Bases de la programmation

Notions fondamentales (Types, variables, affectations)

Présentation

- Enseignant :

  M. Abdessamad IMINE

  Bureau : 124

  email : Abdessamad.Imine@univ-lorraine.fr

- Progression du module :

  Cours, TD et TP (disponibles sur **Arche**)

  Lié au module Algo

  Un soutien est prévu pour les étudiants

- Examens :

  Un partiel (novembre)

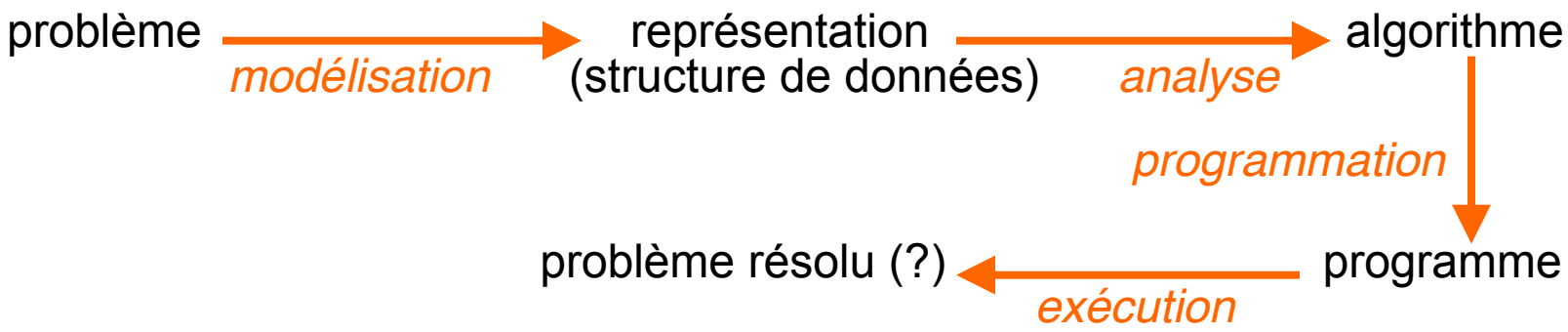
  Des tests (hebdomadaire)

  Un projet à rendre

  Note finale = (60*DS + 25*Test + 15*TP) / 100

Introduction

- But : apprendre les bases de la **programmation** avec **Java**.
- Très lié à l'algorithmique et aux structures de données :



- Ex : contrôler un ascenseur

  Modélisation → n étage courant, liste des étages appelés…

  Algorithme (suite des étapes de résolution) →

  attendre un appel de l'ascenseur
  empêcher tout autre appel
  aller à l'étage correspondant
  supprimer l'appel

  Programme → algorithme exprimé en langage de programmation

- Java permet d'enseigner une méthode de programmation avec des principes forts (≠ « bidouilles »).
- Un programme doit être compréhensible, réutilisable, modifiable.
- Java le favorise mais ne le garantit pas. Efforts requis :
  - réfléchir à l'**organisation** du programme avant de programmer;
  - expliquer ce que fait le programme (**commenter**);
  - **bien présenter** le programme pour que sa structure apparaisse clairement;
  - être critique vis-à-vis des résultats : l'ordinateur aide à corriger un programme qui ne s'exécute pas; mais quand il l'exécute rien ne dit qu'il aboutit à la solution recherchée (**tester** différents cas);
  - **ne pas corriger les erreurs au jugé** sans avoir identifié les causes exactes.

Valeur

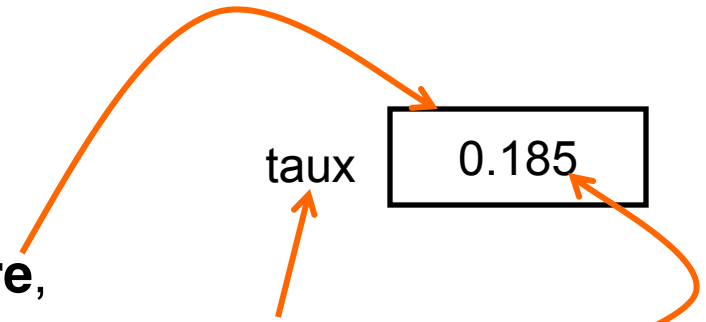
- Information réduite au maximum (non représentable sous une forme plus concise).

  Ex: 2 "1+3" (mais pas 1+3) 4.75

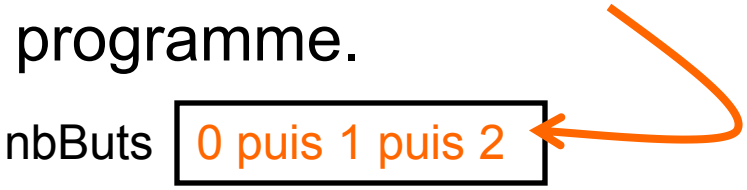
- Chaque valeur est codée/représentée dans la mémoire de l'ordinateur.
- Les valeurs sont les éléments concrets utilisés par la machine.

Variable

- Une variable désigne
  - un **emplacement mémoire**,
  - muni d'un **identificateur** (nom de la variable),
  - auquel une **valeur** a été affectée (contenu codé de l'emplacement mémoire).



- Ne pas confondre variable et valeur. La valeur change pendant l'exécution du programme.



Type

- Ensemble de valeurs.
- Détermine la nature de ces valeurs (leur 'sémantique').
- Indique la méthode de **codage** (implantation) et les **opérations** qui sont applicables aux valeurs.

  Ex : ***int*** est un des types Java pour les valeurs entières; définit comment ces valeurs sont codées et comment on les additionne/multiplie. Idem pour ***double*** pour les valeurs réelles mais le codage et les opérations sont différentes.

Types fondamentaux ou types de base

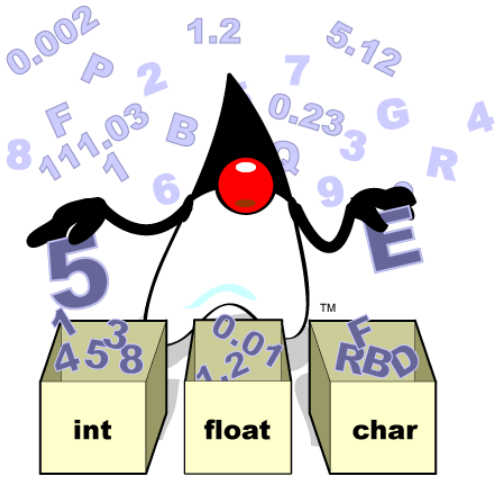
- Regroupe des valeurs simples directement codées en mémoire comme les valeurs numériques entières ou réelles.
- Principaux type fondamentaux en Java :

type	nature	nb octets
int	entier entre -2^{31} et $2^{31}-1$	4
double	nombre réel en double précision	8
boolean	vrai ou faux	1
char	un caractère (lettre, chiffre, ponctuation ...)	2

Valeurs littérales

- Valeurs appartenant aux types fondamentaux.

type	valeurs
int (entier)	2 465
double (réel)	2.0 4.3 3.4e-1
boolean (booléen)	*true* *false* (2 seules valeurs)
char (caractère)	'a' 'b' ... 'z' 'A' ... 'Z' '0' ... '9' ',' '(' ...



« Duke » la mascotte de Java

Déclaration de variable

- Un programme ne peut manipuler des valeurs que via des variables qui les contiennent.
- On doit **déclarer** les variables afin d'indiquer leur type.
- **Déclaration simple** :

  ***type identificateur*;**

  Ex : *int var1;* *// déclare une variable var1 entière*
  double valeur1; *// déclare une variable valeur1 réelle*
- **Déclaration multiple** (N variables avec le même type) :

  ***type identif1, identif2, …, identifN*;**

  Ex : *int num1, num2, val_entiere;*
  double valeur, aux;
- Les déclarations peuvent être placées à n'importe quel endroit **précédant** l'utilisation de la variable.

Identificateur

- Suite de caractères parmi :
  - lettres (minuscules et majuscules non confondues),
  - chiffres,
  - symbole soulignement '_' ('underscore').
- Ne peut pas débuter par un chiffre.
- Convention classique : minuscules + soulignements pour séparer les mots.

  Ex : *int indice_de_boucle;*

  boolean condition_remplie;
- Conseil : choisir des identificateurs qui aident à comprendre le programme.

  Ex : une variable qui stocke une moyenne de notes doit être appelée *moyenne* ou *moy_notes* plutôt que *x* ou *truc* !

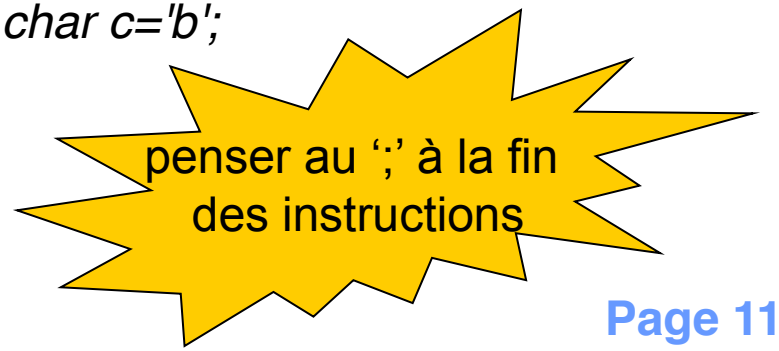
Affectation

- Modifie **le contenu** d'une variable.

  Ex: *var1=37;* affecte la valeur 37 à la variable var1.
  var1 étant de type entier on ne doit pas affecter une valeur réelle (3.5) ou booléenne (*true*)
- Même si on dit «var1 égale 37» ce n'est pas une comparaison ! L'affectation met 37 dans var1.
- Syntaxe : ***identificateur=valeur littérale du type;***
  ou ***type identificateur=valeur littérale type;***
  (déclaration et affectation simultanée).

Ex :
```
int num1,num2,val_entiere;
val_entiere=-65;
double valeur,aux;
num2=5; valeur=2.7;
num1=8762;
boolean valb;
num2=2; aux=-3.98e-14;
valb=false;
```

```
int num1,num2=5,val_entiere=-65;
num1=8762;
num2=2;
char c='b';
```



Affectation du contenu d'une variable à une autre variable

- **Recopie le contenu** d'une variable dans une autre.

  Syntaxe : ***identificateur2=identificateur1*;**

  Ex : *int num1,num2=5,val_entiere;*
  num1=8762;
  val_entiere=num2;
  num2=num1;

- La variable dont le contenu est recopié doit avoir été initialisée (contenir une valeur).
- On peut affecter une valeur de type T à une variable dont le type déclaré « contient » T.

  Ex: l'ensemble des réels « contient » l'ensemble des entiers

```
double somme; int val=9;
somme=6;          // int vers double : ok
somme=val;        // int vers double : ok
val=somme;        /* double vers int : interdit */
```

Note : 2 types de commentaires

Saisie au clavier de la valeur d'une variable

- Les mécanismes **d'entrées-sorties** permettent au programme de communiquer avec l'extérieur. Ils sont complexes et seront détaillés plus tard.
- **Saisie au clavier** :
  - importer la classe *Scanner* et créer une variable *Scanner* :

```
import java.util.Scanner;          // en début du fichier .java
Scanner sc = new Scanner(System.in);
```

  - saisir la valeur :

`int v;`	`double v;`	`String v;`
`v =sc.nextInt();`	`v=sc.nextDouble();`	`v=sc.next();`
`// entier`	`// réel`	`// chaîne sans espace`

Pour une chaîne avec espaces, écrire : *v=sc.nextLine();* mais *nextLine()* lit la fin de ligne contrairement aux autres méthodes ! Si un *nextInt()* précède un *nextLine()* ce dernier lit la fin de ligne qui suit l'entier. Il faut un 2ème *nextLine()* pour saisir une chaîne après.

Expressions

- Permettent de « fabriquer » des valeurs à partir d'autres valeurs : des **opérateurs** sont appliqués à des valeurs ou à des (sous-)expressions placées entre parenthèses.
- **Evaluées** pendant l'exécution. Peuvent alors être assimilées à leur valeur.
- **Expressions arithmétiques** (addition, multiplication de valeurs numériques) et **expressions logiques** (comparaisons dont la valeur est de type *boolean)*.
- Expressions arithmétiques ≈ formules mathématiques.

  Ex : *2*3*

  2/(12+1) division entière!!

  34%5 reste de la division entière (modulo)

  2.4-17e12 notation scientifique; e ou E pour puissance de 10

  4*(-6*(98+97))

- Si au moins un des opérandes est de type *double* alors la valeur de l'expression est de type *double* sinon elle est de type *int* (règle simplifiée).

```
2*3.7        // résultat double
5/2          // résultat entier = 2
5.0/2        // opération réelle, résultat réel = 2.5
```

- Expressions logiques : donnent vrai ou faux.

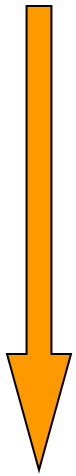
  Utilisent les opérateurs de comparaison et les opérateurs booléens :

<	Inférieur	*23<35*
<=	Inférieur ou égal	*5<=5*
>	Supérieur	*-14>45*
>=	Supérieur ou égal	*6>=-3*
==	Égal	*6==7*

!	négation	*(34<23)||(456 !=654)*
||	ou	
&&	et	*true && false*

Priorité des opérateurs

- Sans parenthèses : évaluation selon les priorités décroissantes suivantes :



1. opérateurs unaires : ! et -
2. opérateurs multiplicatifs : * / %
3. opérateurs additifs : + -
4. opérateurs de comparaison : < <= >= >
5. opérateurs d'égalité : == !=
6. et logique : &&
7. ou logique : ||

Avec des ( ) on évalue les expressions entre ( ) de la plus interne à la plus externe.

Ex :

*34.7+45.6*4* *// équivaut à 34.7+(45.6*4)*

true && 98>=87 || 31+8<56 *// (true && (98>=87)) || ((31+8)<56)*

- Opérateurs de même priorité : évalués de gauche à droite.

Expressions avec variables – affectation d'une expression

- Expressions avec des **variables** qui désignent la valeur qu'elles contiennent.

  Exemple : *(val1<45) && (val2/4==7)*
- Affectation de la valeur d'une expression à une variable **de même type** (ou « contenant » ce type) : Java évalue l'expression puis place la valeur dans la variable.

