x +
            " Paire.x[0] = " + Paire.x[0]);
        Paire s = new Paire();
        s.y = 3;
        System.out.println("s.x = " + s.x + " s.y = " + s.y);
        Paire t = new Paire();
        t.y = 4;
        System.out.println("t.x = " + t.x + " t.y = " + t.y); }}
```

Exercice : Quelles sont les méthodes correctes ?

```
class Paire {
    static int x;
    int y;
    void affiche1_x() {
        System.out.println("s.x = " + this.x); }
    static void affiche2_x() {
        System.out.println("s.x = " + x); }
    void affiche1_y() {
        System.out.println("s.y = " + this.y); }
    static void affiche2_y() {
        System.out.println("y = " + y); }
    static void affiche3_y(Paire s) {
        System.out.println("s.y = " + s.y); }
}
```

Exercice : Quelles sont les méthodes correctes ?

```
class Paire {
    static int x;
    int y;
    static void affiche2_y() {
        System.out.println("s.y = " + y); }
}
```

Génère à la compilation :

```
non-static variable y cannot be referenced
from a static context
```

Aujourd'hui

Types, variables, références

Classes

Structures dynamiques : les listes

Méthodes et variables **final**

- ▶ La déclaration d'une variable **final** est toujours accompagnée d'une initialisation.
- ▶ Une variable **final** ne peut être modifiée.
- ▶ Une méthode **final** ne peut être surchargée. Les méthodes d'une classe **final** sont implicitement **final**.

Chercher les erreurs

```
class Test {
    static final int n = 100;
    static final int[] a = {1, 2, 3};
    public static void main(String args[]) {
        a[0] = 5;
        n = 9; }}
```

Des Structures Dynamiques : Pourquoi ?

- ▶ Limitation principale des tableaux : Taille fixée (→ surdimensionnement)
- ▶ Structures de données économes et simples.
 - ▶ économe = utilisation linéaire de la mémoire
 - ▶ simple = manipulation avec quelques opérateurs élémentaires

Solutions : listes, piles, files, arbres, graphes, etc..

Que faire quand un tableau est plein

```
class Dictionnaire {
    // On commence avec un dictionnaire d'au plus 5 mots
    String[] mots = new String[5];
    // nb de mots dans le dico
    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 mots
    void doubler() { ... }
}
```



Que faire quand un tableau est plein

```
boolean ajouter(String mot) {
    if (contient(mot)) return false;
    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;
}
```

Inconvénients ?



Que faire quand un tableau est plein

```
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] + " ");
}
```



La plus belle et la plus simple des structures dynamiques

Une **liste chaînée** est une suite finie de cellules formées (i) d'un élément et (ii) de l'adresse (référence) vers la cellule suivante

- ▶ En Java, une cellule = 2 champs : un champ pour le contenu, et un champ qui contient la référence vers la cellule suivante
- ▶ La dernière cellule contient une référence **null**
- ▶ L'accès à la liste se fait par la référence à la première cellule.
- ▶ Les opérations usuelles sur les listes
 - ▶ *créer* une liste vide.
 - ▶ *tester* si une liste est vide.
 - ▶ *ajouter* un élément en tête de liste.
 - ▶ *rechercher* un élément dans une liste.
 - ▶ *supprimer* un élément dans une liste.

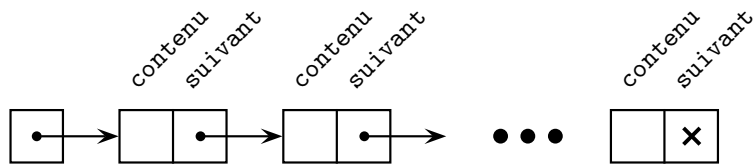


Une liste d'entiers en Java

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

Ainsi, `null` est la liste vide. Pour créer une liste d'un élément (2 ici) : `new Liste(2, null)`. Pour la liste des n premiers entiers pairs :

```
Liste l = null;
for (int i = n; i >= 1; i--)
    l = new Liste(2 * i, l);
```



Tête / Queue

La tête de la liste est le premier élément de la liste. La queue est la liste privée de sa tête.

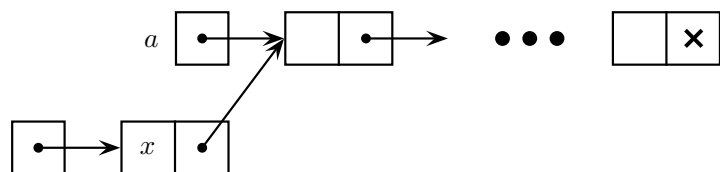
```
static int tete(Liste a) {
    return a.contenu;
}

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);
}
```



Afficher une liste

```
// Version récursive
static void afficher(Liste l) {
    if (l == null)
        System.out.println();
    else {
        System.out.print
            (l.contenu + " ");
        afficher(l.suivant);
    }
}

