

# Chapitre 7 : méthodes, fonctions et procédures

Fonctions en informatique  
Déclaration, application de fonction  
Fonction récursive  
Déclaration, application de procédure  
Passage de paramètres  
Procédures récursives

# Fonctions

# Fonctions en mathématiques

- Une fonction  $f$  définie sur  $A$  (domaine de définition, espace de départ) à valeurs dans  $B$  (codomaine, espace d'arrivée, espace image) est une correspondance qui, à tout élément  $x$  de  $A$  fait correspondre **un élément et un seul**, noté  $f(x)$ , de  $B$ .
- $f(x)$  est le résultat de l'application de  $f$  à l'élément  $x$ .

# Spécification

- Une spécification est une description dans un langage formel (rigoureux) :
  - d'une correspondance entre des données et des résultats ainsi que des propriétés de cette correspondance.
- Un **algorithme** est un cas particulier de spécification.
- C'est une spécification exécutable.

# Les fonctions en informatique

$$f(x)=a*x^2+b*x+c$$

$x$  est le **paramètre formel** de la fonction  $f$

$a$ ,  $b$ ,  $c$  sont les variables globales (libres) de la fonction  $f$ .

L'expression (ici  $ax^2+bx+c$ ) définissant la fonction est appelée **corps de la fonction**.

Dans l'application de  $f$  à une valeur  $v$ , notée  $f(v)$ ,  $v$  est appelé **paramètre effectif** ou argument.

# Fonctions totales

En informatique, une fonction doit être **totale** (partout définie) => tout élément du domaine peut être a priori choisi comme argument.

Les cas particuliers doivent impérativement être traités de manière explicite.

# Déclaration d'une fonction :

## Syntaxe

`<déclaration_fonction> ::= <en-tete> <bloc>`

`<en-tete> ::=`

`<ident_type> <ident_fonction> <liste_paramètres>`

`<liste_paramètres> ::= () | (<déclar_paramètres>)`

`<déclar_paramètres> ::=`

`<type_param> <ident_param> { , <déclar_paramètres> }`

`<bloc> ::= { <suite_ordres> }`

`<suite_ordres> ::= <ordre> | <ordre> { ; <ordre> }`

`<ordre> ::= <instruction> | <déclaration>`

# Exemples de déclaration

`doubler` : identificateur de la fonction

`X` : paramètre formel

`int` : type du paramètre formel

`int` : type de la valeur retournée (résultat)

`int doubler(int X)` : signature de la fonction

`int → int` : type de la fonction

```
int doubler(int X){  
    return X*2;  
}
```

`max` : identificateur de la fonction

`A, B` : paramètres formels

`int` : type des paramètres formels

`int` : type de la valeur retournée (résultat)

`int max(int A, B)` : signature de la fonction

`int*int → int` : type de la fonction

```
int max(int A,B){  
    if(A>B) return A;  
    else return B;  
}
```



# Déclaration d'une fonction : sémantique (1/2)

La fonction dénotée par son identificateur appartient à l'environnement du programme.

A son identificateur est associé son **type**.

Le programmeur déclare ainsi les contraintes d'utilisation de l'objet fonction. Il spécifie un nouvel outil ajouté à l'environnement.

Le compilateur utilisera ces informations pour vérifier que l'utilisation qui est faite de cet objet fonction répond bien à l'intention du programmeur

# Déclaration d'une fonction : sémantique (2/2)

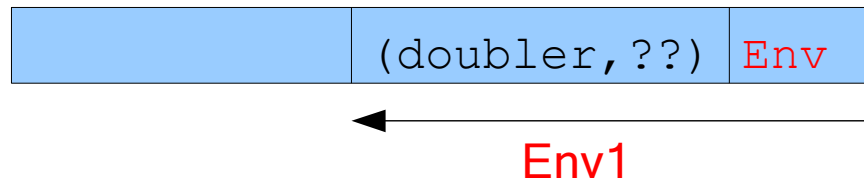
- La **valeur** de la fonction est déterminée et associée à son identificateur.
- Cette valeur est le code binaire de la fonction associé à son environnement de déclaration.
- Elle est appelée **fermeture** car elle réunit en une seule structure son code binaire et le contexte de déclaration de son corps.

# Sémantique de déclaration : exemple (1/4)

Déclaration dans l'environnement **Env** de la fonction :

```
int doubler(int X) { return X*2; }
```

- création de la liaison (doubler, ??)
- extension de **Env** avec cette liaison



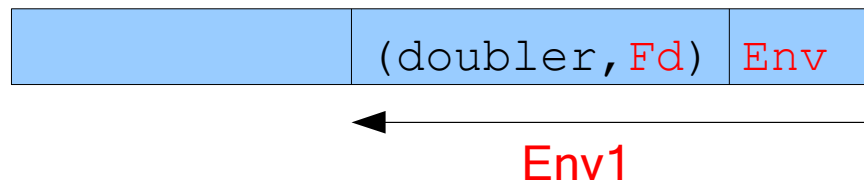
- détermination du type de la fonction : `int` → `int`

# Sémantique de déclaration : exemple (2/4)

Détermination de la fermeture  $F$  de `doubler` :

$Fd = \ll X \rightarrow \text{corps de doubler}, Env1 \gg$

Modification de la liaison dans **Env1**

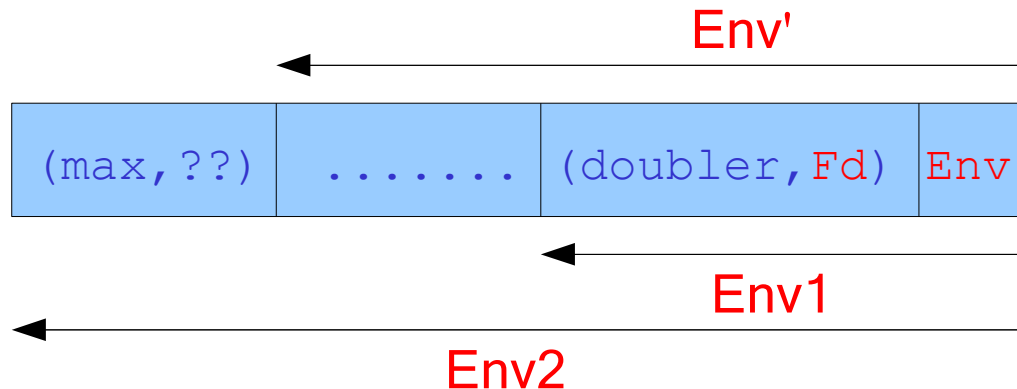


# Sémantique de déclaration : exemple (3/4)

Déclaration dans l'environnement **Env'** de la fonction:

```
int max(int A,B) { ..... }
```