  Syntaxe : ***identificateur=expression*;**

  Ex : double moyenne;

  moyenne=(4.5+3.4)/2; // 3.95

  boolean verite;

  verite=(2==3)||((3>4)||(!(2.3==moyenne)));

  // false||(false||(!false)) c'est-à-dire true

  double valeur1=4.5;

  valeur2=3.4;

  moyenne=(valeur1+valeur2)/2;

Affichages à l'écran

- Affichage d'un **texte** à l'écran :

  System.out.println("bonjour"); *// affiche bonjour*

- Affichage d'une **valeur, variable, expression** :

  System.out.print(expression); ***// sans retour à la ligne***
  System.out.println(expression); *// avec retour à la ligne*

  Ex: *System.out.println(2.8/4); // affiche 0.7*
  double val1=3.4,val2=2.3;
  System.out.print(val1>val2); // affiche true sans retour à la ligne
  int entier=34,diviseur=5;
  System.out.println((entier%diviseur)==1); // affiche false

- Affichages complexes

  double val1=3.4,val2=2.3;
  System.out.print("comparaison : ");
  System.out.println(val1>val2); *// affiche* comparaison : true
  System.out.println("comparaison : "+(val1>val2));
  // identique car l'opérateur + permet de concaténer des textes
  // concaténer = mettre bout à bout

```
System.out.println(1+2+3+4+5+6+7);
/* affiche : 28, car l'opérateur + est celui des int */
System.out.println(""+1+2+3+4+5+6+7);
/* affiche : 1234567, car l'opérateur + est celui de concaténation
des chaînes de caractères, puisqu'on a un texte au début */
```

Bases de la programmation

Notions fondamentales (Blocs, conditionnelles)

Notion de bloc

- On peut regrouper des instructions par paquets : **blocs** identifiés par des **accolades**.
- Bloc assimilable à une instruction. Donc on peut placer des blocs dans des blocs (**blocs imbriqués)**.

Ex :

```
int var,aux,suivant;
aux=7;                          // instruction 1
{                               // début instruction2 (bloc)
    var=7;                      // instruction 2.1
    aux=var;                    // instruction 2.2
    {                           // début instruction 2.3 (bloc)
        suivant=0;              // instruction 2.3.1
        aux=9;                  // instruction 2.3.2
    }                           // fin instruction 2.3
    suivant=aux;                // instruction 2.4
}                               // fin instruction 2
{                               // début instruction3 (bloc)
    var=aux;                    // instruction 3.1
}                               // fin instruction 3
```

Notion de portée

- Une instruction ne peut utiliser une variable que si elle se situe dans la **portée** de la déclaration de la variable.
- Portée d'une déclaration = { instructions } qui se situent **après** la déclaration **dans le même bloc** qu'elle.

  Les instructions d'un bloc B2 imbriqué dans un bloc B1 appartiennent à B1.
- La **re-déclaration** de variable est **interdite** (même dans des blocs imbriqués).
- Déclaration de variable : pas nécessairement en début de bloc.

  Conseil : **juste avant la première utilisation** (ou juste avant le premier bloc utilisateur quand la variable est utilisée par plusieurs blocs).

Trouvez les erreurs dans ce programme

```
int entier1, entier2;
entier1 = 0;
{
   entier2 = 2;
   entier3 = entier2+3;      // entier3 pas déclaré
   int entier3;
   entier3 = entier1+entier2;
}
{
   boolean b;
   {
      b=entier3>0;           // entier3 inconnu dans ce bloc
      double x=2.5*entier1, y=-2.4;
      x = x+y-3.2*entier1;
      b = x>entier2;
    }
    {
      b = x<y;               // x et y inconnus dans ce bloc
      int entier1=0;         // redéclaration interdite
    }
}
```

Instruction conditionnelle

- Souvent utile de n'exécuter des instructions que quand **certaines conditions sont vérifiées**. On parle « d'instructions conditionnelles ».

  Ex : algorithme qui décide si un étudiant valide ou non un module ; il comporte une note d'oral de coefficient 1 et une note d'écrit de coefficient 2 ; la moyenne doit être supérieure ou égale à 10.

  Données initialement disponibles : notes d'écrit *ne* et d'oral *no*.

  moyenne ← *(2*ne+no)/3*
  si *moyenne ≥10*
  alors *validation*
  sinon *non validation*
  fsi

  Traduction en Java :

  – résultat *boolean*
  – données ne et no *double* supposées déjà initialisées

```
double moyenne;
moyenne=(2*ne+no)/3;
boolean valide;
if (moyenne>=10)
    valide=true;
else
     valide=false;
```

- Les **retours à la ligne** et les **indentations** ne sont pas obligatoires mais sont essentiels pour la lisibilité du programme.

  Evitez d'écrire …

  *double moyenne;moyenne=(2*ne+no)/3;boolean valide;if(moyenne >=10)valide=true;else valide=false*;

- Syntaxe :

```
if (condition)
    instruction1;
else
    instruction2;
```

  - *Condition est* une expression logique (de type *boolean*).
  - instruction peut être remplacée par bloc; donc les instructions conditionnelles peuvent contenir des instructions conditionnelles (imbriquées).
  - Conseil : toujours utiliser des blocs, même avec une seule instruction, pour faciliter la compréhension du programme.
  - La partie *else* est facultative.

Ex:
```
if (a<b) {
  double aux=a;
  a=b;
  b=aux;
}
```

Exemple : attribuer aussi le module à un étudiant ayant entre 9 et 10 de moyenne s'il a plus de 15 à l'oral.

Algorithme

```
moyenne ← (2*ne+no)/3
si moyenne ≥10
   alors validation
   sinon
      si moyenne < 9
         alors non validation
         sinon
            si no ≥15
               alors validation
               sinon non validation
            fsi
      fsi
fsi
```

Java

```
double moyenne;
moyenne=(2*ne+no)/3;
boolean valide;
if (moyenne>=10) {
    valide=true;
} else {
    if (moyenne<9) {
        valide=false;
    } else {
        if (no>=15) {
            valide=true;
        } else {
            valide=false;
        }
    }
}
```

Instruction de choix : *switch*

- Permet d'effectuer un choix **entre plus de deux cas** sans utiliser une imbrication complexe de *if*.

Ex:

```
System.out.println("Choisir une option : 1, 2 ou 3");
Scanner sc=new Scanner(System.in);
int choix=sc.nextInt();
switch(choix) {
    case 1:
       System.out.println("Choix 1");
       break;
    case 2:
       System.out.println("Choix 2");
       break;
    case 3:
       System.out.println("Choix 3");
       break;
    default:
       System.out.println("Choix non reconnu");
       break;
}
```

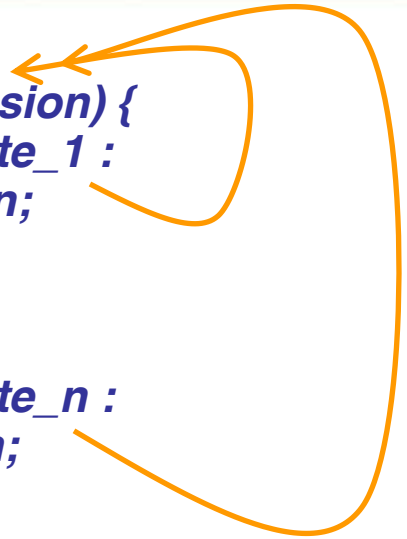
Menu avec *choix* qui vaut 3

Programme équivalent avec des *if* imbriqués :

```
System.out.println("Choisir une option : 1, 2 ou 3");
Scanner sc=new Scanner(System.in);
int choix=sc.nextInt();
if (choix==1) {
    System.out.println("Choix 1");
} else {
     if (choix==2) {
          System.out.println("Choix 2");
     } else {
          if (choix==3) {
               System.out.println("Choix 3");
          } else {
               System.out.println("Choix non reconnu");
          }
     }
}
```

- Syntaxe :

```
switch(expression) {
  case étiquette_1 :
     instruction;
     ...
     break;
  ...
  case étiquette_n :
     instruction;
     ...
     break;
  default:
     instruction;
     ...
     break;
}
```



- Etiquette = expression obligatoirement constante (sans variables).
- Etiquette : type compatible avec le type de l'expression (souvent *char* ou *int*).

- Chaque *case* (et le *default*) avec autant d'instructions que souhaité.
- Le *switch* avec autant de *case* que nécessaire, à condition que les étiquettes soient **uniques**.
- Au maximum un *default*.
- **Fonctionnement** :
  1) le processeur évalue l'expression qui suit le mot *switch,*
  2) le processeur cherche la première étiquette égale à la valeur trouvée et exécute toutes les instructions à partir d'elle,
  3) Si le processeur ne trouve pas une telle étiquette et si l'étiquette *default* existe, il exécute les instructions du *default*,
  4) quand le processeur rencontre l'instruction *break*, il termine l'exécution du *switch* en sautant à l'instruction qui suit l'accolade fermante.

  Oubli du *break* → le processeur continue l'exécution en passant au *case* suivant sans tester l'étiquette !

Bases de la programmation

Programmation objet (Objets, classes, attributs)

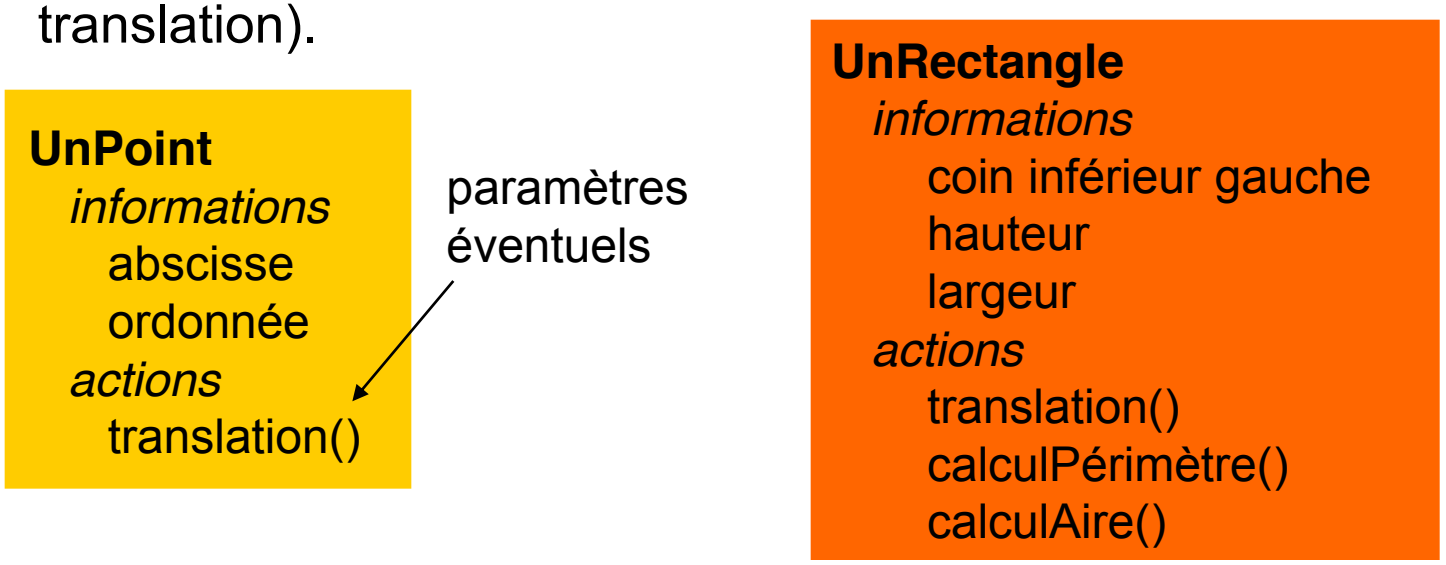
Programmation orientée objet

- Java = langage de programmation « orienté objet ».
- Tâches d'un tel programme = actions effectuées par des **objets** (entités).
- Certaines actions → objets **communiquent** entre eux (échangent des informations).
- Objets identifiés par leurs **propriétés**.

  Ex : un programme de dessin géométrique manipule de nombreux objets; des points, définis par leurs abscisses et ordonnées; des rectangles, définis par leur coin inférieur gauche, leur largeur et leur hauteur…
- Les **modèles** qui définissent les propriétés des objets sont appelés **classes** (ou « types objet »).

Le programme de dessin géométrique définit les classes *UnPoint, UnRectangle*… qui précisent les propriétés des points, des rectangles… c'est-à-dire :

- leurs informations (**attributs**),
- les actions qu'ils autorisent (**méthodes**). Tout point offre un « service » qui permet de le translater. Il faut « envoyer un message » à un objet de la classe *UnPoint* pour lui demander de se translater d'un certain déplacement (appeler sa méthode translation).



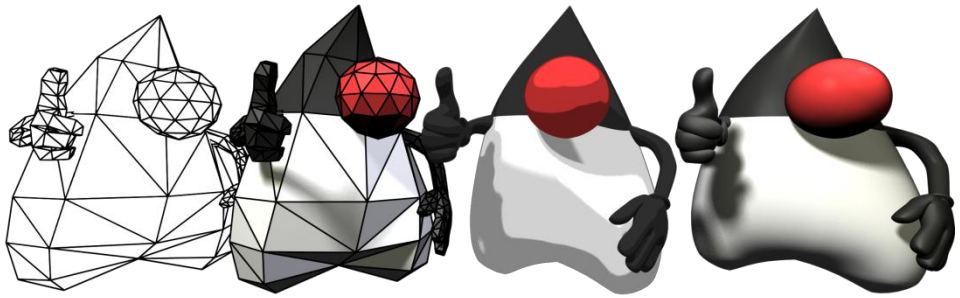
Remarque : l'attribut « coin inférieur gauche » est un point c'est-à-dire un objet de classe *UnPoint*.

- La classe *UnRectangle* est un modèle **général** de rectangle. Ex : tous les rectangles ont la même formule pour calculer leur aire.
- Un objet de la classe *UnRectangle* est un rectangle **particulier** avec sa propre largeur et sa propre longueur. Cet objet (« **instance** ») est construit à partir du modèle donné par la classe par une méthode spéciale appelée « **constructeur** ».

  On peut créer un nombre quelconque d'objets à partir d'une classe.
- 2 rectangles appartiennent à la même classe donc utilisent la même formule de calcul d'aire. Mais ils ont 2 largeurs distinctes → 2 résultats différents quand on leur demande de calculer leur aire.

Résumé du vocabulaire

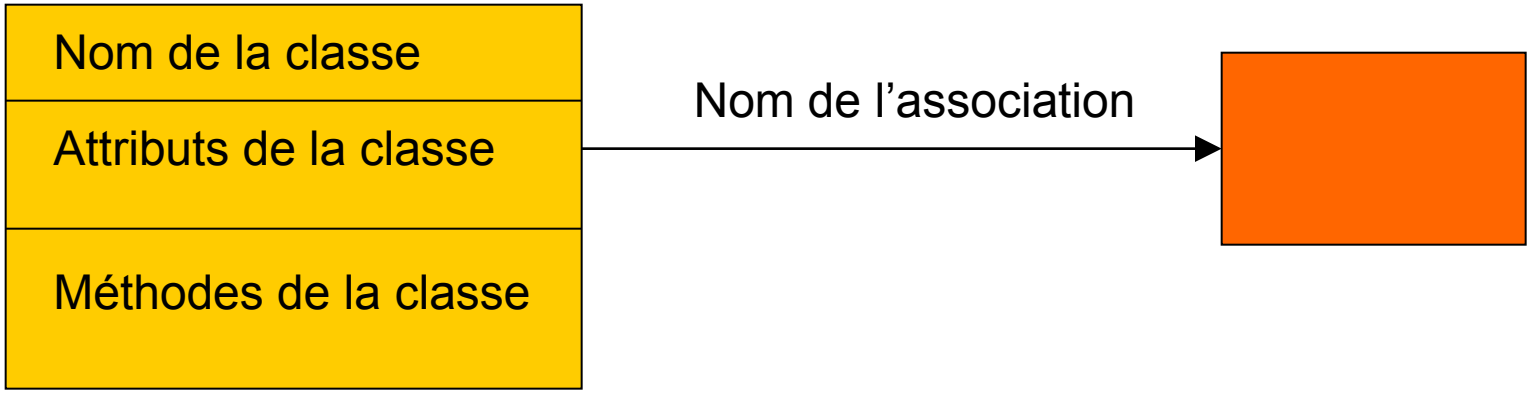
- **Objet** : un point, un rectangle.
- **Classe** : modèle de point, modèle de rectangle.
- **Attribut** : *largeur*, *abscisse*.
- **Méthode** : *translation()*, *calculAire()*
- **Instance** : un point est un objet créé à partir du modèle de point (classe *UnPoint*).
- **Constructeur** : méthode qui permet la création d'une instance.

