Types référence et gestion de la mémoire

Virginia Aponte

CNAM-Paris

3 mars 2016

1. Etudier la répresentation en mémoire

Rappels : types des données en Java

- primitifs : une donnée simple (entier, caractère, etc) ;
- non primitifs ou composite : plusieurs données ensemble
  - **tableaux** : plusieurs données (même type) ;
  - **String** : chaînes de caractères ;
  - **objets** : plusieurs données (types $\neq$).

Rappels : variables, exécution et mémoire

- exécution programme $\Rightarrow$ utilise/modifie valeurs des variables
- variables $\Rightarrow$ valeurs succéssives en mémoire ;
  - selon son type, donnée dans variable :
    - représentée différemment (nombre d'octects, etc) ;
    - mise dans zone particulière
- instruction exécutée $\Rightarrow$ peut changer organisation/contenu mémoire ;

Pourquoi étudier la représentation en mémoire ?

1. pour approfondir notre compréhension des objets ;
2. pour apprendre à mieux structurer/exploiter nos données ;
3. pour mieux comprendre les constructions sophistiquées de Java ;
4. pour écrire des programmes plus robustes (moins d'erreurs).

Pour une exécution, la mémoire comporte

Deux grandes zones :

- Pile (*stack*)
  - stocker *les variables de tous les sous-programmes* en cours d'exécution (empilement de *contextes d'exécution*) ;
  - quelques informations de contrôle (où retourner après un appel, ou s'il y a une erreur)
- Tas (*heap*)
  - *créer dynamiquement* nouvelles données non primitives ;
  - leur taille n'est pas nécessairement connue à la compilation (ex : tableau)

2. Contexte d’exécution d’une méthode

Rappel : méthodes et leurs variables

```
static int plusUn(int x){
   int res = x+1;
   return res;
}
```

Une méthode possède des données :

- paramètres ⇒ valeurs prises à l'appel ;
- variables locales ⇒ modifiées par exécution de la méthode ;
- accès éventuel à `this`

Chaque méthode utilise une mémoire locale d'exécution pour stocker/modifier ses données. ⇒ c'est le contexte d'exécution de la méthode.

Exemple : contexte d'exécution pour méthode statique

```
static int plusUn(int x){
  int res = x+1;
  return res;
}
```

⇒ Le contexte d'exécution de `plusUn` contient 2 emplacements.

(*) ce contexte comporte un emplacement supplémentaire pour la valeur de retour de la fonction. Nous n'en parlerons pas.

Le contexte d'exécution d'une méthode

Mémoire locale à chaque méthode avec emplacements pour :

- ses variables locales + paramètres
- la variable `this`, si méthode d'instance

utilisée uniquement pendant 1 exécution.

Sert pour 1 exécution de la méthode :

1. il y a nouveau contexte par appel ;
2. mis en place dans la pile, au moment de l'invocation,
3. on y enregistre valeurs paramètres de l'appel ;
4. disparaît après exécution.

Exemple : contexte d’exécution pour un appel

```
static int plusUn(int x){
  int res = x+1;
  return res;
}
```

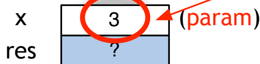
On veut exécuter l’appel `plusUn(3);`

- on mettra (dans la pile, tout en haut) le contexte suivant,
- on y copie la valeur passée en paramètre ($3 \mapsto x$)

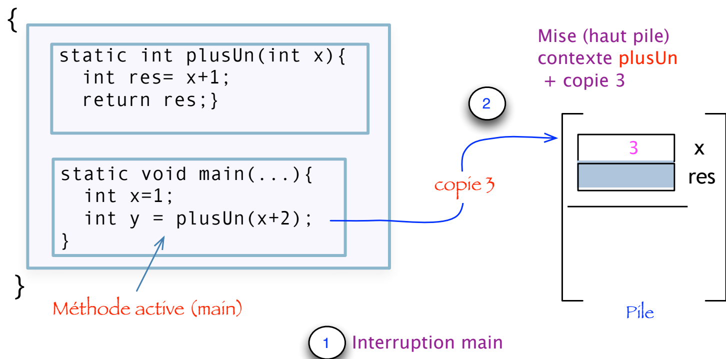
Contexte exécution appel plusUn (3)

valeur passée en paramètre

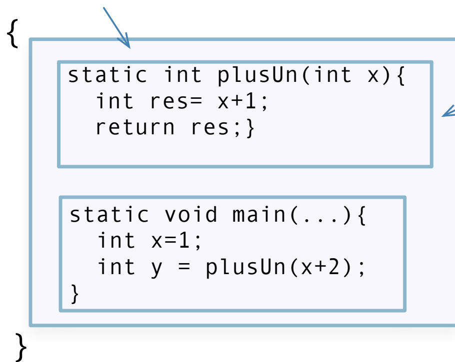
Ce contexte sera mis dans la pile pour exécuter l'appel