- création de la liaison `(max, ??)`
- extension de **Env'** avec cette liaison



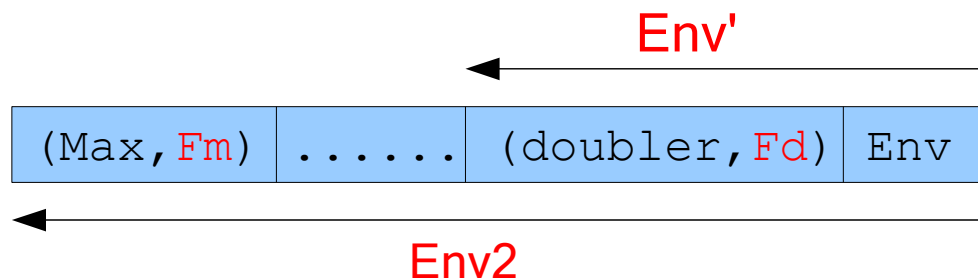
- Détermination du type de la fonction : `int*int → int`

# Sémantique de déclaration : exemple (4/4)

Détermination de la fermeture  $F$  de `max` :

$F_m = \langle\langle A, B \rightarrow \text{corps de max}, \text{Env2} \rangle\rangle$

Modification de la liaison dans **Env2**

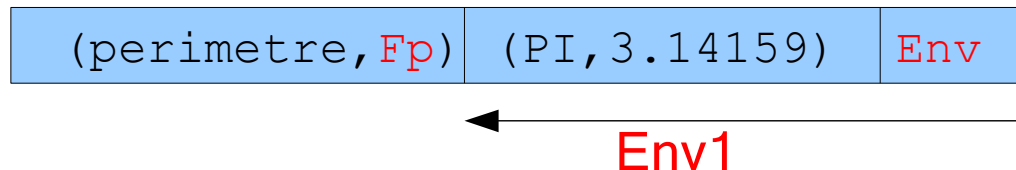


# Autre exemple

```
float perimetre(float R) {return 2.0f*PI*R; }
```

La fonction d'identificateur `perimetre` est introduite dans l'environnement `Env1` du programme avec son type :

`float-->float`



L'environnement contenu dans `Fp` contient `PI` et permet donc à l'expression `2.0f*PI*R` d'être évaluée avec succès

# Fonction produit de matrice (1/3)

Multiplication d'une matrice  $a$  par une matrice  $b$ .

Le nombre de colonnes de  $a$  doit être le même que le nombre de lignes de  $b$ .

Le type des éléments de  $a$  doit être compatible avec celui des éléments de  $b$ .

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \\ b_{41} & b_{42} & b_{43} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \\ c_{41} & c_{42} & c_{43} \end{pmatrix}$$

où

$$c_{ij} = a_{i1} * b_{1j} + a_{i2} * b_{2j} + a_{i3} * b_{3j} + a_{i4} * b_{4j}$$



# Fonction produit de matrice(2/3)

```
static float[][] produitMatrices(float[][] A, float[][] B) {
    float[][] C = new float[A.length][B[0].length];
    for(int i=0; i< A.length; i++){
        for(int j=0; j< B[0].length; j++){
            float s=0.0f;
            for(int k=0; k<A.length; k++)
                s=s+A[i][k]*B[k][j];
            C[i][j]=s;
        }
    }
    return C;
}
```

# Fonction produit de matrice(3/3)

```
public static void main( String[] args ){
    float[][] x={{3f,4f,4f,6f},
                {5f,6f,7f,8f},
                {8f,7f,3f,1f},
                {4f,8f,1f,9f}};
    float[][] y={{7f,4f,6f},
                {2f,6f,8f},
                {8f,2f,1f},
                {4f,3f,9f}};
    float[][] z = produitMatrices(x,y);
    for(int i=0;i<z.length;i++){
        for (int j=0;j<z[0].length;j++)
            System.out.print(z[i][j]+" ");
        System.out.println();
    }
}
```

## Résultat affiché

85.0	62.0	108.0
135.0	94.0	157.0
98.0	83.0	116.0
88.0	93.0	170.0

# Application d'une fonction : syntaxe

```
<application_fonction> ::=  
    <ident_fonction><liste_paramètres_effectifs>  
<liste_paramètres_effectifs> ::= () | (<param_effectifs>)  
<param_effectifs> ::= <val_param>{ , <param_effectifs> }  
<val_param> ::= ident | une expression du type associé
```

## Exemples

```
Math.max(78, 45)  
doubler(X*5)  
perimetre(37.97)
```

# La fonction `eliminer`

Il s'agit d'éliminer toutes les occurrences d'un caractère dans une chaîne de caractères

Soit `c` le caractère et `t[0..n]` le tableau représentant la chaîne

## Raisonnement par récurrence

1- hypothèse de récurrence

$$\forall k \in [0..j-1], t[k] \neq c$$

2- condition d'arrêt

$$j > n$$

3- corps de boucle

si `t[j]=c`

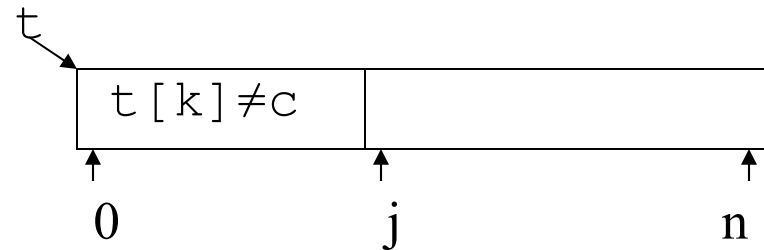
alors `t=t[0..j-1]+t[j+1..n];`

`n←n-1; j←j+1;`

sinon `j←j+1;`

4- initialisation : `n←dernier indice`

$$j=0$$



# Déclaration de la fonction :

## `eliminer`

```
static String eliminer(char c,String t){
    int j = 0;
    int n = t.length()-1;
    while( j<=n ){
        if( t.charAt(j)==c ){
            String t1 = t.substring(0,j);
            String t2 = t.substring(j+1,n+1);
            t=t1.concat(t2);
            n--;
        }
        j++;
    }
    return t;
}
```

```
"abc".substring(2,3); → c
"abc".substring(0,0); → 
"abc".substring(0,3); → abc
```

# Programme Eliminer

```
public class Eliminer {
    public static void main(String[] args) {
        String texte=
            "hier c'est le passe;demain c'est le" +
            "futur;aujourd'hui, c'est un cadeau.C'est pour" +
            " cela qu'on l'appelle present";
        System.out.println(eliminer(' ',texte));
        System.out.println(eliminer('e',texte));
    }
}
```

# Résultat

hier c'est le passé; demain c'est le futur; aujourd'hui  
c'est un cadeau. C'est pour ça qu'on l'appelle présent

hier c'est le passé; demain c'est le futur; aujourd'hui  
c'est un cadeau. C'est pour ça qu'on l'appelle présent

# Application d'une fonction : sémantique

```
T' h ( P ) {  
  // corps de h  
}
```

Evaluons  $h(exp)$ ,

L'évaluation de  $h$  permet de récupérer le code binaire qui lui est associé et donc l'environnement dans lequel son corps avait été déclaré.