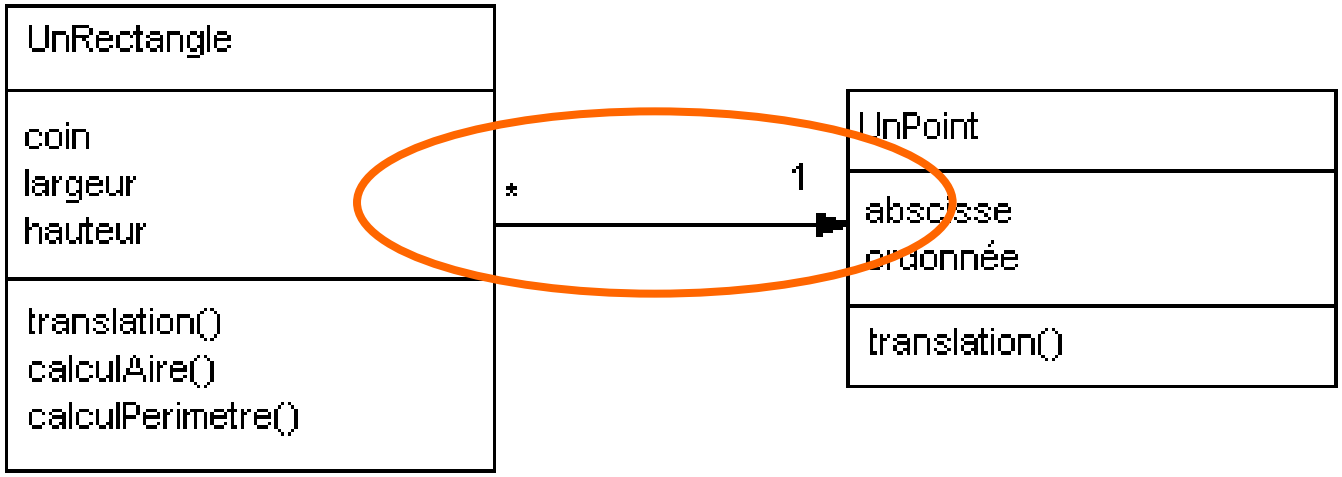


- UML = langage de modélisation **graphique** orienté objet.
- Propose plusieurs types de diagrammes parmi lesquels les « **diagrammes de classes** ».
- Décrivent les **classes** et les **relations** entre classes.

  Parmi les relations on se limite ici aux **associations** qui représentent des relations entre des instances.

  Ex : une personne travaille pour une entreprise, une voiture a un propriétaire...





Les **cardinalités** d'une association expriment le nombre d'instances pouvant participer à l'association.

Ex : on a exactement 1 point associé à un rectangle. Pour un point, on peut avoir plusieurs rectangles associés (noté *). On peut aussi écrire 0..1 pour indiquer une instance optionnelle dans l'association.

La flèche (facultative) indique la **navigabilité**, c.à.d. la possibilité d'accès : UnRectangle doit fournir un moyen d'accéder au point associé (à son coin).

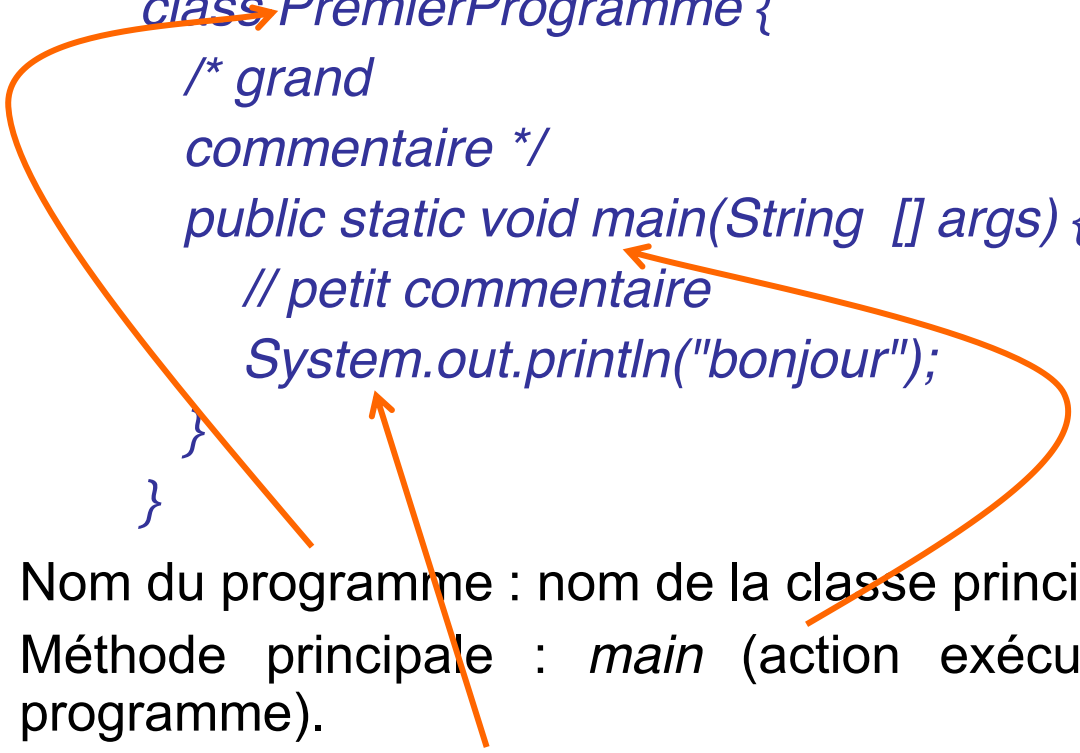
- Programme Java = { classes }.
- Classe = { attributs, constructeurs, méthodes }.
- Il faut déterminer les objets qui vont interagir pour accomplir la tâche demandée.
- Les objets échangent des messages (appels de méthode).

  Quand un objet A envoie un message à un objet B c'est B qui réagit et exécute la méthode appelée en tenant compte de son état (et en le modifiant éventuellement). Puis B envoie un résultat à A (réponse au message). Eventuellement, un objet peut s'envoyer des messages à lui-même.

Un premier programme

- Ne définit qu'une seule classe, sans attribut, ni méthode, ni constructeur.

```
class PremierProgramme {
  /* grand
  commentaire */
  public static void main(String  [] args) {
     // petit commentaire
     System.out.println("bonjour");
  }
}
```



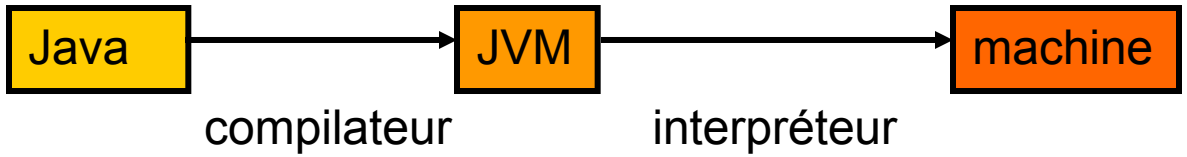
Nom du programme : nom de la classe principale (une seule ici).

Méthode principale : *main* (action exécutée au lancement du programme).

Chemin des méthodes : l'instruction d'affichage demande à l'attribut *out* de la classe *System* d'exécuter la méthode *println*.

Compilation - exécution

- Le code java est compilé (traduit) en un pseudo-code machine qui est interprété (exécuté) par une machine virtuelle Java (JVM).
- Chaque ordinateur doit savoir faire fonctionner la JVM.



- Utilisation
  - Ecriture du programme PremierProgramme.java
  - Compilation : *javac PremierProgramme.java*
    (crée Premier Programme.class)
  - Exécution : *java PremierProgramme*
- Les environnements de programmation (IDE) facilitent le travail avec des fonctionnalités pour aider par exemple à corriger les programmes. Mais le trio « compilation-JVM-interprétation » est toujours présent.

Classes et objets

- Rappels :
  - Une classe est le modèle d'un ensemble d'objets.
  - Une classe décrit la façon de fabriquer un objet (constructeurs), les propriétés des objets (attributs) et la façon de répondre aux messages reçus (méthodes).
  - Une classe n'a pas d'existence concrète.
  - Un objet est une instance d'une classe. Un objet a une existence concrète (en mémoire).
- Création d'un objet → allocation de mémoire pour les données liées et assignation d'une **référence**.
- Référence : permet de désigner un objet particulier et d'accéder à ses propriétés.
- En Java, les variables contiennent soit des valeurs de type fondamental soit des références d'objets.

  Une variable ne contient jamais un objet.

Définition d'une classe

- Syntaxe (forme simple)

```
class NomClasse {
    déclaration des attributs
    déclaration des constructeurs
    déclaration des méthodes
}
```

NomClasse est un identificateur (dont chaque mot commence en général par une majuscule).

Ex : classe UnPoint, classe Client, classe Commande

(des explications détaillées suivent dans le reste du chapitre).

```
// définition simple de la classe UnPoint
class UnPoint {
    // attributs
    int abscisse, ordonnee;

    // constructeurs
    UnPoint(int a, int o) {
       this.abscisse=a;
       this.ordonnee=o;
    }

    // méthodes
    void translation(int dx, int dy) {
        this.abscisse=this.abscisse+dx;
        this.ordonnee=this.ordonnee+dy;
    }
}
```

```
// définition simple d'une classe Client
class Client {
   // attributs
   int numero_client;
   int code_banque;
   int code_agence;
   int numero_compte;
   char lettre_compte;
   boolean autorise_prelevement;
   // constructeurs

  Client(int b,int a,int nc,char lc,int ncl) {
      this.numero_client=ncl;
      this.code_banque=b;
      this.code_agence=a;
      this.numero_compte=nc;
      this.lettre_compte=lc;
      this.autorise_prelevement=true;
   }
```

```
 // méthodes
void autorisePrelevement() {
this.autorise_prelevement=true;
   }

void annulePrelevement() {
this.autorise_prelevement=false;
   }
}
```

```
// définition simple d'une classe Commande
class Commande {
   // attributs
   Client debiteur;
   int code_article;
   int quantite;
   double prix_unitaire;
   // constructeurs
   Commande(Client c, int ca, double p) {
      this.debiteur=c;
      this.code_article=ca;
      this.quantite=1; // choix par défaut
      this.prix_unitaire=p;
   }
   // méthodes
   int montantTotal() {
       return this.quantite*this.prix_unitaire;
   }
}
```

Les attributs

- Syntaxe de déclaration d'un attribut : celle de déclaration d'une variable.
- Mais **un attribut n'est pas une variable** : ne désigne pas un emplacement mémoire; on ne peut pas lui affecter de valeur lors de sa déclaration.

  Ex : ***int abscisse;*** // dans la classe *UnPoint*

  Tout objet de la classe *UnPoint* a un attribut d'identificateur *abscisse* et de type *int*. **Chaque objet a son propre attribut abscisse**.
- Un objet peut utiliser ses attributs comme des variables : affectation de valeurs, expressions qui les contiennent…
- ***this.abscisse*** indique l'abscisse de l'objet qui est en train d'être manipulé
- Il est possible (mais déconseillé) d'accéder aux attributs d'autres objets.

Les constructeurs

- Indique une façon de « fabriquer » un objet instance de la classe. Il s'agit généralement d'initialiser les attributs de l'objet à créer (tout ou partie).
- Syntaxe de base :

```
NomClasse (liste paramètres formels) {
    corps du constructeur
}
```

- Nom du constructeur : **celui de la classe** à laquelle il correspond.
- Appel du constructeur : expression de syntaxe spéciale.

Ex : pour construire un objet de classe *UnPoint* de coordonnées initiales (35,60), on écrit l'expression ***new UnPoint(35, 60)***

Pour construire un objet de classe *Client*, on écrit une expression comme ***new Client(30001, 7150, 11111, 'A', 31524).***

```
// définition simple de la classe UnPoint
class UnPoint {
    // attributs
    int abscisse, ordonnee;

    // constructeurs
    UnPoint(int a, int o) {
       this.abscisse=a;
       this.ordonnee=o;
    }

    // méthodes
    void translation(int dx, int dy) {
        this.abscisse=this.abscisse+dx;
        this.ordonnee=this.ordonnee+dy;
    }
}
```

- ***UnPoint p = new UnPoint(35,60)*** → construction d'une nouvelle instance de *UnPoint*. p est la **référence** de cet objet. On ne peut accéder à l'objet que grâce à cette référence.
- Quand une instance a été créée ses attributs désignent des **cases mémoires propres à l'objet**.
- 35, 60 est la liste des **paramètres effectifs** grâce à laquelle le constructeur *UnPoint(int a, int o)* va connaître les valeurs des **paramètres formels** a et o pour la création de la nouvelle instance.

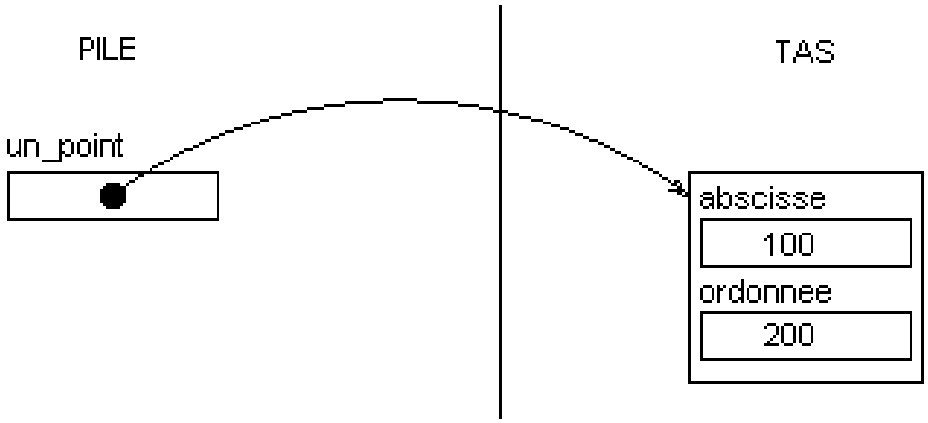
Le nouvel objet *UnPoint* est créé avec ses attributs *abscisse* et *ordonnee* qui valent initialement 35 et 60.