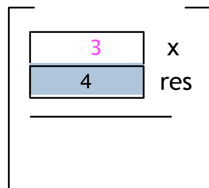
Mise en place contexte pour exécuter `plusUn(3)`



Exécution `plusUn(3)` dans son contexte



3 exécution corps



Pile

3. La pile d'exécution

Rappel : appels imbriqués, variables locales

- une méthode P1 peut invoquer P2, qui peut invoquer P3, etc.
- les variables de chaque méthode sont locales :
  - inaccessibles pour les autres ;
- une méthode P1 transmet des données à une autre :
  - $\Rightarrow$ `P2(x)` : P1 transmet « sa » valeur x vers P2 ;
  - $\Leftarrow$ `return ...`.
    transmet résultat vers son appelant ;

Pile d'exécution

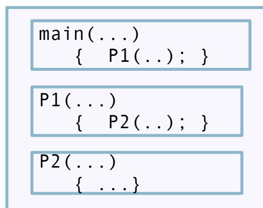
Pile d'exécution (*stack*)

Zone mémoire organisée comme un empilement de contextes d'exécution :

- contient les contextes de toutes les méthodes appelées et non encore terminées ;
- Haut de la pile : contexte de la méthode active (qui s'exécute actuellement).

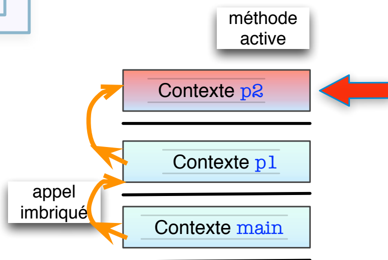
Fonctionnement pile d'exécution (2)

- Haut de la pile : contexte de la méthode active (qui s'exécute) ;
- chaque appel à P_i, met en place son contexte en haut de la pile ;
- juste au dessous : contexte de la méthode appelante $P_{(i-1)}$;
- dès que P_i termine, son contexte sort de la pile.
- Se retrouve en haut de la pile $\Rightarrow$ contexte méthode appelante $P_{(i-1)}$, qui devient active.



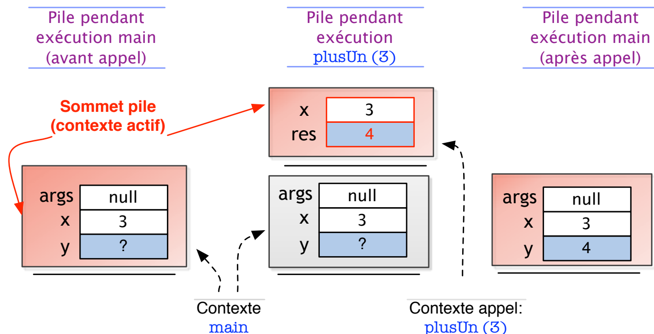
Code exécuté:

main →
p1(..) →
p2(..) → exécution p2



Exemple : pile pour exécution de plusUn(3)

```
static int plusUn(int x){
   int res = ....
}
public static void main (String [] args){
   int x = 3;
   int y = plusUn(x); // <--- appel
```



Demo 1 : contexte + pile d’exécution, appels imbriquées (méthodes statiques)

4. Répresentation des données

Rappel : données et leur taille mémoire

- données de type primitif
  - données élémentaires (nombre, booléan, caractère) ;
  - occupent taille mémoire fixe. Ex : `int x;` requiert toujours 32 bits en mémoire.
- données non primitifs :
  - plusieurs données ensemble (tableaux, String, objets) ;
  - taille **non nécessairement connue** à la compilation :

```
int[] t = new int [Terminal.lireInt()];
String s = Terminal.lireString();
```

Réprésentation des données en mémoire

Soit `x` déclarée de type `T` et initialisée correctement :

```
T x = valeur ;
```

- (quelque part en mémoire) un espace associé à `x` est destinée à contenir « sa donnée » :
    1. quelle est la taille de cet espace ?
    2. que contient-il ?

Les réponses à 1 et 2 dépendent du type `T` déclaré pour `x`.