L'évaluation de  $exp$  permet d'attribuer une valeur à l'argument.

A partir de ces informations, le **contexte d'exécution de l'appel** est construit. La valeur de  $exp$  est associée à l'identificateur du paramètre formel et le code binaire exécuté.

A la fin de l'exécution, ce contexte est détruit et remplacé par le contexte d'avant appel.



# Transmission de paramètre

- Lors de l'appel d'une fonction, le paramètre effectif est transmis au paramètre formel.
- Tout se passe comme si une **copie de la valeur** du paramètre effectif initialisait le paramètre formel
- On parle de **passage par valeur**
- Conséquence : l'exécution de la fonction ne peut pas modifier le paramètre effectif
- Le paramètre formel  $x$  est une **variable** pendant toute l'exécution de la procédure
- Note : la valeur transmise est une **référence** si le paramètre est un **tableau** ou un **objet**

# Application de la fonction double (1/2)

```
int doubler(int X){  
    X = X*2;  
    return X;  
}
```

Evaluons

```
int a = 18;int b = doubler(a);
```

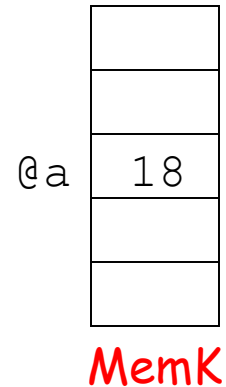
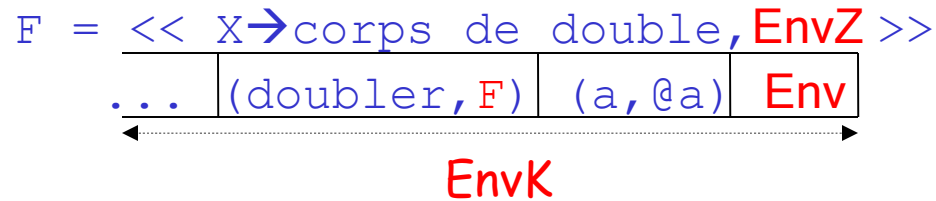
L'évaluation de `doubler` ramène le code binaire de la fonction et donc son **environnement de déclaration**

Le paramètre formel `X` prend la valeur de `a` (soit `18`). Cette association étend l'environnement de déclaration pour former l'**environnement d'exécution** de la fonction.

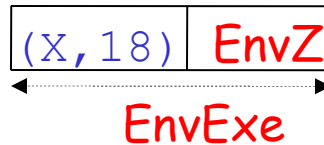
Le code s'exécute alors, la valeur retournée est `36` et le contexte d'exécution détruit.

On revient donc au contexte d'exécution initial dans lequel `a` vaut toujours `18`

# Application de la fonction double (2/2)



- ✓ Evaluation de `double(a)` dans  $(\text{EnvK}, \text{MemK})$ 
  - l'évaluation de l'identificateur `doubler` retourne la fermeture
 
$$F = \ll X \rightarrow \text{corps de doubler}, \text{EnvZ} \gg$$
  - 18 s'évalue en lui-même
- ✓ Construction de l'environnement d'exécution  $\text{EnvExe}$  de la fonction



- ✓ Exécution du corps de `doubler` dans l'état  $(\text{EnvExe}, \text{MemK})$  : la valeur retournée est 36
- ✓ Destruction de  $\text{EnvExe}$  et restitution de l'environnement initial  $\text{EnvK}$

# Fonctions récursives

- Une fonction récursive est une fonction qui fait appel à elle-même pour son exécution.
- Toute fonction java appartient à l'environnement de sa fermeture.
- Autrement dit, toute fonction Java se connaît elle-même. Elle est donc en mesure de s'appeler elle-même.

# Exécution d'une fonction récursive (1/2)

Soit la fonction `sommeCarres(n)` qui calcule la somme des carrés des `n` premiers entiers positifs ou nuls.

Elle est définie récursivement par :

```
sommeCarres(n) = si n=0 alors 0  
                sinon SommeCarres(n-1)+n*n
```

Traduction Java

```
int sommeCarres(int n) {  
    if (n==0) return 0;  
    else return sommeCarres(n-1)+n*n;  
}
```

# Exécution de `sommeCarres` (3)

étapes du calcul	environnement d'exécution	pile des contextes
<code>sommeCarres(3)</code>	<code>n=3</code>	<code>[]</code>
<code>sommeCarres(n-1)+n*n</code>	<code>n=3</code>	<code>[]</code>
<code>sommeCarres(2)</code>	<code>n=2</code>	<code>[n=3]</code>
<code>sommeCarres(n-1)+n*n</code>	<code>n=2</code>	<code>[n=3]</code>
<code>sommeCarres(1)</code>	<code>n=1</code>	<code>[n=2, n=3]</code>
<code>sommeCarres(n-1)+n*n</code>	<code>n=1</code>	<code>[n=2, n=3]</code>
<code>sommeCarres(0)</code>	<code>n=0</code>	<code>[n=1, n=2, n=3]</code>
<code>0</code>	<code>n=0</code>	<code>[n=1, n=2, n=3]</code>
<code>0+n*n--&gt;1</code>	<code>n=1</code>	<code>[n=2, n=3]</code>
<code>1+n*n--&gt;5</code>	<code>n=2</code>	<code>[n=3]</code>
<code>5+n*n--&gt;14</code>	<code>n=3</code>	<code>[]</code>

# Conception d'une fonction réursive

Éliminer toutes les occurrences d'un caractère donné dans une chaîne.