Pile et tas

- Pour manipuler un objet il faut :
  - déclarer une variable de référence de classe *UnPoint* :

    UnPoint un_point; (on parle de variable de type objet).
  - créer un point, par ex de coordonnée initiales (100, 200), dont la référence est placée dans la variable *un_point* :

    un_point = new UnPoint(100, 200);
- Représentation en mémoire : la **pile** stocke les variables du programme et le **tas** stocke les objets



- Un programme ne peut accéder aux données du tas que par l'intermédiaire des références dans les variables.
- Plusieurs variables peuvent désigner un même objet (elles contiennent la même référence).
- Il est possible d'accéder aux attributs d'un objet, mais cela est le plus souvent déconseillé.

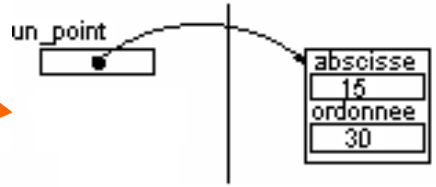
  Ex : ***int a = un_point.abscisse;***

  pour accéder à l'abscisse de l'objet *un_point.*
- Si la référence d'un objet n'est plus placée dans aucune variable ni dans aucun attribut → le programme n'a plus aucun moyen d'accéder à cet objet qui est donc inutile. Il est **automatiquement supprimé** du tas (par le *garbage collector* ou « ramasse miettes »).

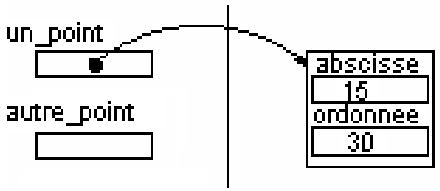
Exemples :

UnPoint un_point;

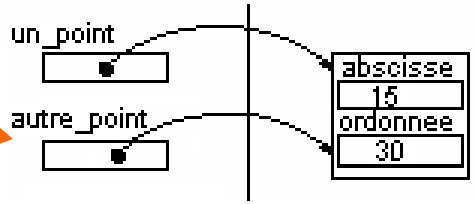
un_point=new UnPoint(15,30);



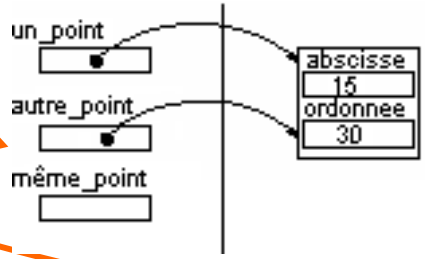
UnPoint autre_point;



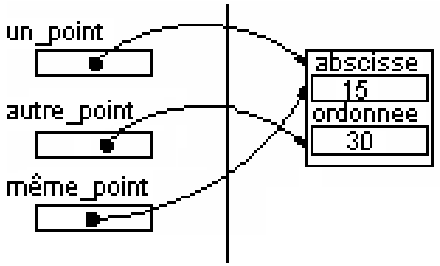
autre_point=un_point;



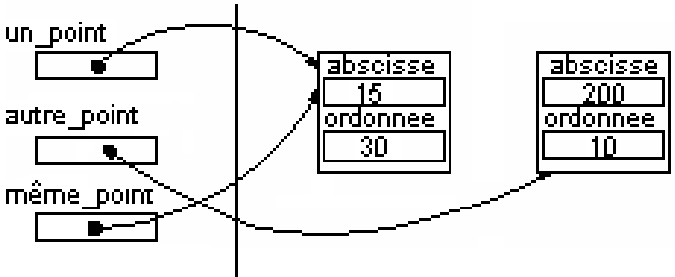
UnPoint meme_point;



meme_point=autre_point;



autre_point=new UnPoint(200,10);



Paramètres formels et effectifs

- Liste de paramètres formels : suite de déclarations de paramètres formels séparées par des ‘,’.
  Déclaration : similaire à celle d’une variable
  type identificateur
- Paramètre effectif : expression évaluable.
  Liste des paramètres effectifs séparés par des ‘,’.
  Type d’un paramètre effectif : type fondamental (*double*, *int…)* ou type objet (valeur = référence à un objet de ce type).
- Appel du constructeur : affectation des paramètres effectifs (valeurs) aux paramètres formels (variables), dans l’ordre de leur déclaration **si les types coïncident.**
- Le corps de la méthode peut utiliser les paramètres formels comme des variables (portée limitée au bloc du corps de la méthode comme pour les variables déclarées dans le corps de la méthode).

Constructeurs multiples

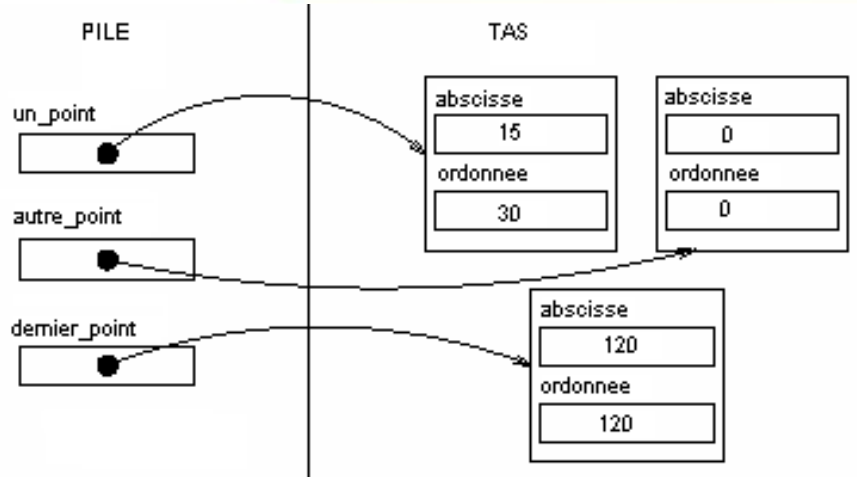
- **Un constructeur est identifié par la liste des types de ses paramètres formels** → on peut avoir plusieurs constructeurs pour une même classe avec des listes de paramètres formels de tailles distinctes ou dont un type au moins diffère.
- Le « **constructeur vide** » a zéro paramètre.

Ex :

```
class UnPoint {
    int abscisse, ordonnee;
    UnPoint(int a, int o) {  // constructeur habituel; initialise les attributs
        this.abscisse=a;
        this.ordonnee=o;
     }
    UnPoint() {    // constructeur vide (construit un point à l'origine)
        this.abscisse=0;
        this.ordonnee=0;
    }
    UnPoint(int coord) {  // donne la même valeur aux 2 coordonnées
        this.abscisse=coord;
        this.ordonnee=coord;
    }
}
```

Ex :

```
UnPoint un_point;
un_point=new UnPoint(15,30);
UnPoint autre_point;
autre_point=new UnPoint();
UnPoint dernier_point;
dernier_point=new UnPoint(120);
```



- **Si aucun constructeur n'a été déclaré**, et **seulement dans ce cas**, Java fournit un constructeur par défaut, qui est toujours un constructeur vide.

Javadoc

- Javadoc est un outil qui permet de décrire les attributs, méthodes et constructeurs d'une classe. Il s'agit de commentaires particuliers qui permettent de créer automatiquement une documentation pour l'utilisateur.

- La javadoc est devenu un standard incontournable en JAVA et vous serez amené à en produire en entreprise.

- Ajouter la javadoc consiste à ajouter des commentaires particuliers débutant par /** et finissant par */.

- A l'intérieur de ces commentaires, on écrit le descriptif du constructeur.

- Pour chaque paramètre, on y ajoute le mot clef @param suivi du nom du paramètre et de son descriptif.

Javadoc

La classe **UnPoint** s'écrit donc de la manière suivante !

```
1  Class UnPoint{
2    /**
3    * attribut abscisse du point
4    */
5    int abscisse;
6    /**
7    * attribut ordonnee du point
8    */
9    int ordonnee;
10
11   /**
12   * constructeur vide construit un nouveau point
13   * a partir de coordonnees passees en parametre
14   * @param a abscisse du point cree
15   * @param o ordonnee du point cree
16   */
17   UnPoint(int a, int o){
18     this.abscisse=a;
19     this.ordonnee=o;
20   }
21 }
```

Tests

La démarche consiste à tester progressivement tous les cas qui peuvent se produire et à vérifier que ces cas conduisent bien au résultat attendu.

- Dans un premier temps, les classes permettant de tester votre programme vous seront fournies. Elles vous permettront de valider votre travail de manière autonome.

- Dans un second temps, vous serez amenés à écrire vos propres classes de test pour valider les programmes que vous aurez écrits.

Bases de la programmation

Programmation objet (Méthodes)

Déclaration d'une méthode

- Méthode : action exécutée par un objet en fonction de la valeur de ses attributs.
- Syntaxe de base :

```
type_retourné nomMéthode (liste_paramètres_formels) {
        corps_de_la_méthode
}
```

nomMéthode est un identificateur qui débute en général par une minuscule avec une majuscule pour les mots suivants.

- Corps de la méthode = bloc. Ses instructions sont exécutées **par l'objet** lors de l'**appel** de la méthode.
- Comme les constructeurs, une méthode est identifiée par son nom et la liste des types de ses paramètres formels.
- Une classe peut avoir des méthodes distinctes de même nom si ces méthodes ont des listes de paramètres formels de tailles distinctes ou dont un type au moins diffère.

Appel d'une méthode

- Appel de méthode = expression.
- Permet une communication (transmission d'information) dans les 2 sens :
  - l'instruction appelante envoie des informations à la méthode par les **paramètres effectifs** (comme avec les constructeurs)
  - la méthode appelée répond par l'intermédiaire du **résultat** de l'appel qui est une valeur de type *type_retourné*
- Appel de la méthode : **nom complet** de la méthode suivi entre parenthèses par la liste des paramètres effectifs.

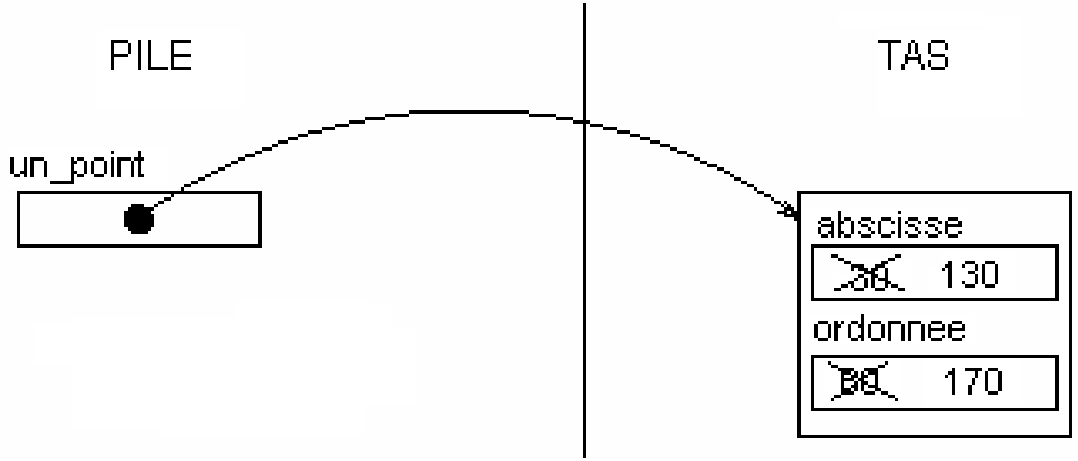
  Nom complet de la méthode = **référence de l'objet suivie d'un point suivi du nom de la méthode et de ses paramètres effectifs**.

  Ex : ***un_point.translation(100, 110);***

Exécution d'une méthode

- Evaluation d'une expression « appel de méthode » : les paramètres effectifs (valeurs) sont affectés aux paramètres formels (variables) puis les instructions du corps de la méthode sont exécutées.
- L'exécution est spécifique à l'objet indiqué par la référence. L'exécution peut modifier son état (valeur de ses attributs).

  Ex : *un_point* contient la référence de l'objet créé par *un_point = new UnPoint(30, 60);* l'évaluation de *un_point.translation(100, 110);* modifie l'objet placé dans le tas.



Modification des paramètres

- **Paramètres formels = variables locales aux méthodes** → leur modification éventuelle dans le corps de la méthode **ne se répercute pas** au bloc où se trouve l'appel de la méthode.

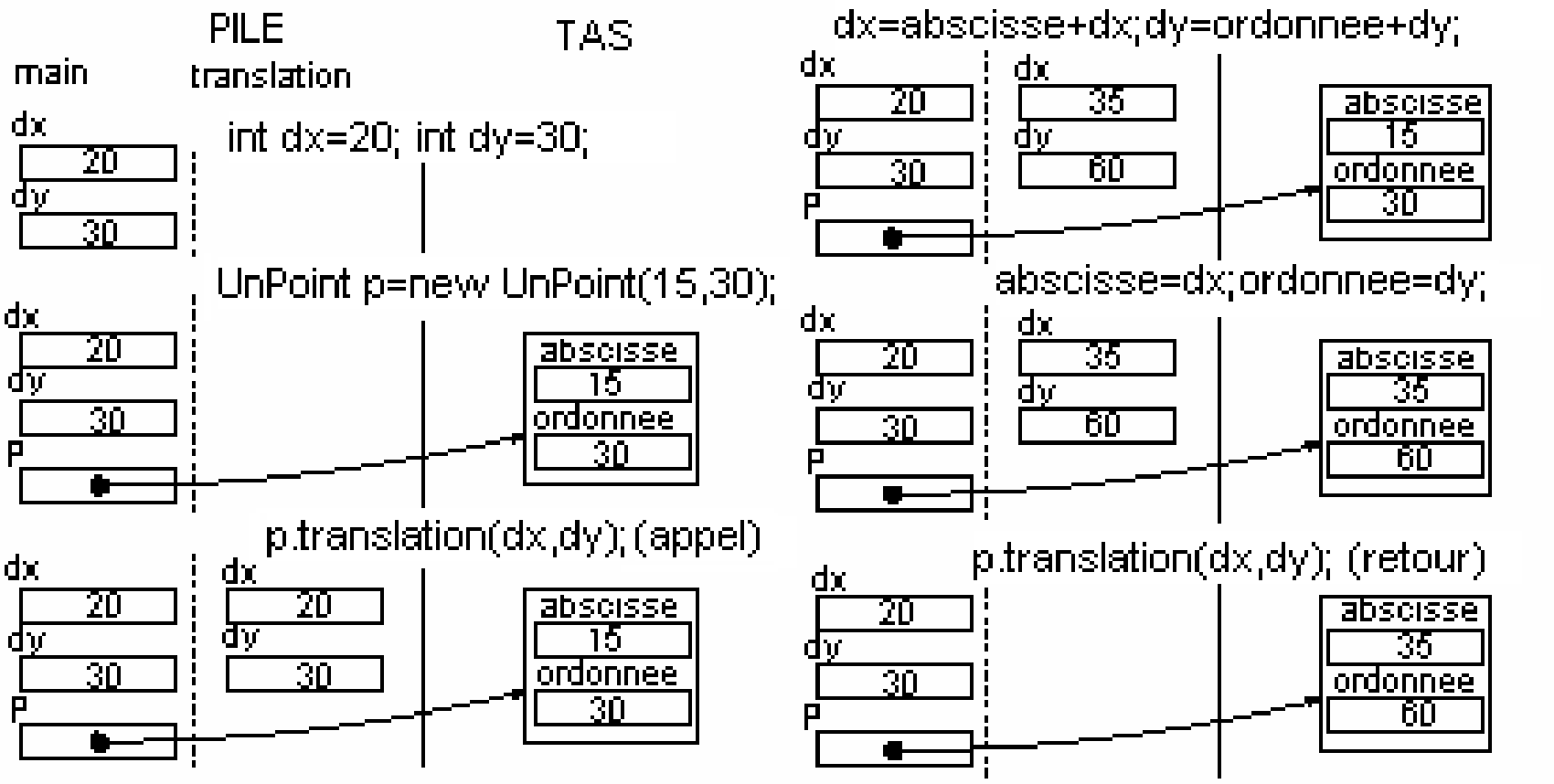
Ex : on reprogramme la translation de la façon suivante

```
class UnPoint {
   int abscisse,ordonnee;
   UnPoint(int a, int o) {
      abscisse = a;
      ordonnee = o;
   }
   void translation(int dx, int dy) {
       dx = this.abscisse + dx;          // modifie dx de translation
       dy = this.ordonnee + dy;          // modifie dy de translation
       this.abscisse = dx;
       this.ordonnee = dy;
   }
}
```