Représentation mémoire des types primitifs

Si `x` est déclarée de type primitif :

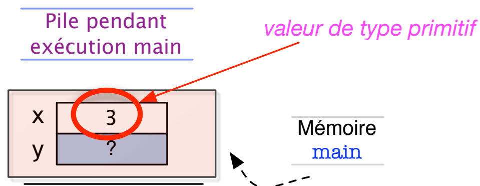
```
int x = 5;
```

- (quelque part) un emplacement mémoire est associé à `x` :
  1. la taille de cet emplacement ? Fixe, ici 32 bits.
  2. que contient-il ? 5, encodé sur 32 bits.

Les données primitives (int, char, double, etc.) sont représentées en mémoire dans un espace de taille fixe, qui dépend du type de la donnée. La variable primitive contient sa donnée.

Exemple 1 : variables type primitif

```
public static void main (String [] args){
   int x = 3;
   int y = x+2;
```



Représentation mémoire des types non primitifs

```
int[] x = new int [5];
```

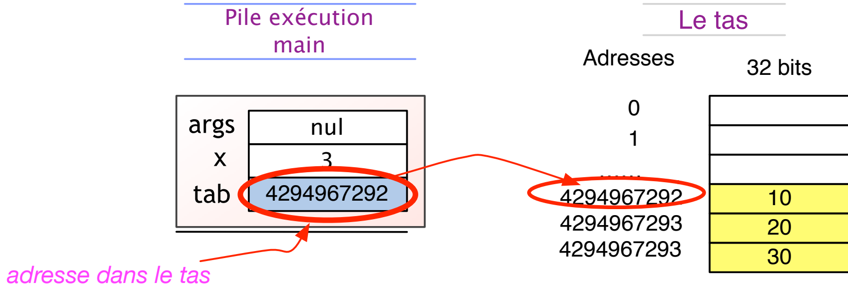
- (quelque part) un emplacement mémoire est associé à `x` :
  1. la taille de cet emplacement ? 32 ou 64 bits.
  2. que contient-il ? une adresse mémoire du Tas
     - à cette adresse se trouve un espace d'au moins 5x32 bits pour stocker les 5 entiers du tableau, initialisés à 0.

Une valeur non primitive est représenté par une adresse mémoire dans le tas. À cette adresse sont stockées les données de la variable. La variable non primitive ne contient pas directement sa donnée.

Exemple : variable non primitive

variable non primitive = (contient) adresse (du tas) vers ses composantes.

```
public static void main (String [] args){
   int x = 3;
   int [] tab  = {10, 20, 30};
```



Types non primitifs = types référence

Données non primitives ⇒ représentées de manière indirecte :

- mises dans le tas (et non dans la pile) ;
- les variables non primitives *ne contiennent pas leurs données*,
  - mais plutôt l'adresse dans le tas où elles se trouvent ;

Pointeurs, références

Les variables contenant une adresse (vers leurs données) sont appelées pointeurs ou références.

En Java, on parle du type *référence*.

Exemples de données de types référence

- Une variable de type String, ne contient pas la chaîne elle-même, mais l'adresse mémoire où se trouve la chaîne.
- La variable `int [] t = {4, 6, 3}` ne contient pas le tableau, mais l'adresse où se trouve le tableau.
- Une variable de type `Compte` ne contient pas l'objet, mais l'adresse où celui-ci se trouve.

Chacune de ces variables est un *pointeur ou référence*.

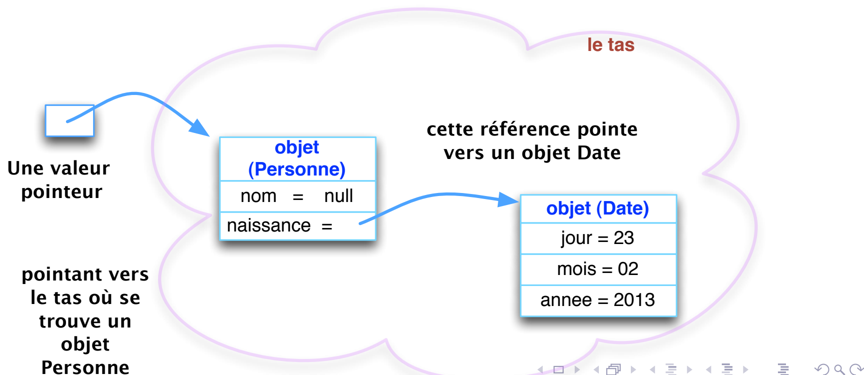
Bilan contenu des variables

La valeur contenue dans une variable est soit

- primitive : un entier, caractère, etc ;
- adresse (ou référence) vers un emplacement dans le tas.