Raisonnement par récurrence sur la longueur de la donnée. Soit  $t$  le texte et  $c$  le caractère à éliminer :

On connaît le résultat pour une valeur de la longueur :

**si**  $\text{longueur}(t)=0$ , le résultat est  $t$  (ou "")

On suppose que l'on sait éliminer toutes les occurrences de  $c$  pour un texte  $t$  de longueur  $i-1$ .

On montre que l'on peut éliminer tous les caractères  $c$  de  $t$  de longueur  $i$ ;

# Algorithme

Soit une chaîne représentée par le tableau  $t$  de longueur  $i$

**si** longueur( $t$ )=0 **alors** le résultat est  $t$

**sinon**

**si** le premier caractère =  $c$

**alors**

le résultat provient de **l'élimination** de toutes les occurrences de  $c$  du texte  $t$  de longueur  $i-1$  (privé de son premier caractère)

**sinon**

le résultat est construit par la **concaténation** du 1er caractère ( $t.charAt(0)$ ) avec la sous-chaîne  $t$  de  $1$  à  $i-1$  dans laquelle toute occurrence de  $c$  a été supprimée



# Implantation de la fonction réursive

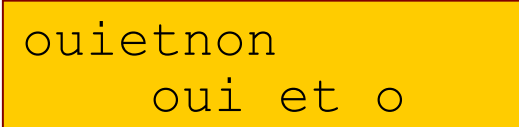
```
static String eliminer(char c,String t){
    int j=0;
    int i=t.length();
    if(i<=j+1) return "";
    else
        if(t.charAt(j)==c){
            t=t.substring(j+1,i);
            return eliminer(c,t);
        }else{
            char car=t.charAt(j);
            t=t.substring(j+1,i);
            return (String.valueOf(car)).concat(eliminer(c,t));
        }
    }
}
```



# Implantation du programme

```
public class EliminerRec{
    public static void main(String[] args) {
        String texte="      oui et non      ";
        System.out.println(eliminer(' ',texte));
        System.out.println(eliminer('\n',texte));
        System.out.println(texte);
    }

    static String eliminer(char c,String t){
        ...
        ...
    }
}
```



```
ouietnon
  oui et o
```

# Procédures

# Intérêt

- ❖ Créer une instruction nouvelle qui deviendra une primitive pour le programmeur
- ❖ Structurer le texte source du programme et améliorer sa lisibilité
- ❖ Factoriser l'écriture lorsque la suite d'actions nommée par la procédure intervient plusieurs fois

# Différence entre fonction et procédure (1/2)

Une **fonction** peut être vue comme un opérateur produisant une valeur.

Une fonction n'a pas pour rôle de modifier l'**état** courant du programme en exécution.

Une bonne programmation interdit aux fonctions de réaliser des effets de bords.

On appelle **effet de bord** toute modification de la mémoire (affectation d'une variable, opération de lecture en mémoire) ou toute modification d'un support externe (disque, écran, disquette, etc ).

Une fonction n'est pas une instruction.

Une fonction réalise une simple opération dont le résultat peut être, par la suite, utilisé par une instruction.

# Différence entre fonction et procédure (2/2)

Une **procédure** est une instruction composée qui peut prendre des paramètres et dont le rôle est de modifier l'état courant.

Les procédures ne retournent pas de résultat.

Les 2 aspects d'une procédure :

- Déclaration
- Appel

# Déclaration de procédure

- Le but de la **déclaration d'une procédure** est d'introduire dans l'environnement courant :
  - son identificateur
  - son type
  - sa fermeture

# Déclaration de procédure : syntaxe

```
<déclaration_proc> ::= <en-tete_proc><corps>;  
<en-tete_proc> ::= void <ident_proc><liste_paramètres>  
<liste_paramètres> ::= vide | (<déclar_paramètres>)  
<déclar_paramètres> ::=  
    <ident_param><type_param>{ , <déclar_paramètres> }  
<corps> ::=
```

où

```
<ident_param> ::= identificateurs des paramètres  
formels
```

```
<type_param> ::= type des paramètres formels
```



# Exemple : procédure

## permuter

```
void permuter(char[] a, char[] b) {  
    char tmp;  
    for(int i=0; i<a.length; i++) {  
        tmp = a[i];  
        a[i] = b[i];  
        b[i] = tmp;  
    }  
}
```

identificateur : `permuter`  
type de la procédure : `char[]*char[] → vide`  
fermeture :  $F = \langle\langle a, b \rangle \rightarrow \text{corps de permuter, env} \rangle\rangle$

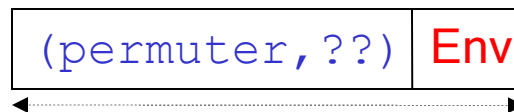
**Remarque** : l'instruction `return` devient inutile puisqu'aucune valeur n'est retournée par la procédure

# Déclaration de la procédure

## permuter (1/2)

Mécanisme d'évaluation de cette déclaration de procédure dans l'environnement **Env** :

- Création de la liaison : `(permuter, ??)`
- Extension de **Env** avec cette liaison : on obtient le nouvel environnement **Env1**



**Env1**

- Détermination du type des paramètres et du type du résultat. Le type de la fonction est :

`char []*char []-->vide`

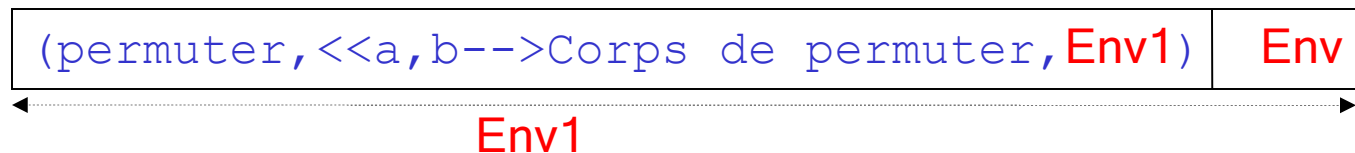
# Déclaration de la procédure

## permuter (2/2)

Détermination de la fermeture  $F$  de `permuter`, dans un environnement `Env1`, à partir du corps de la procédure

$$F = \langle\langle a, b \rightarrow \text{Corps de permuter}, \text{Env1} \rangle\rangle$$

Modification de la liaison de l'environnement `Env1`



# Appel de procédure

L'application d'une procédure à des paramètres effectifs constitue l'appel de cette procédure.

## Syntaxe

```
<appel_procedure> ::=  
    <id_proc> (<liste_params_effectifs>) | ();  
<liste_params_effectifs> ::= <param> { , <param> }  
<param> ::= <valeur> | <id_param> | <expression>
```

Le type des paramètres effectifs doit être le même que celui des paramètres formels correspondants.