On exécute le programme suivant :

```
class ProgrammeUnPoint {
    public static void main(String[ ] args) {
        int dx = 20;
        int dy = 30;
        UnPoint p=new UnPoint(15, 30);
        p.translation(dx, dy);
        System.out.println(dx + dy);
    }
}
```

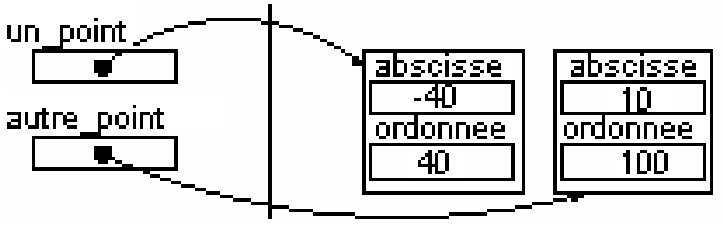
dx et *dy* du *main* et de *translation* ne désignent **pas les mêmes cases mémoires**. Le résultat affiché est 50 (car *dx* et *dy* du *main* sont inchangées).


PILE
TAS
main
translation
dx
20
dy
30
int dx=20; int dy=30;
UnPoint p=new UnPoint(15,30);
P
abscisse
15
ordonnee
30
p.translation(dx,dy); (appel)
dx=abscisse+dx;dy=ordonnee+dy;
35
60
abscisse=dx;ordonnee=dy;
p.translation(dx,dy); (retour)

Méthode sans résultat

- On peut vouloir que l'objet exécute une méthode sans avoir besoin d'un résultat. C'est le cas de *translation*. On indique ***void*** comme type retourné (cf. exemple précédent).
- Appel de cette méthode : instruction qui ne contient que l'appel de la méthode suivi d'un point-virgule.

Ex:
```
UnPoint un_point;
un_point=new UnPoint(15,30);
UnPoint autre_point;
autre_point=new UnPoint();
un_point.translation(-60,0);
autre_point.translation(10,100);
un_point.translation(5,10);
```



Méthode avec résultat

- Si le type retourné ≠ *void*, l'évaluation de l'appel de la méthode est une valeur qui est fournie par une instruction spéciale dans le corps de la méthode :

  return expression_valeur_résultat;

- Le type de cette valeur doit être le *type_retourné* indiqué dans la déclaration.

Ex :

```
class UnRectangle {
    UnPoint coin;
    int largeur,hauteur;
    UnRectangle() {
        this.coin=new UnPoint();
        this.largeur=1;
        this.hauteur=1;
    }
int perimetre() {
return 2*(this.hauteur+this.largeur);
    }
}
```

```
UnRectangle un_rectangle;
un_rectangle=new UnRectangle();
int var;
var=un_rectangle.perimetre();
                    // vaut 4
```

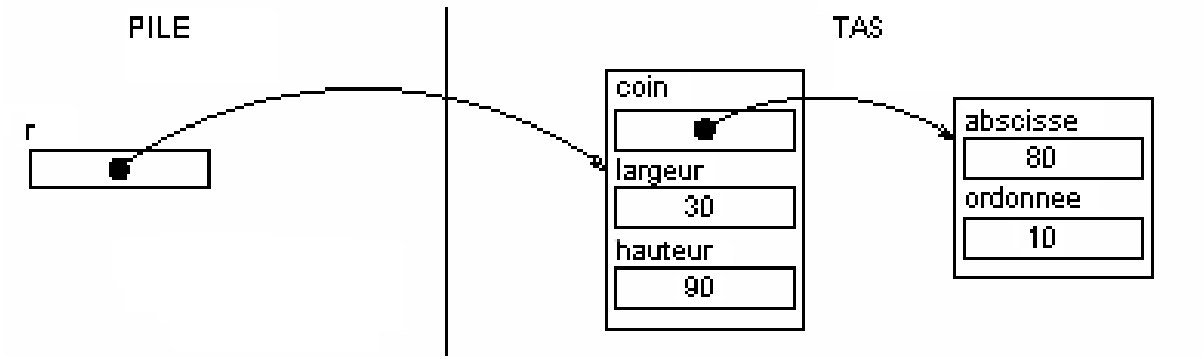
Communication entre méthodes d'objets distincts

- Pour réaliser une action un objet peut faire appel à une méthode d'un autre objet. Dans ce cas, le corps de la méthode contient au moins une instruction qui provoque elle-même un appel de méthode.

  Ex : pour translater un rectangle, il suffit de translater son coin inférieur gauche sans changer ni sa hauteur, ni sa largeur.

```
class UnRectangle {
  UnPoint coin;
  int largeur, hauteur;
  UnRectangle(UnPoint cig, int l, int h) {
    this.coin=cig;
    this.largeur=l;
    this.hauteur=h;
  }
  void translation(int dx, int dy) {
     this.coin.translation(dx, dy);
  }
}
```

```
class ProgrammeTestRectangles {
    public static void main(String[] args) {
        UnRectangle r;
        r=new UnRectangle(new UnPoint(20,50),30,90);
        r.translation(60,-40);
    }
}
```

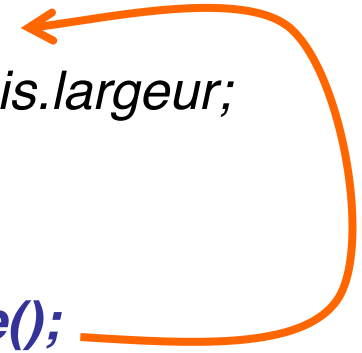


Communication entre méthodes d'un même objet

- Quand une méthode fait appel à une autre méthode **du même objet** on peut désigner la méthode appelée par son **nom incomplet** c'est à dire **sans indiquer la référence (on utilise seulement *this*)**.

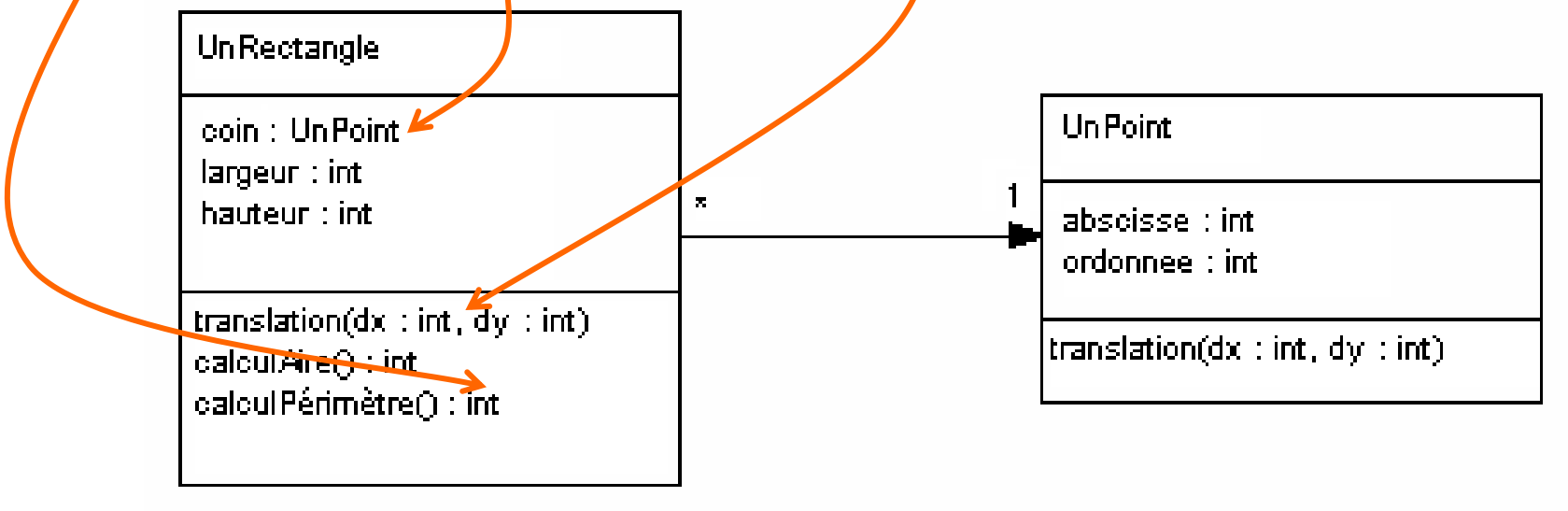
Ex : (pas très réaliste …)

```
class UnRectangle {
    UnPoint coin;
    int largeur,hauteur;
    int demiPerimetre() {
        return this.hauteur+this.largeur;
    }
    int perimetre() {
    int dp;
    dp=this.demiPerimetre();
    return 2*dp;
    }
}
```



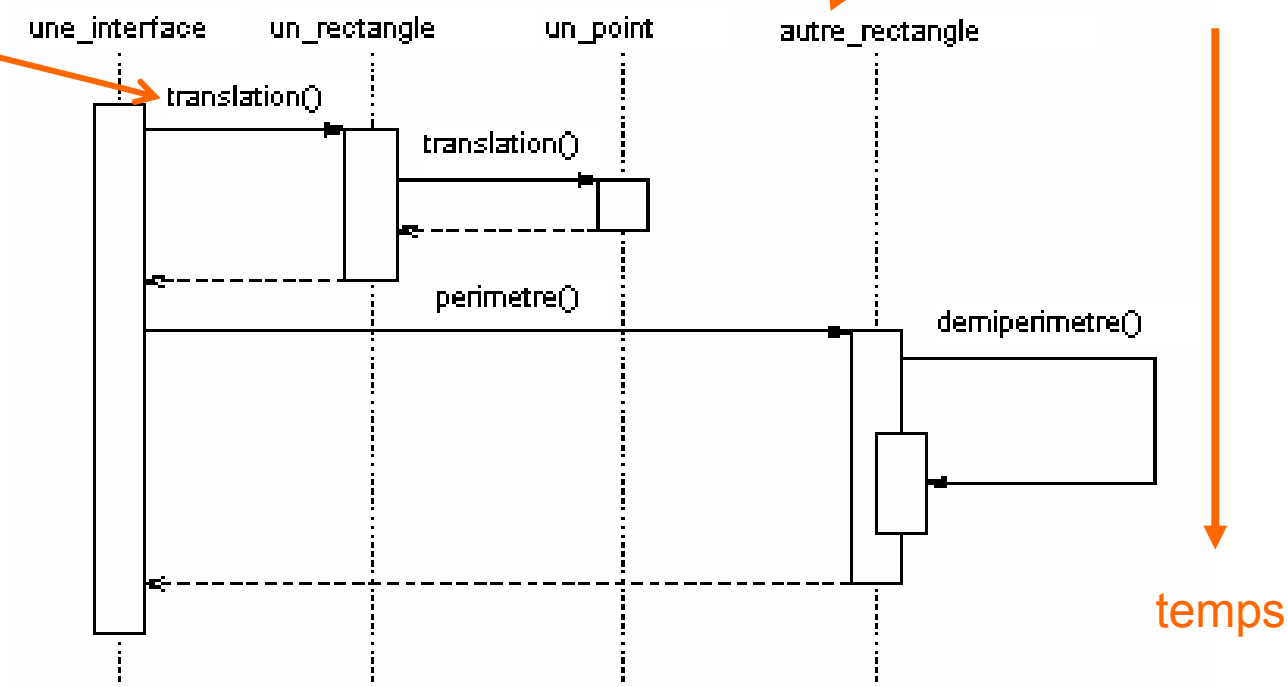
UML : diagramme de classes

- On peut indiquer dans le diagramme de classes UML les types des attributs, des paramètres et des valeurs retournées.



UML : diagramme de séquences

- Le « diagramme de séquences » UML permet des décrire les **enchaînements dans le temps** (vertical) des appels de méthode (lignes horizontales) entre les différents objets (lignes verticales).



Bases de la programmation

Programmation objet (Classes prédéfinies – API Java)

La classe String

- L'ensemble des **chaînes de caractères** est plus complexe qu'un type fondamental. Par exemple, leur taille en mémoire varie.
- La classe **String** est le modèle le plus simple de chaînes.
- On manipule donc des références sur des chaînes via des variables de type String. On peut ainsi accéder à **des méthodes déjà programmées qui facilitent beaucoup la programmation**.
- Un objet chaîne de caractères utilise en mémoire un nombre de cases variable mais on le manipule via une référence qui occupe toujours le même nombre de cases et qui peut donc être placée dans une variable de référence.
- Quelques méthodes prédéfinies de la classe String :

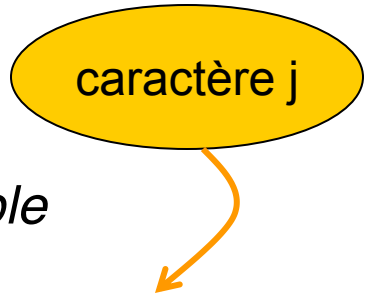
int length() // nombre de caractères de la chaîne
char charAt(int pos) // accès au caractère à une position
// donnée (premier caractère : pos=0)

int compareTo(String s)
// retourne un entier négatif si la chaîne qui exécute la méthode
// est avant la chaîne en paramètre dans l'ordre lexicographique,
// positif si après, et nul si les chaînes sont égales

String concat(String s)
// crée une nouvelle chaîne par concaténation de la chaîne en
// paramètre à la fin de celle qui exécute la méthode (on a déjà
// l'opérateur +)

Ex :

```
String ref_string;
ref_string=new String("bonjour");
/* création de l'objet et affectation à la variable
de référence */
if ((ref_string.length()>5)&&(ref_string.charAt(3)=='j')) {
    System.out.println("ok");
}   // affichera ok
```



- Création simplifiée : dès qu'un programme Java contient un texte entre guillemets un objet de classe *String* correspondant à cette chaîne de caractères est créé.
- Pour cette classe (et uniquement pour elle) il n'est donc pas nécessaire de passer par l'opérateur *new*.
  Ex : ***String ref_string;***
  ref_string="bonjour";
- La classe String correspond à des objets **non modifiables**.
  Une fois créés ils désignent toujours la même chaîne de caractères.
  L'opérateur de concaténation '+' crée un nouvel objet.