// Version itérative
static void afficher(Liste l) {
    while (l != null) {
        System.out.print
            (l.contenu + " ");
        l = l.suivant;
    }
    System.out.println();
}
```



Afficher une liste (3ème et dernière)

```
// Une version plus objet.
class Liste {
    int contenu;
    Liste suivant;
    Liste (int x, Liste a) {
        contenu = x;
        suivant = a;
    }
    public String toString() {
        if (suivant != null)
            return contenu + " " + suivant.toString();
        return contenu + " ";
    }
}
```

Tester l'appartenance

```
// Une version itérative
static boolean
estDans(int x, Liste a) {
    while (a != null) {
        if (a.contenu == x)
            return true;
        a = a.suivant;
    }
    return false;
}

// Une version récursive
static boolean
estDans(int x, Liste a) {
    if (a == null)
        return false;
    if (a.contenu == x)
        return true;
    return estDans(x, a.suivant);
}
```

Longueur d'une liste

Deux versions.

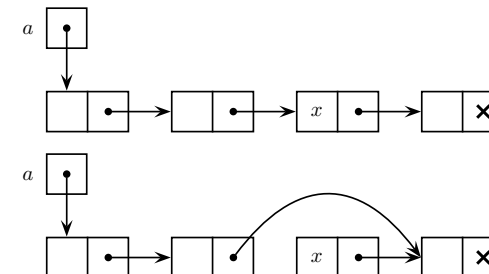
```
// Une version itérative
static int longueur(Liste l) {
    int n = 0;
    for (; l != null; l = l.suivant)
        n++;
    return n;
}

// Une version récursive
int longueur() {
    if (suivant != null)
        return 1 + suivant.longueur();
    return 0;
}
```

Supprimer

Supprimer un élément x = éliminer la 1^{ère} cellule qui le contient

- ▶ Trouver la cellule qui contient x
- ▶ modifier les champs (succ du préd \rightarrow succ de x).



Supprimer (en détruisant la liste)

```
// Une version itérative
static Liste supprimer(int x, Liste a) {
    if (a == null) return null;
    if (a.contenu == x) return a.suivant;
    Liste prec = a, cour = prec.suivant;
    for (; cour != null; prec = cour, cour = prec.suivant)
        if (cour.contenu == x) {
            prec.suivant = cour.suivant;
            return a;
        }
    return a;
}
```

- ▶ deux variables `prec` et `cour` qui contiennent les références vers la cellule courante et la cellule précédente.
- ▶ Invariant : `cour = prec.suivant`

Le $k^{\text{ème}}$ élément : Une version itérative

```
static int nth(int i, Liste p) {
    for (; p != null; p = p.suivant) {
        if (i == 0) return p.contenu;
        i--;
    }
    throw new Error("Erreur d'index ");
}
```

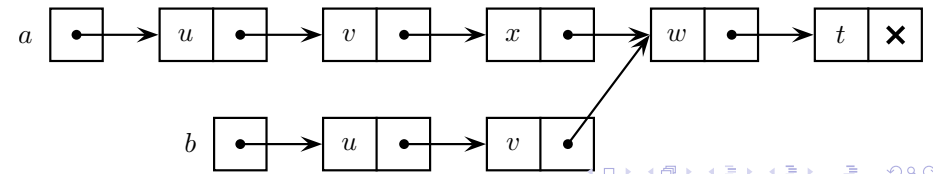
A propos de la dernière ligne `throw new Error()`;

- ▶ On est à la fin de la liste et $i \neq 0$ et le i initial n'était pas dans $[0, \text{length}(p)]$
- ▶ Il s'agit donc de "générer" une erreur
 - ▶ Des détails sur la gestion des erreurs plus tard

Supprimer (sans détruire la liste)

- ▶ Une fois la suppression effectuée, la liste initiale est "cassée"
- ▶ Pour ne pas casser la liste, procéder par copie (attention → allocation mémoire)

```
// Suppression non destructive
static Liste supprimer(int x, Liste a) {
    if (a == null) return a;
    if (a.contenu == x)
        return a.suivant;
    return new Liste(a.contenu, supprimer(x, a.suivant));
}
```



Le $k^{\text{ème}}$ élément : Une version itérative

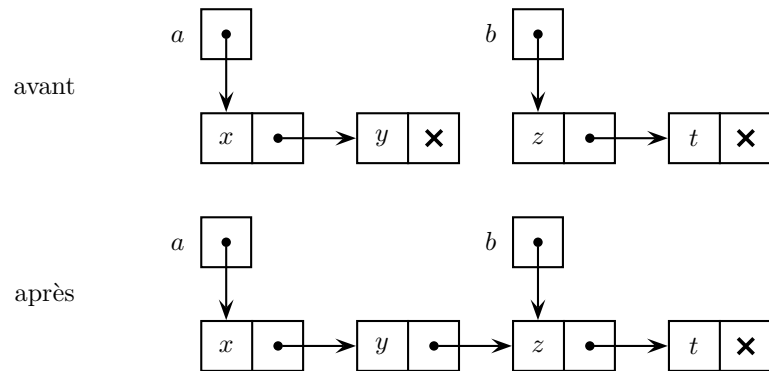
```
static int nth(int i, Liste p) {
    for (; p != null; p = p.suivant) {
        if (i == 0) return p.contenu;
        i--;
    }
    throw new Error("Erreur d'index dans nth");
} ...
Liste l = new Liste(7, new Liste(9, new Liste(3, new Liste(1, null))));
System.out.println("l[" + 3 + "] = " + nth(3, l));
System.out.println("l[" + 4 + "] = " + nth(4, l));
```