# Passage de paramètre

Le contrôle de type a été effectué. Les paramètres formels et effectifs correspondants ont donc le même type.

Transmission du paramètre passé par **valeur** :

1. Etablissement d'un canal de communication entre le paramètre effectif et le paramètre formel
2. Copie de la valeur du paramètre effectif  $y$  et affectation de cette valeur au paramètre formel correspondant  $x$
3. Après retour vers l'appelant, ce mode de transmission garantit que le paramètre effectif conserve sa valeur d'appel.

Le paramètre formel  $x$  est une **variable** pendant toute l'exécution de la procédure

# Passage de paramètre par valeur: exemple (1/3)

```
class Permutation{
    public static void main(String[] args) {
        char[] y = {'a','b','c','d'};
        char[] z = {'w','x','y','z'};
        System.out.println("z="+z);
        System.out.println("z[0]="+z[0]);
        permuter(y,z);
        System.out.println("z="+z);
        System.out.println("z[0]="+z[0]);
    }
    static void permuter(char[] a,char[] b){
        char tmp;
        for(int i=0;i<a.length;i++){
            tmp = a[i];
            a[i] = b[i];
            b[i] = tmp;
        }
    }
}
```

valeur affichée:

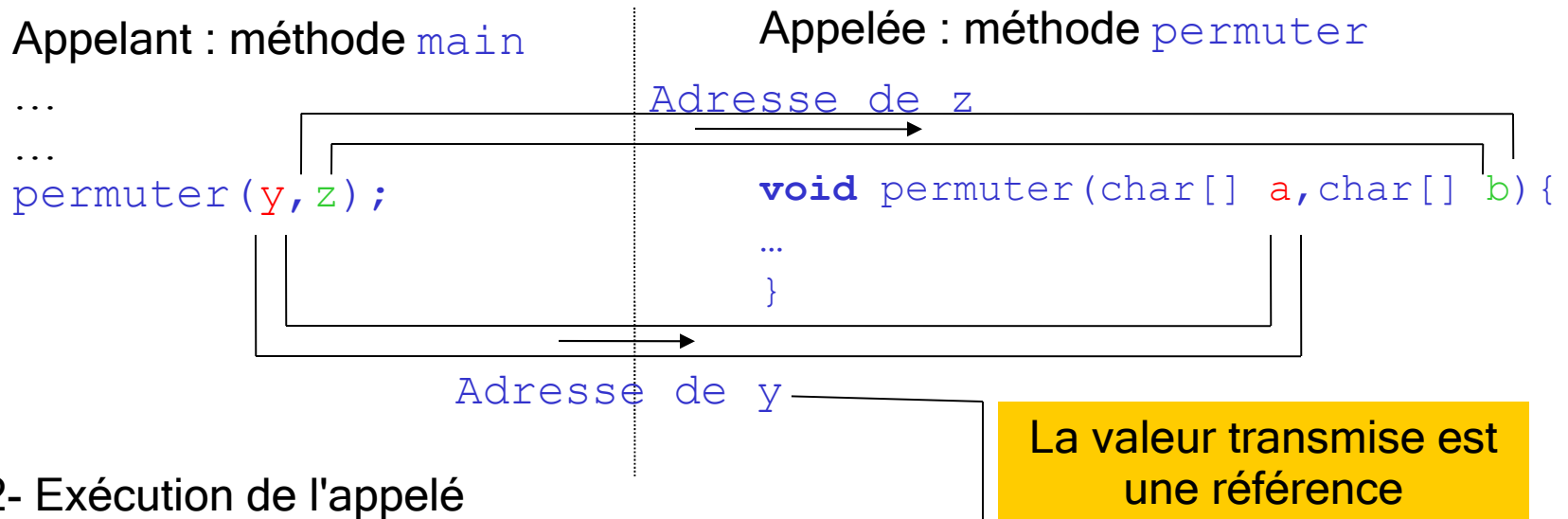
adresse de z=[C@179953c  
z[0]=a

valeur affichée:

adresse de z=[C@179953c  
z[0]=w

# Passage de paramètre par valeur : exemple (2/3)

## 1- Transmission des paramètres



## 2- Exécution de l'appelé

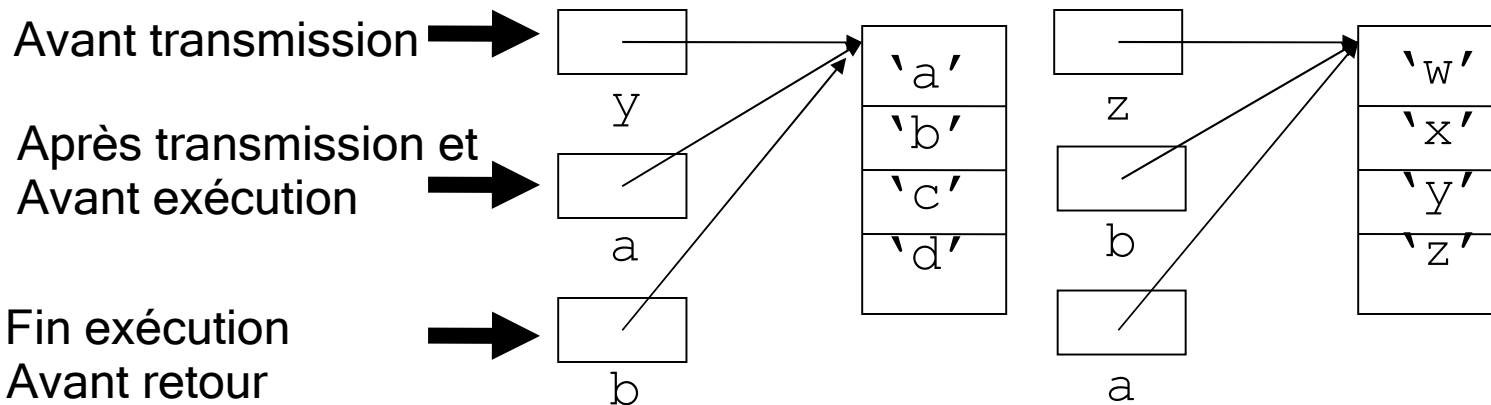
## 3- Retour vers l'appelant :

- le contrôle est passé à l'appelant derrière l'appel.
- aucune valeur n'est transmise en retour
- y et z conservent la même valeur qu'avant l'appel

# Passage de paramètre par valeur : exemple (3/3)

Les paramètres étant des tableaux, c'est leur référence et non leur contenu qui est copié dans le paramètre formel correspondant.