La classe StringBuffer

- Il est parfois utile d'avoir des chaînes **modifiables en taille et contenu**. La classe *StringBuffer* le permet.
- Il faut passer par un constructeur car les chaînes entre guillemets sont des *String*.
  - ***StringBuffer()*** construit un objet de classe *StringBuffer* correspondant à la chaîne vide ""
  - ***StringBuffer(String s)*** construit un objet de classe *StringBuffer* correspondant à la chaîne s.
- *length()* et *charAt()* sont disponibles comme dans *String*.

- On a aussi des méthodes de modification :
  - modification d'un caractère :

    void setCharAt(int n, char c)
  - concaténation :

    StringBuffer append(String s)

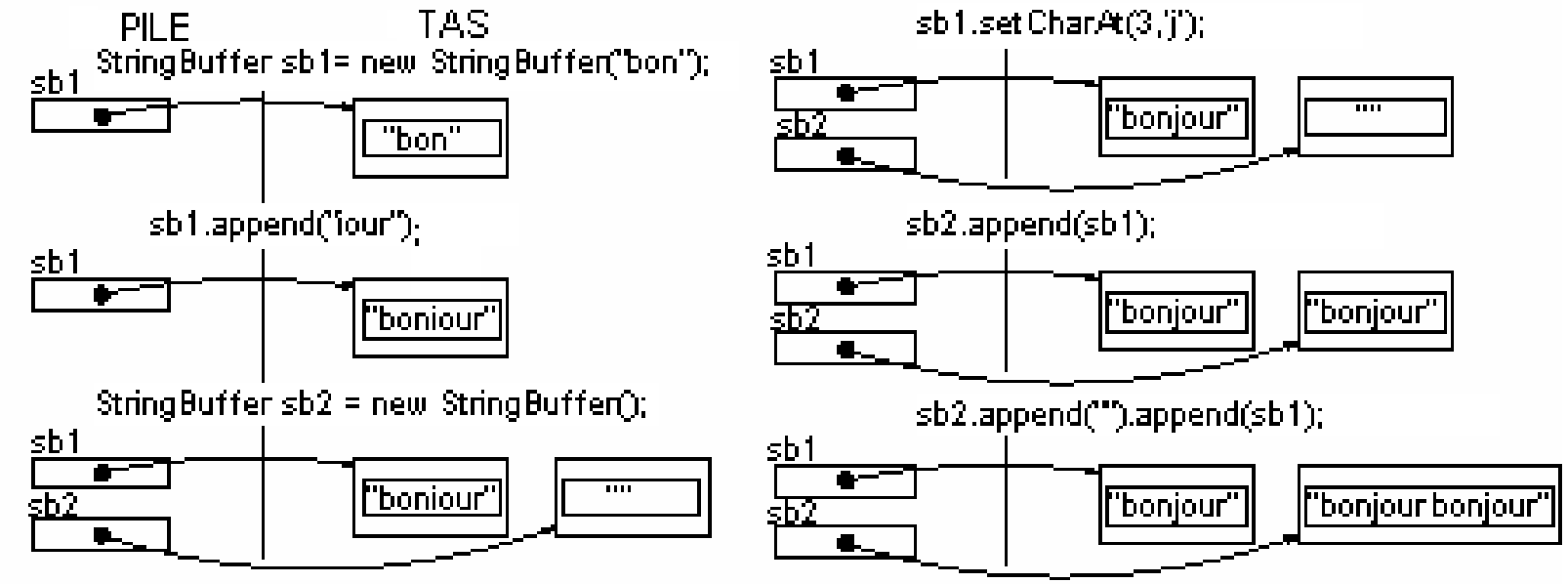
    concatène la chaîne en paramètre à celle qui exécute la méthode.
  - autre concaténation :

    StringBuffer append(StringBuffer s)
  - insertion d'une chaîne à une place donnée :

    StringBuffer insert(int offset, String str)
  - inversion de l'ordre des caractères :

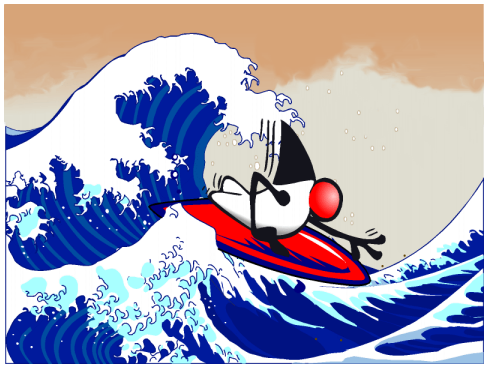
    StringBuffer reverse()

Ex : ***StringBuffer sb1=new StringBuffer("bon");***
System.out.println(sb1); // affiche "bon"
sb1.append("iour");
System.out.println(sb1); // affiche "boniour"
StringBuffer sb2=new StringBuffer();
sb1.setCharAt(3,'j');
System.out.println(sb1); // affiche "bonjour"
sb2.append(sb1);
sb2.append(" ").append(sb1);
/* car sb2.append(" ") retourne une référence à partir de laquelle
**** on peut effectuer un second appel de méthode */***
System.out.println(sb2); // affiche "bonjour bonjour"



L'API Java

- Les classes prédéfinies de Java sont si nombreuses et couvrent une telle variété de domaines qu'il est impossible de les enseigner et de les connaître toutes.
- Le but de l'enseignement de Java est de donner suffisamment de bases pour acquérir une autonomie de programmation c'est à dire la capacité à programmer en s'aidant de l'API Java (*Application Programming Interface)* qui documente toutes les classes.
- L'API est accessible en surfant sur Internet (chercher « API Java 7 » sur Google).



La notion de *package*

- Le grand nombre de classe prédéfinies dans Java et l'ajout incessant de nouvelles classes → nécessité d'un classement (thématique) par **groupes de classes** ou « *packages* ».
- Utiliser une classe prédéfinie → indiquer le *package* où elle se trouve pour accéder à sa définition.
- Les classes très courantes (ex: *String* et *StringBuffer*) sont dans le « ***package* principal** » toujours accessible.
- Pour les classes moins courantes → indiquer le *package* par l'instruction ***import*** (l'API donne le nom du package).

Ex :

```
import java.awt.*;          // importe toutes les classes du package
class Programme {
    public static void main(String args[]) {
        Image im;
        im=new Image();    // Image est dans le package java.awt
        // ...
}   }
```

- On peut définir des *packages* (non détaillé ici)

Bases de la programmation

Programmation objet (Références et objets)

Références et comparaisons/modifications

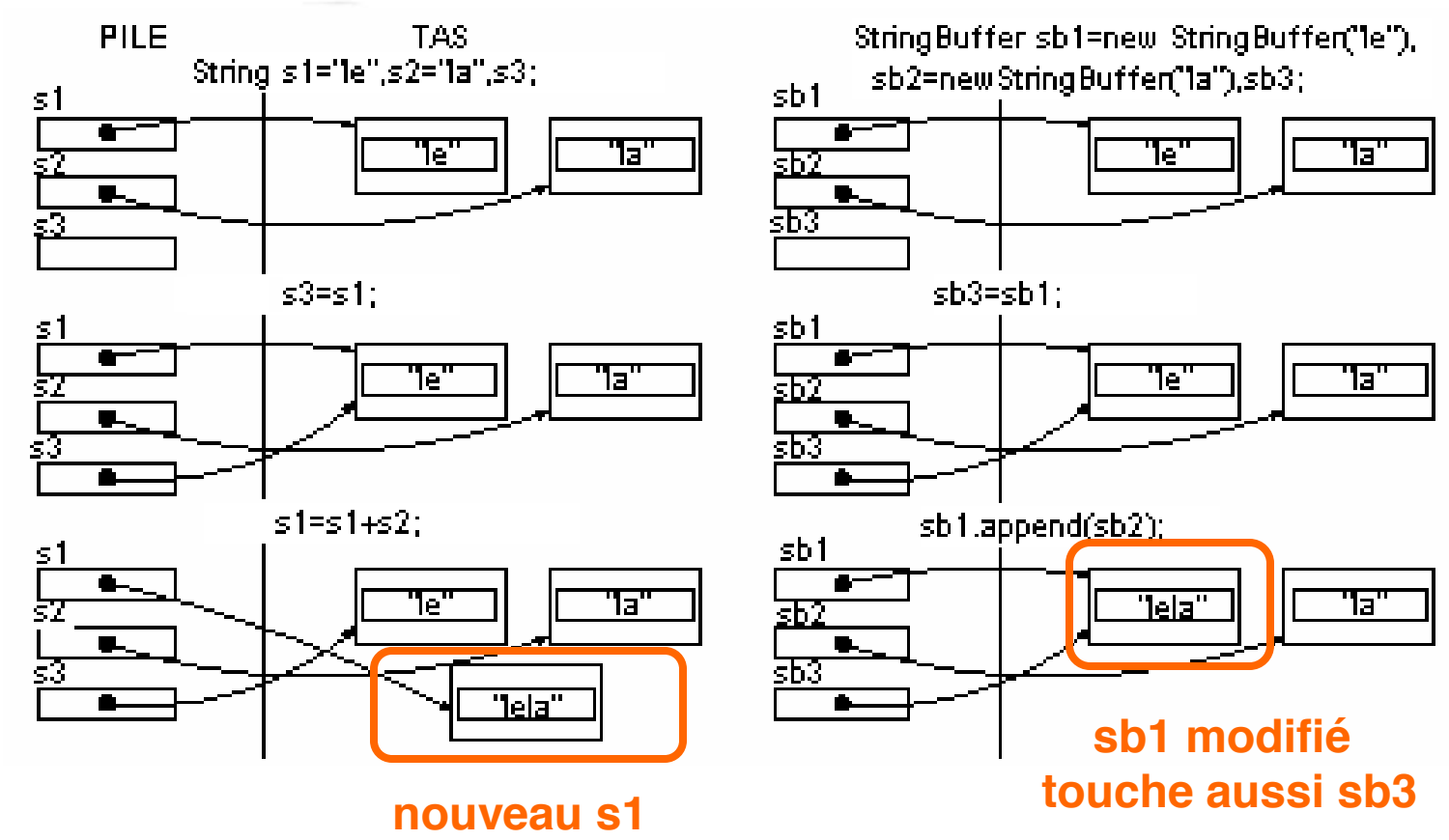
- Comparer 2 variables dont le type est une classe = **comparer le contenu de ces variables** = **comparer des références**.

  Ex : *StringBuffer sb1 = new StringBuffer("exemple");*
  StringBuffer sb2 = new StringBuffer("exemple");
  System.out.println(sb1==sb2); // affiche false

- On appelle «effet de bord» la modification d'une variable qui n'apparaît pas explicitement dans l'instruction.

  Ex : *StringBuffer* étant modifiable peut en générer (pas *String*).

```
String s1="le",s2="la",s3;
s3=s1;
s1=s1+s2;          // s1 contient "lela", s2 contient "la", s3 contient "le"
StringBuffer sb1=new StringBuffer("le"),
                     sb2=new StringBuffer("la"),sb3;
sb3=sb1;
sb1.append(sb2); // sb1 contient "lela", sb2 contient "la",
// sb3 contient "lela" ; il y a effet de bord sur sb3
```


PILE
TAS
String s1="1e",s2="1a",s3;
s1
s2
s3
"1e"
"1a"
s3=s1;
s1=s1+s2;
"1ela"
StringBuffer sb1=new StringBuffer("1e"),
sb2=new StringBuffer("1a"),sb3;
sb1
sb2
sb3
sb3=sb1;
sb1.append(sb2);
nouveau s1
sb1 modifié
touche aussi sb3

Références et paramètres

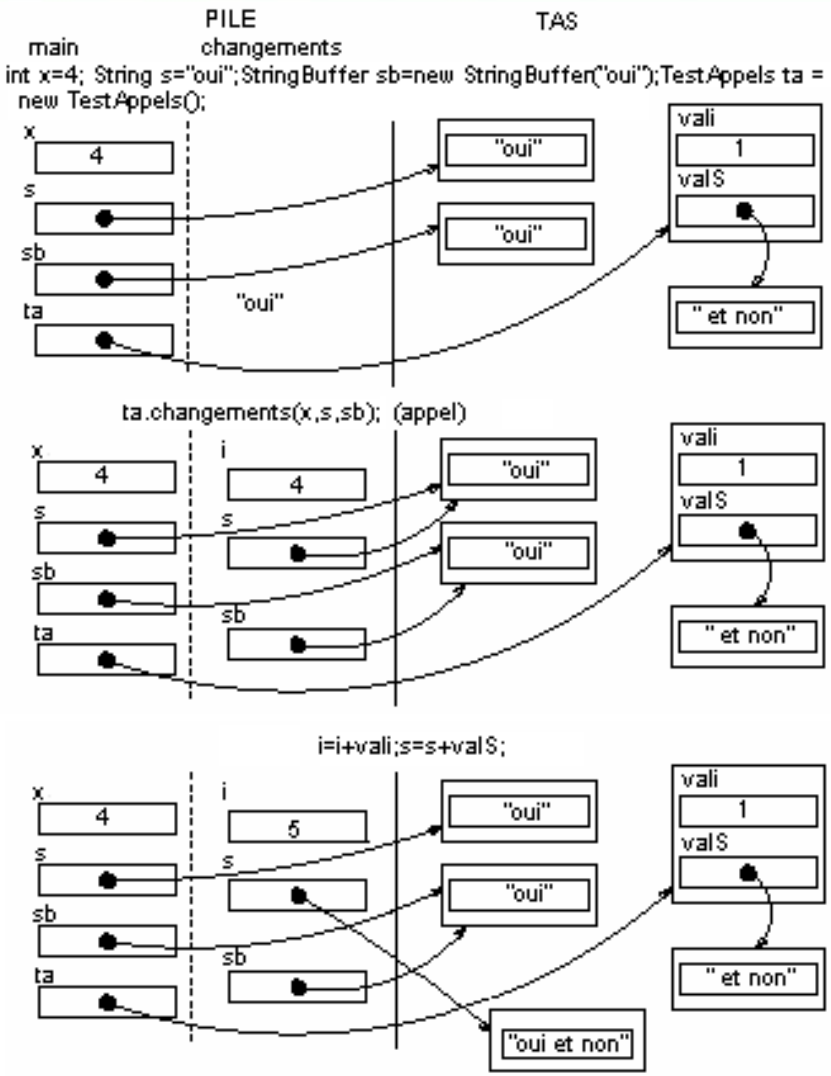
- Le paramètre effectif affecté à un paramètre formel dont le type est une classe est une référence à un objet. On manipule donc une variable de référence dans la méthode. Une modification de l'objet désigné par cette référence a donc **un effet sur la méthode appelante**.