Dessiner les pointeurs

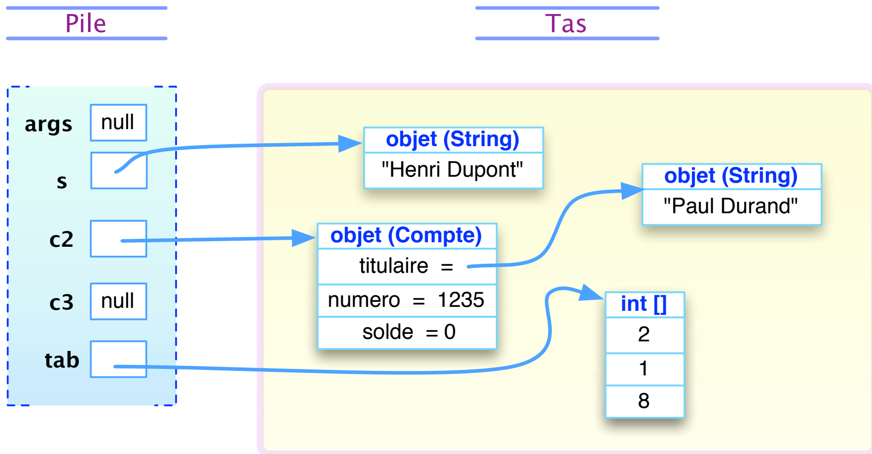
- flèche vers emplacement où trouver les valeurs dans le tas ;
- *début* de la flèche = adresse de cet emplacement,
- *fin* de la flèche = début de l'emplacement dans le tas.



Exemples de données de types référence

```
public static void main(String[] args){
   String s= "Henri␣Dupont";
   int [] tab = {2,1,8};
   Compte  c2, c3;
   c2 = new Compte("Paul␣Durand",  1235, 0);
}
```

Exemples de données de types référence



4. Création (dans le tas) de données de type référence

Création de valeurs de type référence

```
String s= "Henri␣Dupont";
int [] tab = new int [3];
Compte  c2 = new Compte("Paul",1235, 0);
```

Création de valeurs référence

Se fait (sauf parfois pour les types prédéfinis) via la syntaxe :

new *Nom-type-ou-constructeur*

1. **réserve** dans le tas la place nécessaire pour toutes les cases de tableau ou variables d'instance. Leur donne des valeurs initiales ;
2. **retourne** le numéro de l'adresse réservée.

Exemple : création d'un tableau

```
static void main(...){
  int[] t = new int[3];
}
```

1

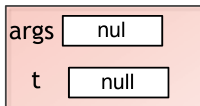
création +
initialisation

adr1

int[]
0
0
0

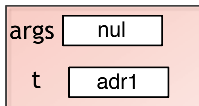
Tas

Contexte main



Pile
(avant affectation)

Contexte main



Pile
(après affectation)

2

affectation
adresse
stockage

Création d'objets

```
Compte c1 = new Compte();
```

1. `Compte c1` : **Déclaration variable c1**
   (quelque part) ⇒ réserver espace + initialiser `null`.

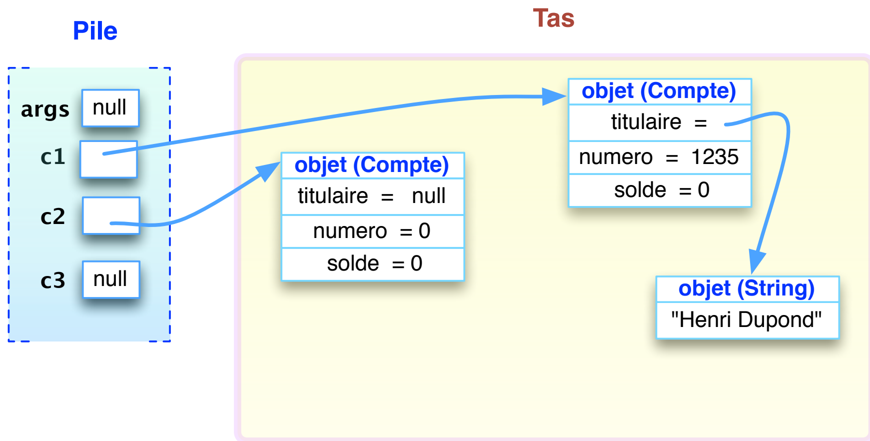
2. `new Compte()` : **Création objet**
   (a) (Tas) ⇒ réserver espace pour stocker objet instance de Compte (variables instance, méthodes) ;
   (b) Initialiser variables d'instance.