En effet, la valeur d'une variable de type tableau est une adresse (ou référence)



Après retour, seules subsistent les variables `y` et `z` qui n'ont pas changé de valeur



# Passage de paramètre par valeur d'un type primitif(1/3)

```
class Permutation{
    public static void main(String[] args){
        char y = 'A';
        char z = 'X';
        System.out.println("z="+z);
        permuter(y,z);
        System.out.println("z="+z);
    }
}
```

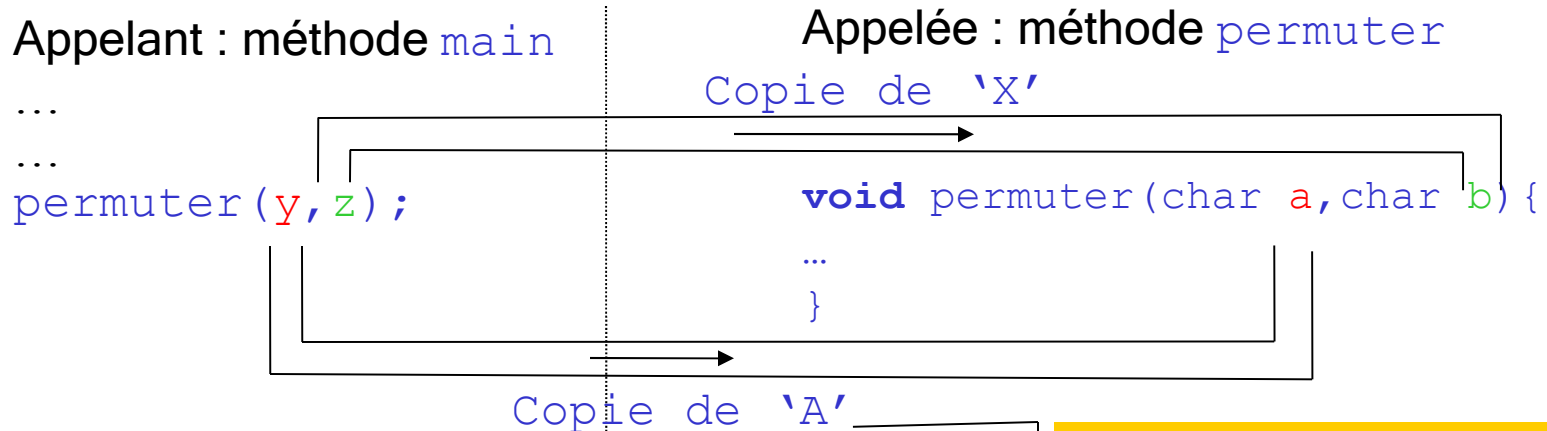
valeur affichée :  
z=X

valeur affichée :  
z=X

```
static void permuter(char a,char b){
    char tmp = a;
    a = b;
    b = tmp;
}
} (
```

# Passage de paramètre par valeur d'un type primitif (2/3)

## 1- Transmission des paramètres



La valeur transmise est une valeur primitive

## 2- Exécution de l'appelé

## 3- Retour vers l'appelant :

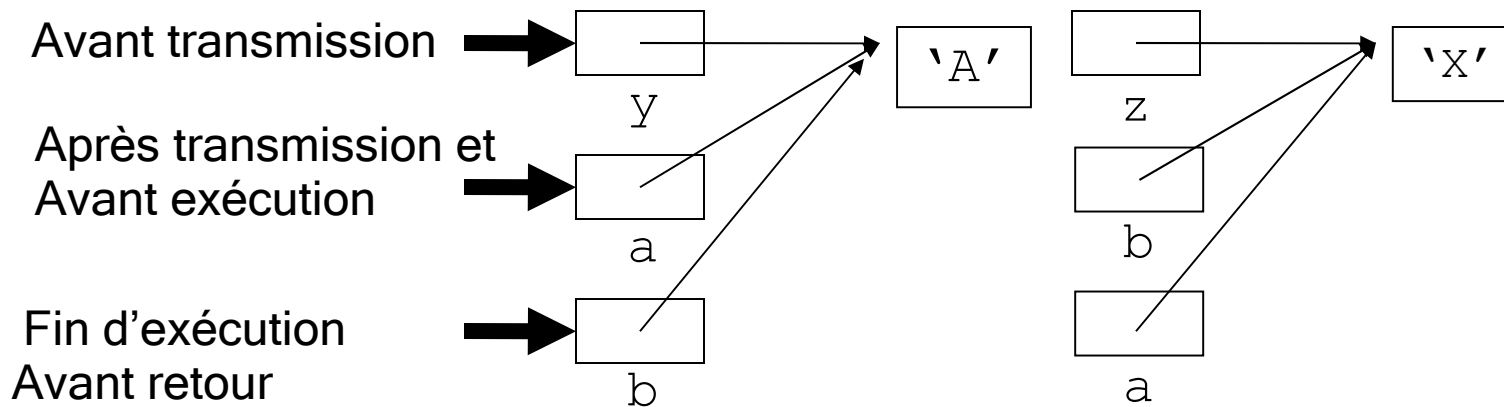
le contrôle est passé à l'appelant derrière l'appel.

aucune valeur n'est transmise en retour

y et z conservent la même valeur qu'avant l'appel

# Passage de paramètre par valeur d'un type primitif (3/3)

Dans cet exemple, les paramètres sont d'un type primitif, c'est leur contenu qui est copié dans le paramètre formel correspondant.



Après retour, z vaut toujours 'X'

# Profil des paramètres d'une procédure

Deux procédures ont le même profil de paramètres si :

- elles ont le même nombre de paramètres
- à chaque position, les paramètres ont le même type

```
void permuter(char a, char b) ;
```

```
void swap(char x, char y) ;
```

Les procédures `permuter` et `swap` ont le même profil de paramètres

# Surcharge de procédures

Un identificateur de procédure est surchargé si :

- il identifie plusieurs procédures
- si ces procédures n'ont pas le même **profil de paramètres**

Un tel identificateur possède plusieurs sémantiques

```
void permuter(char a,char b);
```

```
void permuter(int a,int y);
```

Le code de ces deux procédures sera différent puisque leur type est différent.

Ces deux déclarations sont dites **surchargées**.