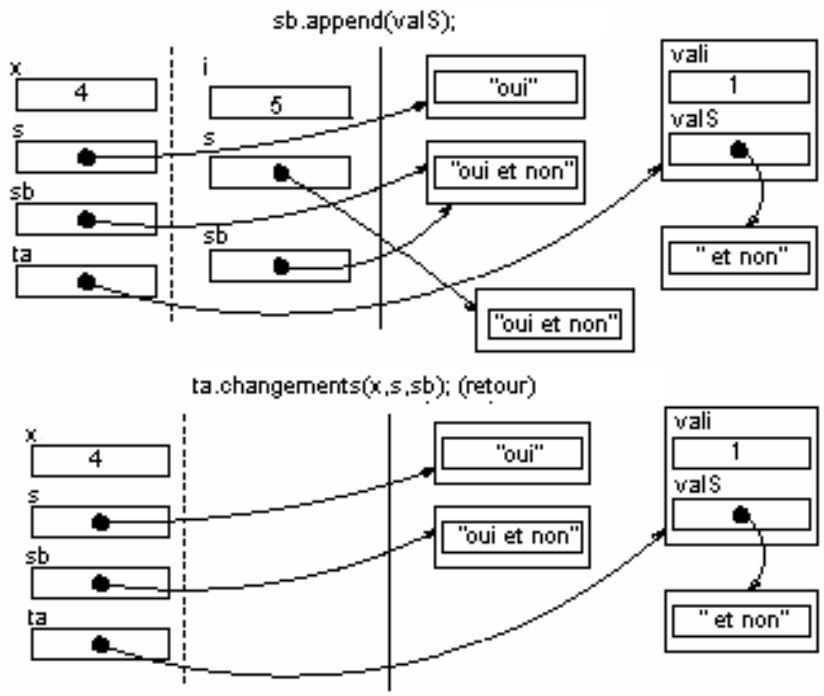
Ex :

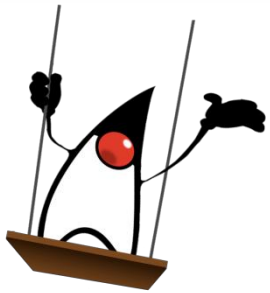
```
class TestAppels {
  int vali;
  String valS;
  TestAppels() {
    vali=1;
    valS=" et non";
  }
  void changements(int i,
    String s, StringBuffer sb) {
    i = i + vali;
    s = s + valS;
    sb.append(valS);
  }
}
```

```
class ProgrammeTest {
  public static void main(String[] args) {
    int x=4;
    String s="oui";
    StringBuffer sb=new StringBuffer("oui");
    TestAppels ta=new TestAppels();
    ta.changements(x,s,sb);
    System.out.println(x+" "+s+" "+sb);
  }
}
```

x conserve la valeur 4
s conserve la valeur "oui"
sb prend la valeur "oui et non"


PILE
TAS
main
changements
int x=4; String s="oui";StringBuffer sb=new StringBuffer("oui");TestAppels ta =
new TestAppels();
x
4
s
sb
ta
"oui"
"oui"
"oui"
vali
1
valS
" et non"
ta.changements(x,s,sb); (appel)
i
4
i=i+vali;s=s+valS;
5
"oui et non"


sb.append(valS);
x
4
i
5
s
sb
ta
"oui"
"oui et non"
vali
1
valS
" et non"
ta.changements(x,s,sb); (retour)



Mauvais usage des attributs

- Modification directe des attributs.

Ex :

```
class UnRectangle {
   UnPoint coin;
   int largeur, hauteur;

   UnRectangle() {
   this.coin = new UnPoint();
   this.largeur = 1;
   this.hauteur = 1;
        }

   UnRectangle(int x, int y,
              int l, int h) {
    this.coin = new UnPoint(x,y);
    if (l<1) l = 1;
    if (h<1) h = 1;
    this.largeur = l;
    this.hauteur = h;
        }
    }
```

```
class ProgrammeTest {
   public static void main(String[] args) {
      UnRectangle r1;
      r1 = new UnRectangle(5, 6, -7, -8);
      r1.largeur = -7;
      r1.hauteur = -8;
   }
}
```

Quelle est la largeur de r1 ?

Quelle est la hauteur de r1 ?

Encapsulation

- Classe = { ensemble cohérent de données et de services (méthodes) }.
- L'utilisateur d'un objet ne doit voir que les services offerts
  - il n'a pas à connaître la façon dont ils sont implantés,
  - il ne doit pas pouvoir modifier directement l'état d'un objet (sans passer par une méthode) car l'objet ne pourrait plus garantir aucune cohérence.
- Ces principes correspondent à la notion d'**encapsulation**.
- Dans la déclaration des attributs et des méthodes on peut préciser des **droits d'accès** :
  - ***private :*** non accessible/visible en dehors de la classe,
  - ***public* :** accessible par tous,
  - rien : accessible dans le même répertoire/*package*.

- Les choix habituels sont :
  - **Classes = *public***. Car le type objet est en général réutilisable. Attention : une seule classe publique par fichier qui porte nécessairement le nom de cette classe même s'il contient d'autres classes non publiques.
  - **Attributs = *private*.** Car les attributs représentent l'état de l'objet dont la modification doit être contrôlée par l'objet lui-même pour garantir la cohérence. Si des classes externes doivent consulter la valeur de ces attributs on définit des **méthodes publiques d'accès** (*get*). Si des classes externes doivent pouvoir modifier la valeur de ces attributs on définit des **méthodes publiques de modification** (*set*).
  - **Méthodes-services = *public***. Car les méthodes qui correspondent aux services offerts doivent être visibles depuis les autres classes du programme.
  - **Autres méthodes : *private***. Car elles apparaissent suite à la décomposition en plusieurs étapes des algorithmes de certains services et ont donc une utilité limitée à la classe en cours de définition.

```
public class UneClasse {
    private int valeur;
    public int getValeur() {
        return valeur;
    }
    public void setValeur(int new_val) {
        if (new_val >=0) valeur=new_val;
    }
}
```

- Un objet d'une certaine classe ne peut accéder qu'aux propriétés **publiques** d'un objet d'une autre classe. En revanche, deux objets d'une même classe ont accès à toutes leurs propriétés, publiques ou privées.

Référence *this*

- Toute méthode dispose d'une **variable cachée *this*** qui contient la référence à l'objet qui exécute la méthode. Cette variable est sous-entendue lorsqu'on utilise le nom incomplet d'une méthode.

  Ex :

```
class UneClasse {
    private int val;
    public UneClasse(int v) {
        this.val=v;      // on peut écrire val=v;
    }
    public void addVal(int x) {
        this.val+=x;     // on peut écrire val+=x;
    }
}
```

Bases de la programmation

Répétitions

Répétition inconditionnelle (*for*)

- Elle s'écrit :

```
for (int i=1;i<=9;i++) {
   System.out.println(i*9);          // table de 9
}
```

- Dans la parenthèse qui suit le *for* on distingue 3 parties séparées par des ';' :
  - *int i=1* : déclaration et initialisation du **compteur de boucle**, représenté par la variable i, dont la portée est limitée à cette instruction for.
  - *i<=9* : **condition** (type *boolean*) que doit remplir le compteur de boucle pour que le corps de la boucle (bloc) soit **exécuté**.
  - *i++* : manière dont **évolue le compteur** (les deux plus symbolisent l'incrémentation de 1); on peut trouver aussi i-- (compteur décroissant).

Ex :
```
for (int i=5;i>=0;i--) {
   System.out.println(i);          // 5,4,3,2,1,0
}
```

- Le *for* permet aussi d'établir peu à peu un résultat de façon récurrente.

  Ex : calcul de la somme des 100 premiers carrés

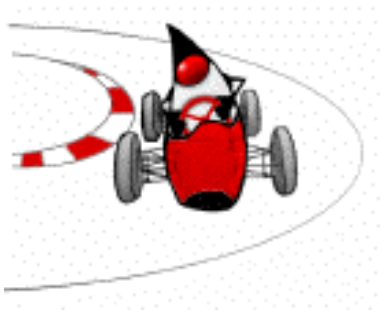
```
int s=1;
for (int i=2; i<=100; i++) {
   s=s+(i*i);
}
```

- Le corps d'une boucle est une suite d'instructions qui peut donc contenir une **boucle** utilisant un compteur différent. A chaque passage dans le corps de la boucle principale la **boucle imbriquée** est exécutée en totalité.

  Ex : afficher un triangle constitué de caractères '*' dont la hauteur est donnée

```
    *
   ***
  *****
 *******
*********
```

```
for (int i=1; i<=h; i++) {
   for (int j=1; j<=h-i; j++) {
      System.out.print(' ');
   }
   for (int j=1; j<=2*i-1; j++) {
      System.out.print('*');
   }
   System.out.println();
}
```



Si h=5, pour i=1 le 1er j varie de 1 à 4 et affiche 4 espaces
le 2e j varie de 1 à 1 et affiche une *
pour i=2 le 1er j varie de 1 à 3 et affiche 3 espaces
le 2e j varie de 1 à 3 et affiche trois *
pour i=3 le 1er j varie de 1 à 2 et affiche 2 espaces
le 2e j varie de 1 à 5 et affiche cinq *
pour i=4 le 1er j varie de 1 à 1 et affiche 1 espace
le 2e j varie de 1 à 7 et affiche sept *
pour i=5 le 1er j varie de 1 à 0 et affiche 0 espace
le 2e j varie de 1 à 9 et affiche neuf *

Répétition conditionnelle

- Le *for* implique de connaître à l'avance le nombre d'itérations (valeur finale du compteur).
- On peut souhaiter répéter une instruction **tant qu'une certaine condition est remplie** alors qu'il est impossible de savoir combien d'itérations seront nécessaires.
- C'est le but de l'instruction *while* :

  while (condition) instruction;

  où condition est une expression logique (type *boolean*).

Ex :

```
System.out.println("indiquez la largeur du rectangle");
Scanner sc=new Scanner(System.in);
int l=sc.nextInt();
while (l<1) {
   System.out.println("erreur : indiquez une valeur > 0");
   System.out.println("indiquez la largeur du rectangle");
   l=sc.nextInt();
}
```

Bases de la programmation

Tableaux

Déclaration et création

- Un tableau permet de désigner une suite finie d'éléments de même type (*int, String*, références à des objets quelconques…) au moyen d'une unique variable.
- Les tableaux sont des objets munis de propriétés, mais d'utilisation et de syntaxe particulière :
  - déclaration d'une variable de référence sur un tableau d'entiers : ***int [ ] tab_int;***
  - déclaration d'une variable de référence sur un tableau de rectangles : ***UnRectangle[ ] tab_rect;***
- Création d'un tableau : opérateur *new* spécifique
  - création d'un tableau de 10 entiers : ***tab_int=new int[10];***
  - création d'un tableau de 15 rectangles via une expression

    int n=10;

    tab_rect=new UnRectangle[n+5];

Accès aux éléments

- Ce n'est pas un appel de méthode : on a une écriture simplifiée où l'indice (expression) est indiqué entre crochets après la référence au tableau.

  Attention : le premier élément a pour indice 0. Donc pour accéder au septième élément du tableau d'entier créé ci-dessus, on écrit : ***tab_int[6]***
- Un élément de tableau peut être utilisé comme n'importe quelle variable.

  Ex:
  ```
  double[ ] ref_tab;
  ref_tab=new double[10];
  ref_tab[0]=4.5;
  double d;
  d=2*ref_tab[0];
  ```
- Chaque tableau a un attribut ***length*** de type *int* qui donne sa taille.

Parcours d'un tableau et initialisation

- Les boucles *for* sont le moyen classique de parcours d'un tableau.

  Ex: création d'un tableau avec les 20 premiers multiples de 19

```
int[] ref_tab;
ref_tab=new int[20];
for (int i=0;i<ref_tab.length;i++) {
    ref_tab[i]=(i+1)*19;
}
```

- A la création d'un tableau les éléments sont initialisés comme suit : *boolean* → *false*, *int* → 0, *double* → 0.0, *char* → *'\u0000'* (caractère unicode de code 0), objet → *null*.
- Quand on crée un tableau de rectangles on a donc un tableau de références valant *null*. Il faut ensuite fabriquer les objets rectangles :

```
UnRectangle[ ] ref_tab;
ref_tab=new UnRectangle[15];
for (int i=0;i<ref_tab.length;i++) {
    ref_tab[i]=new UnRectangle();
}
```

- Initialisation immédiate *:* on peut donner le contenu d'un tableau en une seule étape, à condition de le faire dès sa déclaration avec des **{ }.**

Ex : *int[ ] test={0, 1};*

```
char[ ] test2={'a', 'b', 'c'};
StringBuffer[ ] test3={null, new StringBuffer(),
                       new StringBuffer("ok")};
```

Tableaux et références (paramètres)