3. `Compte c1 = new Compte()` : **Copier**
   adresse de l'objet (dans le tas) sur le contenu de `c1` dans la pile.

Exemple

```
public static void main(String[] args){
  Compte c1, c2, c3;
  c1 = new Compte();
  c2 = new Compte();

  c1.titulaire =  "Henri␣Dupond";
  c1.numero =  1235;
}
```


Pile
Tas
args
null
c1
c2
c3
null
objet (Compte)
titulaire = null
numero = 0
solde = 0
objet (Compte)
titulaire =
numero = 1235
solde = 0
objet (String)
"Henri Dupond"

5. Affecter, comparer des références

Affectation entre variables de type référence

```
int [] t1, t2;
t1 = {1,2};
t2 = {10,2, 9, 7};
t1 = t2;    // <--- Affectation
```

Affectation entre variables référence

Seulement si leurs types sont compatibles.

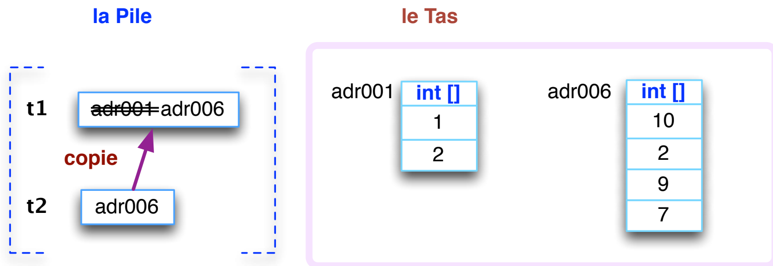
Comportement :

- copie du contenu d'une variable vers l'autre ;
- ce contenu est **une adresse**.

⇒ les deux variables contiennent la même adresse.

Dessin affectation références (adresses explicites)

```
int [] t1, t2;
t1 = {1,2};
t2 = {10,2, 9, 7};
t1 = t2;
```

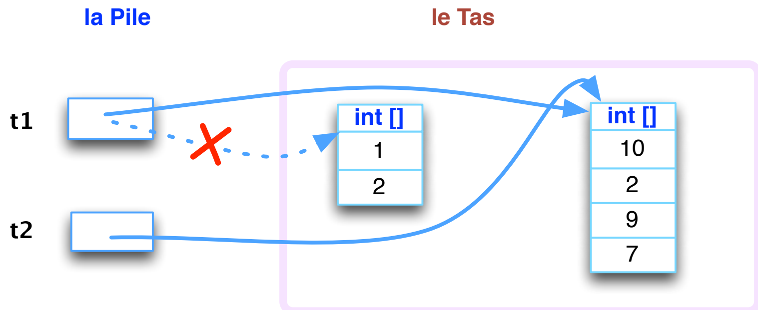


Affectation t1 = t2

copier adr006 dans t1

Le même avec flèches

```
int [] t1, t2;
t1 = {1,2};
t2 = {10,2, 9, 7};
t1 = t2;
```



Après affectation `t1`, `t2` contiennent la même adresse

Partage de données

```
int [] t1, t2;
t1 = {1,2};
t2 = {10,2, 9, 7};
t1 = t2;
t1[0] = 50;
Terminal.ecrireInt(t2[0]);
```

- On copie le contenu d'une variable dans l'autre. Ce contenu est **une adresse**.
  $\Rightarrow$ t1 et t2 contiennent la **même adresse**.
- Elles pointent vers le même emplacement physique de la mémoire.
  $\Rightarrow$ tout changement dans l'une modifie ce qui est pointé par l'autre.

On dit de t1 et t2 qu'elles partagent la même donnée en mémoire.