# Exemple: calcul du prix TTC

Cet exemple invoque à la fois une fonction `PTTC` et une procédure `println`

```
public class Prix{
    static final double TVA=19.6;
    public static void main(String[] args){
        System.out.println("prix TTC="+PTTC(10.0));
    }

    static double PTTC(double X){
        return X+0.01*TVA*X;
    }
}
```

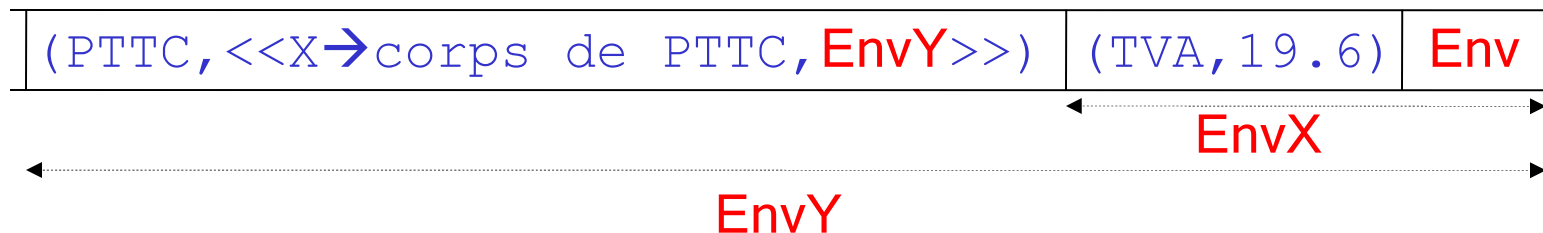
# Evaluation des déclarations

L'évaluation de la déclaration de la constante `TVA` associe l'identificateur `TVA` au type `double` et à la valeur `19.6`

L'évaluation de la déclaration de la fonction `PTTC`

associe l'identificateur `PTTC` au type `double`→`double` et à une valeur indéfinie.

L'évaluation du corps de la fonction crée sa fermeture qui remplace la valeur indéfinie



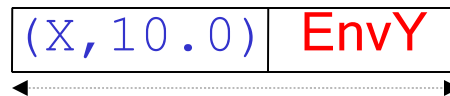
```
System.out.println(
    PTTC(10.0))(1/2)
```

Evaluation de `PTTC` et `10.0`

`PTTC`  $\Rightarrow$   $\langle\langle X \rightarrow$  corps de `PTTC`, `EnvY` $\rangle\rangle$

`10.0`  $\Rightarrow$  `10.0`

Construction de l'environnement d'exécution de `PTTC(10.0)`



`EnvExe`

Exécution du corps de `PTTC` dans l'état  $(\text{EnvExe}, \text{Mem})$

$\Rightarrow X + 0.01 * \text{TVA} * X = 10.0 + 0.01 * 19.6 * 10.0 = 11.96$

Destruction de `EnvExe`, restitution de `EnvY`



```
System.out.println(
    PTTC(10.0))(2/2)
```

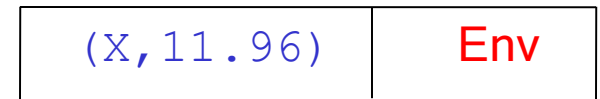
Evaluation de `System.out.println(PTTC(10.0))` (dans `Env`)

`println` =>  $\langle\langle X \rightarrow \text{corps de println}, \text{Env} \rangle\rangle$

`PTTC(10.0)` => 11.96

Construction de l'environnement d'exécution de

`System.out.println(PTTC(10.0))`



← EnvPrintlnExe →

Exécution du corps de `println` dans l'état (`EnvPrintlnExe`, `Mem`)

=> affichage de la valeur 11.96 à l'écran

Destruction de `EnvPrintlnExe`, restitution de `EnvY`

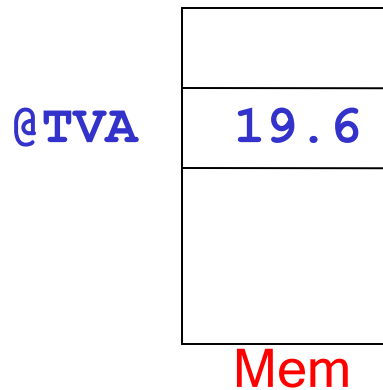
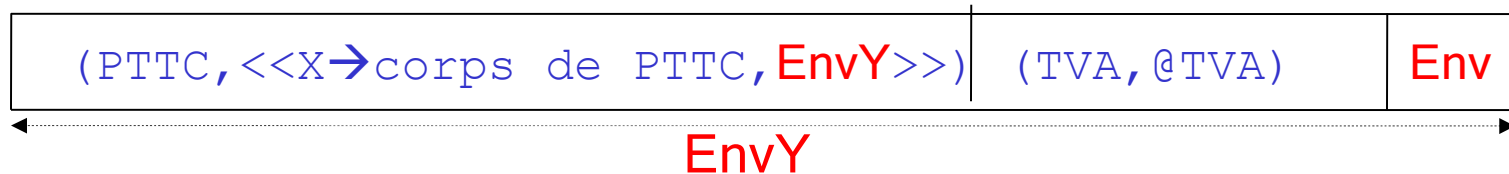
# Exemple : calcul du PTTC variante (1/6)

```
public class Prix{
    static double TVA=19.6;
    public static void main(String[] args){
        System.out.println("prix TTC="+PTTC(10.0));
    }

    static double PTTC(double X){
        return X+0.01*TVA*X;
    }
}
```

# Exemple : variante (2/6)

Après évaluation des déclarations, nous obtenons l'état suivant :



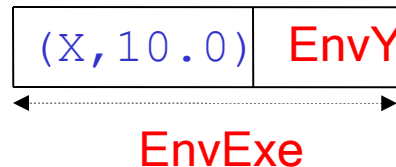
# Exemple : variante (3/6)

Evaluation de `PTTC` et `10.0`

`PTTC` =>  $\langle\langle X \rightarrow \text{corps de PTTC}, \text{EnvY} \rangle\rangle$

`10.0` => `10.0`

Construction de l'environnement d'exécution de `PTTC (10.0)`



Exécution du corps de `PTTC` dans l'état  $(\text{EnvExe}, \text{Mem})$

$\Rightarrow X + 0.01 * \text{TVA} * X = 10.0 + 0.01 * 19.6 * 10.0 = 11.96$

Destruction de `EnvExe`, restitution de `EnvY`

# Exemple : variante (4/6)

```
public class Prix{
    static double TVA=19.6;
    public static void main(String[] args){
        System.out.println("PTTC(10.0)="+PTTC(10.0));
        TVA=28.0;
        System.out.println("PTTC(10.0)="+PTTC(10.0));
    }

    static double PTTC(double X){
        return X+0.01*TVA*X;
    }
}
```

## résultat

PTTC(10.0)=11.96

PTTC(10.0)=12.80

# Exemple : variante (5/6)

On remarque que cette façon de coder ne préserve pas la notion de fonction car, appliquée à un même paramètre effectif, `PTTC` retourne 2 résultats différents.