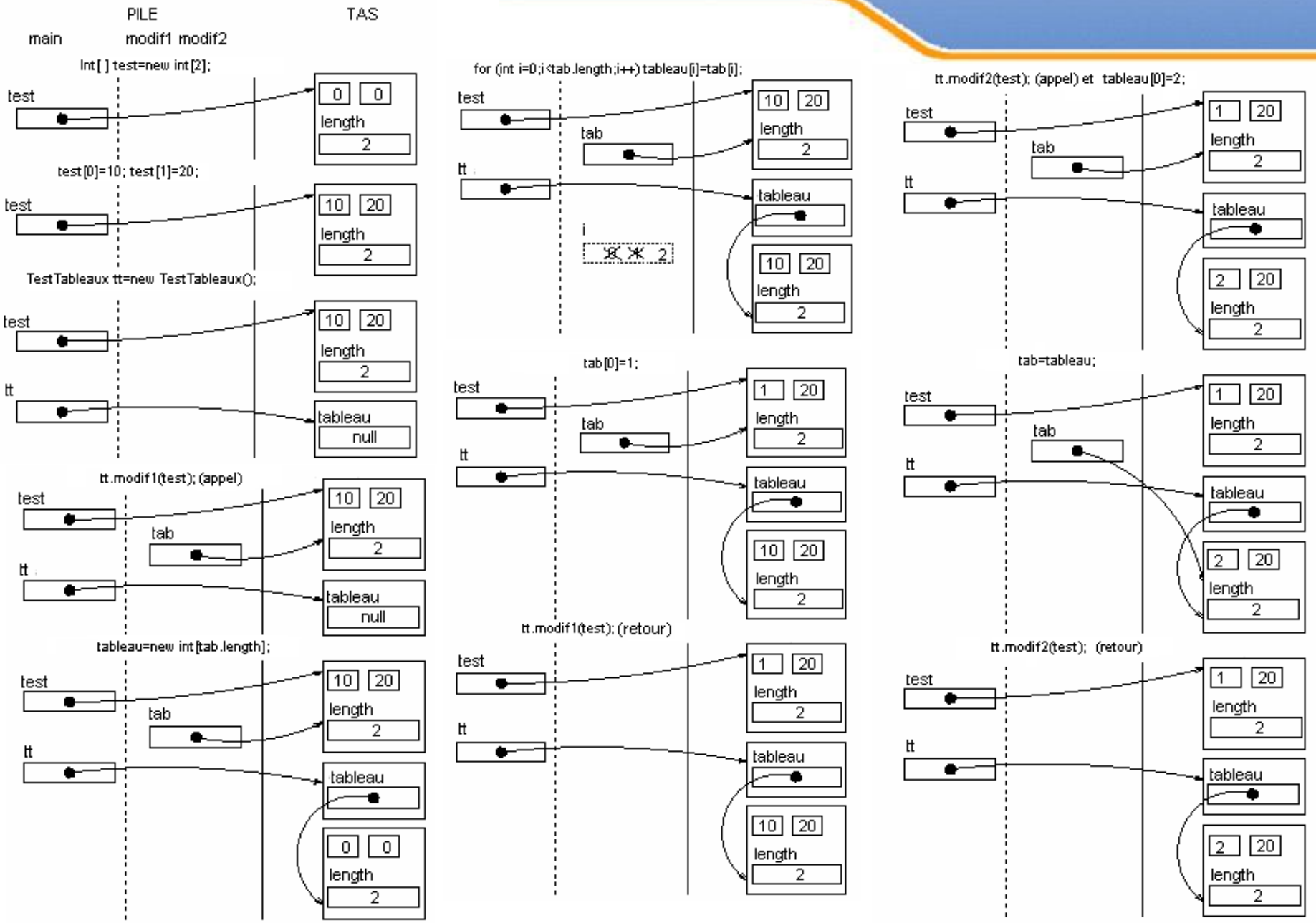
- Les tableaux sont des objets. Manipuler des variables désignant des tableaux revient donc à manipuler leurs références.
- Tableaux en paramètres : si un paramètre formel est de type tableau, le paramètre effectif qui lui est affecté est une référence qui permet donc d'accéder mais aussi de **modifier dans la méthode appelante** les éléments de l'objet tableau.

```
public class TestTableaux {
  private int[ ] tableau;

public void modif1(int[ ] tab) {
     this.tableau=new int[tab.length];
     for (int i=0;i<tab.length;i++) this.tableau[i]=tab[i];
     tab[0]=1;
  }
  public void modif2(int[ ] tab) {
     this.tableau[0]=2;
     tab=this.tableau;
  }
}
public class ProgrammeTest {
   public static void main(String[ ] args) {
       Int[ ] test=new int[2];
       test[0]=10;
       test[1]=20;
       TestTableaux tt=new TestTableaux();
       tt.modif1(test);
       tt.modif2(test);
   }
}
```

test[0] vaut 1
test[0] pas modifié

?


PILE
TAS
main
modif1 modif2
Int [ ] test=new int[2];
test
0 0
length
2
test[0]=10; test[1]=20;
10 20
TestTableaux tt=new TestTableaux();
tt
tableau
null
tt.modif1(test); (appel)
tab
tableau=new int[tab.length];
for (int i=0;i<tab.length;i++) tableau[i]=tab[i];
i
2
tab[0]=1;
tt.modif1(test); (retour)
tt.modif2(test); (appel) et tableau[0]=2;
tab=tableau;
tt.modif2(test); (retour)

Tableaux et références (résultats)

- Le type de retour d'une méthode peut être un type tableau. La valeur retournée est une référence à un objet tableau. Ex :

```
import java.util.Scanner;
public class TestTableaux {
   private int[ ] tableau;
   public TestTableaux(int t) {
      this.tableau=new int[t];
      Scanner sc=new Scanner(System.in);
      for (int i=0; i<t; i++) this.tableau[i]=sc.nextInt( );
   }
   public int[ ] creeTableauTriples() {
      int[ ] tab=new int[this.tableau.length];
      for (int i=0; i<tab.length; i++) tab[i]=3*this.tableau[i];
      return tab;
   }
}
public class ProgrammeTest {
   public static void main(String[ ] args) {
      TestTableaux tt=new TestTableaux(6);
      int[ ] tab_triples=tt.creeTableauTriples( );
   }
}
```

- Un tableau est un objet modifiable donc sujet à des **effets de bord**.

  Ex :

  ```
  int[ ] tab1, tab2;
  tab1=new int[2];
  tab1[0]=0;
  tab1[1]=1;
  tab2=tab1;
  tab2[0]=-1; // modifie aussi tab1[0]
  ```

- Pour les éviter il est parfois indispensable de faire des **copies de tableaux**. Ecrire une méthode qui construit une copie d'un tableau exige de connaître le type de ses éléments.

  Ex : copie d'un tableau d'entiers

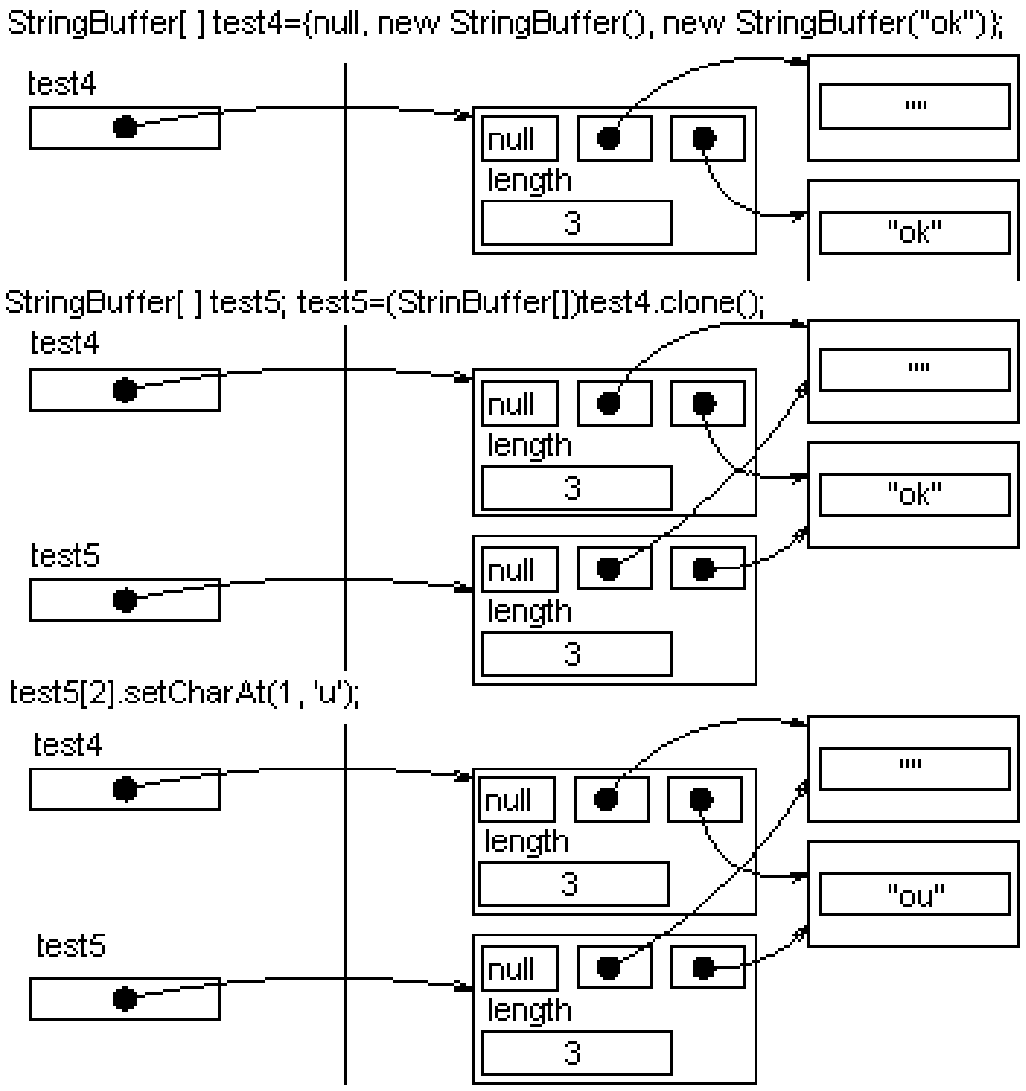
```
public class CopieurTableaux {
   public int[ ] copieTabInt(int[ ] tab) {
      int[ ] tab_copie=new int[tab.length];
      for (int i=0; i<tab.length; i++)
         tab_copie[i]=tab[i];
     return tab_copie;
  }
  public double[ ] copieTabDouble(double[ ] tab) {
     double[ ] tab_copie=new double[tab.length];
     for (int i=0; i<tab.length; i++)
        tab_copie[i]=tab[i];
     return tab_copie;
   }
   // ...
}
```

- Les tableaux sont des objets ‘clonables’, ce qui évite de devoir passer par de telles méthodes spécifiques à chaque type.

  int[] tab_copie=(int[]) tab.clone();

  On expliquera plus tard pourquoi l'indication du type objet tableau int[ ] entre parenthèses est obligatoire.
- Attention! Le clonage d'un tableau d'objets ne fait que **recopier les références** à ces objets, pas les objets eux-mêmes. Les effets de bord restent donc possibles. Ex :

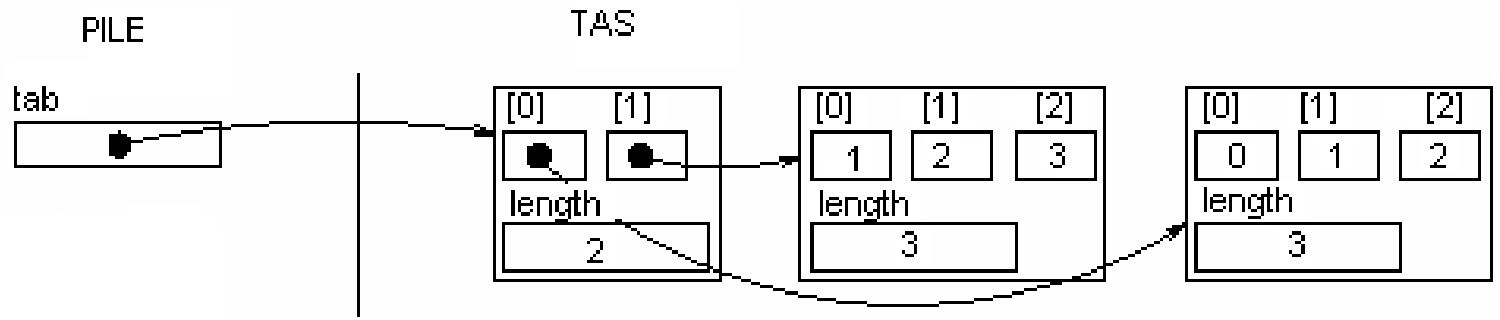
```
StringBuffer[ ] test4={null, new StringBuffer(), new StringBuffer("ok")};
StringBuffer[ ] test5;
test5=(StringBuffer[ ])test4.clone( );
test5[2].setCharAt(1, 'u');  // la modification d’un élément de test5
                             // se répercute sur test4
System.out.println(test4[2]); // affiche "ou"
```


StringBuffer[ ] test4={null, new StringBuffer(), new StringBuffer("ok")};
test4
null
length
3
""
"ok"
StringBuffer[ ] test5; test5=(StrinBuffer[])test4.clone();
test4
null
length
3
""
"ok"
test5
null
length
3
test5[2].setCharAt(1 , 'u');
test4
null
length
3
""
"ou"
test5
null
length
3

Tableaux multidimensionnels

- Les tableaux étant des objets et pouvant contenir des objets on peut définir **des tableaux de tableaux, des tableaux de tableaux de tableaux** ... On obtient ainsi des tableaux multidimensionnels.
- Création :

```
int[][] tab=new int[2][3];
for (int i=0; i<2; i++)
   for (int j=0; j<3; j++)
      tab[i][j]=i+j;
```



- Cas général :

type[ ][ ]...[ ] identificateur;

identificateur = new type[dim1][dim2]...[dimD];

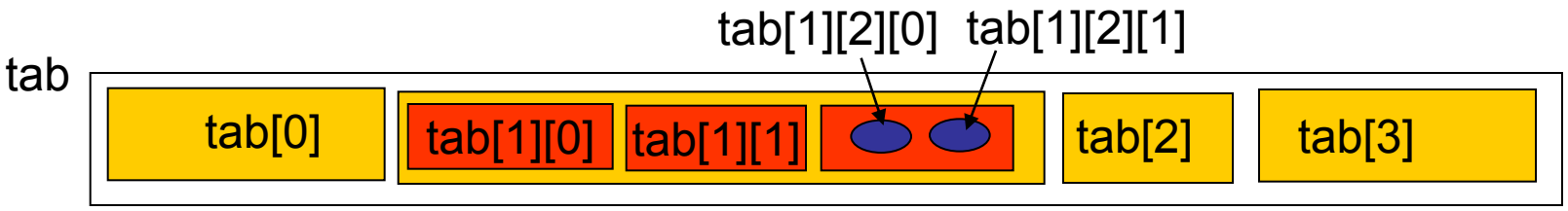
identificateur[indice1][indice2]...[indiceD] = expression;

int taille = identificateur.length;

int taille i = identificateur[i].length;

int taille i j = identificateur[i][j].length; ...

Ex :
```
for (int i = 0; i< tab.length; i++)
    for (int j = 0; j<tab[i].length; j++)
        for (int k = 0; k<tab[i][j].length; k++)
            tab[i][j][k] = 0;
```



tab.length = **4** tab[1].length = **3** tab[1][2].length = **2**

- On peut construire des tableaux irréguliers où chaque tableau élément du tableau de tableaux a sa propre taille.

  Ex :

```
int[ ][ ] tab=new int[2][ ];
tab[0]=new int[2];
tab[1]=new int[4];
int[ ][ ] autre={{0,1},{0,1,2,3}};
```

- Les effets de bord sont possibles.

  Ex :

```
int[ ][ ] tab={{0,1},{2,3}};

int[ ] autre=tab[0];
autre[0]=autre[1];     // tab={{1,1},{2,3}}
```

- Pour cloner un tableau bidimensionnel on ne peut pas se contenter de l'appel de la méthode *clone()* (on crée juste une copie du tableau des références de chaque ligne). Pour effectuer un vrai clonage bidimensionnel il faut faire une boucle qui clone chaque ligne contenue dans le tableau.