```
~phb > java test
l[3]= 1
```

```
Exception in thread "main" java.lang.Error: Erreur d'index
    at test.nth(test.java:222)
    at test.main(test.java:228)
```

Concaténation de deux listes

La concaténation de deux listes a et b produit une liste obtenue en ajoutant les éléments de la liste b à la fin de a



Inversion (rappel)

Construire une liste qui contient les mêmes éléments à l'envers : $(1, 2, 3) \rightarrow (3, 2, 1)$

```
static Liste inverser(Liste a) {
    Liste b = null;
    while (a != null) {
        b = new Liste(a.contenu, b);
        a = a.suivant;
    }
    return b;
}
```

Concaténation destructive de deux listes

Attention destructif !

```
static Liste dernier(Liste a) {
    if (a == null)
        return null;
    while (a.suivant != null)
        a = a.suivant;
    return a;
}
static Liste Concaténation(Liste a, Liste b) {
    if (a == null)
        return b;
    dernier(a).suivant = b;
    return a;
}
```

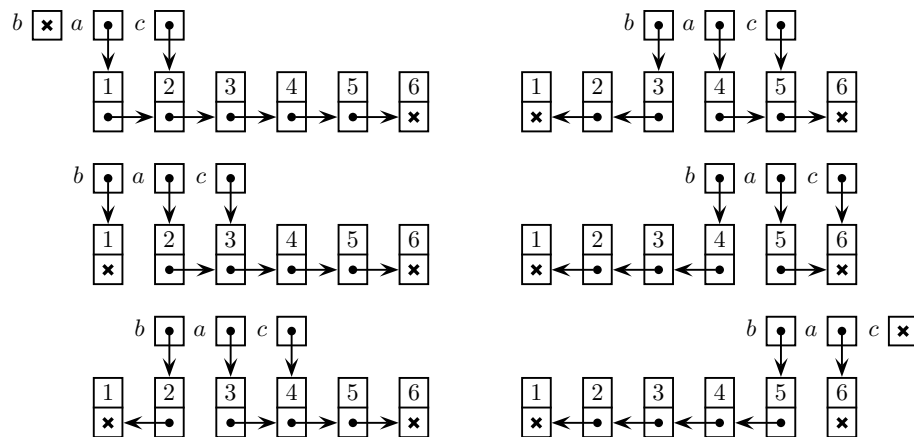
Inversion en place

On ne veut pas allouer la mémoire \rightarrow Inverser les flèches.

Attention destructif !

```
static Liste inverserEnPlace (Liste a) {
    Liste b = null;
    while (a != null) {
        Liste c = a.suivant;
        a.suivant = b; // inverse
        b = a;
        a = c;
    }
    return b;
}
```

Inversion en place



Fusionner deux listes triées

```

static Liste fusion(Liste l1, Liste l2) {
    if (l1 == null)
        return l2;
    if (l2 == null)
        return l1;
    if (l1.contenu < l2.contenu)
        return new Liste(l1.contenu, fusion(l1.suivant, l2));
    else
        return new Liste(l2.contenu, fusion(l1, l2.suivant));
}

```

Destructif ou pas ?

Fusionner deux listes triées

- ▶ Deux listes $\mathcal{L}_1, \mathcal{L}_2$ triées que l'on souhaite fusionner (en respectant le tri)
- ▶ $(1, 6, 12, 67, 98, 454), (5, 6, 11, 32, 123, 324, 444) \rightarrow (1, 5, 6, 6, 11, 12, 32, 67, 98, 123, 324, 444, 454)$
- ▶ Idée 1 : Concaténer puis trier.
 - ▶ Très mauvaise idée. Pourquoi ?
- ▶ Mieux :
 - ▶ Construire une liste vide \mathcal{L}
 - ▶ Tant que les deux listes \mathcal{L}_1 et \mathcal{L}_2 ne sont pas vides :
 - ▶ Prendre l'élément de tête le plus petit
 - ▶ L'enlever de la sous-liste et l'ajouter à \mathcal{L}

Complexité ?

LES DELEGUES

- ▶ 2 délégués
- ▶ Plus 1 délégué par groupe de TP

AVANT VENDREDI PROCHAIN