Nous avons transmis de l'information à la fonction par **effet de bord** et non par paramètre.

Comment éviter cela ?

Par l'introduction d'un paramètre supplémentaire qui permettra de transmettre la nouvelle valeur de la TVA de manière explicite

# Exemple : variante (6/6)

```
public class Prix{
    static double TVA=19.6;
    public static void main(String[] args){
        System.out.println("PTTC(10.0)="+PTTC(10.0,TVA));
        TVA=28.0;
        System.out.println("PTTC(10.0)="+PTTC(10.0,TVA));
    }

    static double PTTC(double X,double tva){
        return X+0.01*tva*X;
    }
}
```

## résultat

PTTC(10.0,19.6)=11.96

PTTC(10.0,28.0)=12.80

# Procédures récursives

Comme les fonctions, les procédures sont naturellement récursives.

Exemple : **Les tours de Hanoï**

3 pieux **A,B,C** sont plantés en terre. Initialement, sur le pieu **A** sont empilés des disques de taille décroissante. Les pieux **B** et **C** sont alors vides.

A l'état final tous les disques sont empilés sur le pieu **C**.

Les règles sont les suivantes :

on ne peut déplacer qu'un disque à la fois et bien sûr, il ne peut être placé qu'en haut d'une pile (au sommet d'un pieu).

il est possible d'utiliser le pieu **B**

on peut toujours empiler un disque sur un autre à la condition que sa taille soit inférieure



# La procédure `hanoi` (1/4)

- La procédure `hanoi` construit et affiche la suite des déplacements de disques de manière récursive.
- Les 3 pieux sont représentés par les caractères 'A', 'B', 'C'.
- Les disques sont représentés par un entier qui simule leur taille.

# La procédure `hanoi`(2/4)

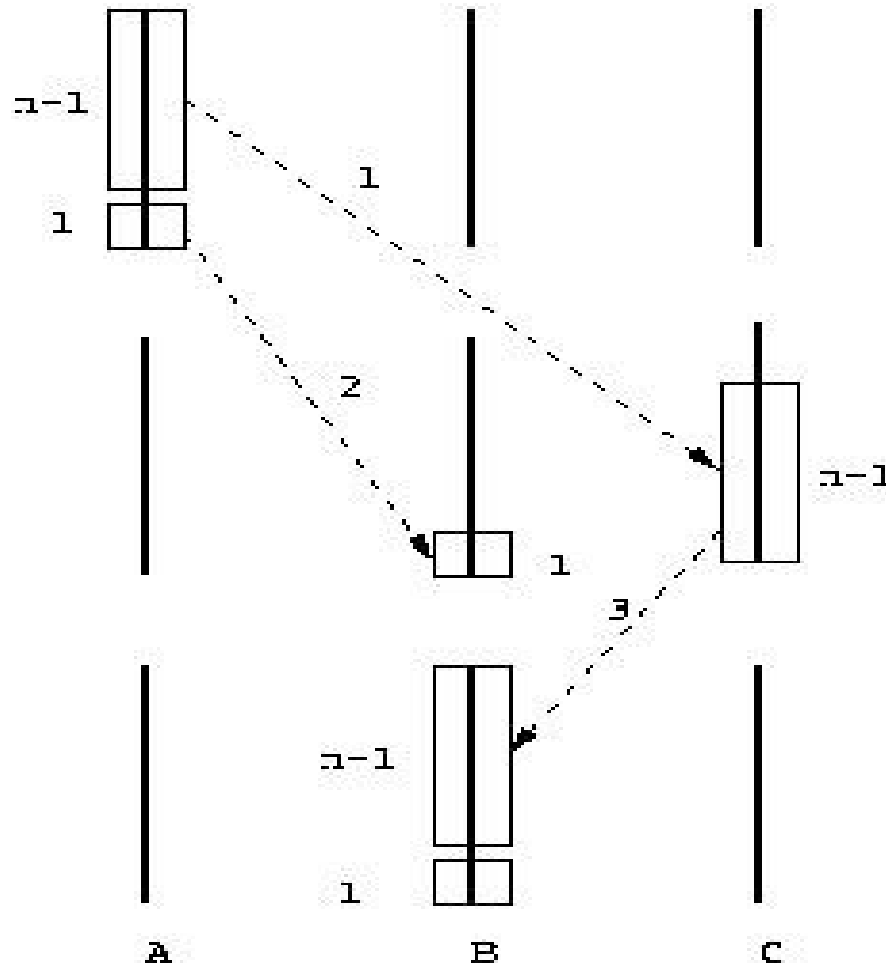
L'idée est, ici, de dire :

Si aucun disque n'est empilé sur `A`, alors il n'y a rien à faire

Supposons que je sache déplacer les  $n-1$  disques les plus petits de `A` vers `C` en utilisant `B`

Alors il suffit de placer le plus gros disque restant de `A` sur le pieu libre `B` et de recommencer (ce que j'ai supposé savoir faire), c'est à dire déplacer les  $n-1$  disques (cette fois de `C` vers `B` en utilisant `A`).

# La procédure hanoi (3/4)



# La procédure `hanoi` (4/4)

```
public class Hanoi{
    public static void main(String[] args){
        hanoi(3, 'A', 'B', 'C');
    }
    static void hanoi(int n, char X, char Y, char Z){
        if (n!=0){
            // n-1 disques de X vers Z via Y
            hanoi(n-1, X, Z, Y);
            // 1 disque de X vers Y
            System.out.print("disque "+n+" : ");
            System.out.println(X+" --> "+Y);
            // n-1 disques de Z vers Y via X
            hanoi(n-1, Z, Y, X);
        }
    }
}
```

```
disque 1 : A --> B
disque 2 : A --> C
disque 1 : B --> C
disque 3 : A --> B
disque 1 : C --> A
disque 2 : C --> B
disque 1 : A --> B
```

# La procédure `hanoi`: résultats

Le résultat pour un pieu `A` où sont empilés 3 disques est :

```
disque : 1 : A --> C
disque : 2 : A --> B
disque : 1 : C --> B
disque : 3 : A --> C
disque : 1 : B --> A
disque : 2 : B --> C
disque : 1 : A --> C
```