Affectation entre variables de type référence

```
int [] t1 = {1,2};
int [] t1 = {10,2, 9, 7};
t1 = t2;
t1[0] = 50;
Terminal.ecrireInt(t2[0]);
```

Ici `Terminal.ecrireInt(t2[0])` $\Rightarrow$ affiche 50

Affectation entre variables objet

Le même phénomène de partage sur les objets :

```
Compte c1 = new Compte();
Compte c2 = new Compte();
c1.solde =100; c1.numero=218;
c1.titulaire="Dupont";
c2 = c1;
c2.solde = 60;
Terminal.ecrireStringln
  ("c1="+c1.solde + ","+ c1.titulaire +","+c1.numero);
Terminal.ecrireStringln
  ("c2="+c2.solde+","+c2.titulaire+","+c2.numero);
```

```
> java testBis
c1 = 60 , Dupont ,218
c2 = 60 , Dupont ,218
```

Egalité des types référence

L'opérateur == compare les bits contenus dans les variables.

S'il s'agit d'adresses, cela teste si les adresses sont égales, c.a.d. si les variables référencent le même objet en mémoire.

```
int [] t1 =  {1,2};
int [] t2 =  {10,2, 9, 7};
int [] t3 =  {1,2};
t2 = t1;
if (t1==t2){ Terminal.ecrireStringln("t1==t2"); }
if (t1==t3){ Terminal.ecrireStringln("t1==t3"); }
else {Terminal.ecrireStringln("t1!=t3");  }
```

Affichages :

```
t1==t2
t1!=t3
```

Egalité des types référence

Qu'affiche ce programme ?

```
Date d1 = new Date(1,1,2000);
Date d2 = d1;
Date d3 = new Date(1,1,2000);
if (d1==d2){ Terminal.ecrireStringln("d1==d2");
} else {
     Terminal.ecrireStringln("d1!=d2");   }
if (d1==d3){
     Terminal.ecrireStringln("d1==d3");
} else {
     Terminal.ecrireStringln("d1!=d3");   }
```

Egalité des types référence

L'exécution de ce programme produit :

```
> java Chap12d
d1==d2
d1!=d3
```

Comparer tableaux, Strings, objets

- Tableaux et Strings sont des types référence : **ce sont des objets**.
- L'opérateur == utilisé pour les comparer, **compare leurs adresses**, autrement dit, cela teste s'il s'agit du même objet en mémoire.
- Ce n'est pas la bonne méthode si l'on veut comparer **leur contenu**, c.a.d, si leurs valeurs internes sont identiques.

On doit donc utiliser ou écrire des méthodes qui comparent une à une chacune de leurs composantes internes.

6. Références dans références

Références dans références

Il s'agit de données de types référence, avec composantes de type références. Par exemple :

- Tableaux d'objets.
- Objets avec composantes objet.
- Objets avec composantes tableaux, etc.

Exemple 1 : Tableaux d'objets

- Tableau d'objets = tableau de pointeurs ;
- On doit donc créer :
  - le tableau lui-même, d'une certaine taille ;
  - un objet à affecter au contenu de chaque case.

Suite exemple 1 : tableau de comptes

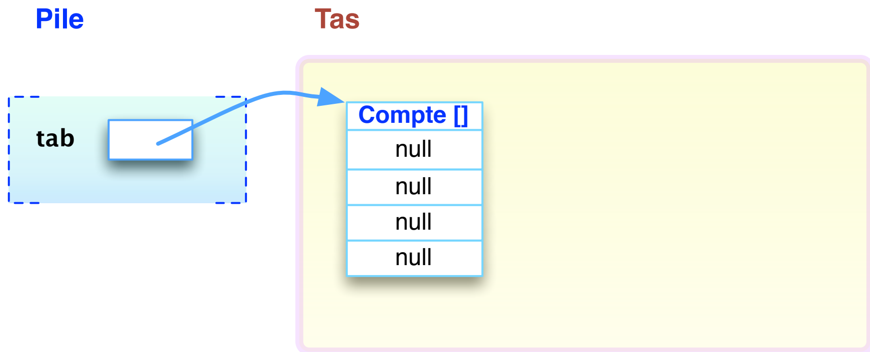
Création d'un tableau avec :

- 4 cases de type Compte ;
- un objet de type Compte par case, de numéro 1235+i, et avec solde 0.

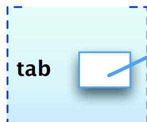
```
Compte [] tab = new Compte [4];
for (int i=0; i<4; i++) {
  tab[i] = new Compte(1235+i, 0);
}
```

Le tableau de comptes (après création)

Le tableau après création, et avant entrée dans la boucle :



Le tableau de comptes (après la boucle)



Exemple 2 : objets dans les objets

```
class Entier{
    public int val;
    Entier(int x){ val = x; }
}
class EntBool{
    Entier ent;
    boolean bo;
    EntBool(Entier e, boolean b){
    ent = e; bo = b;
}
```

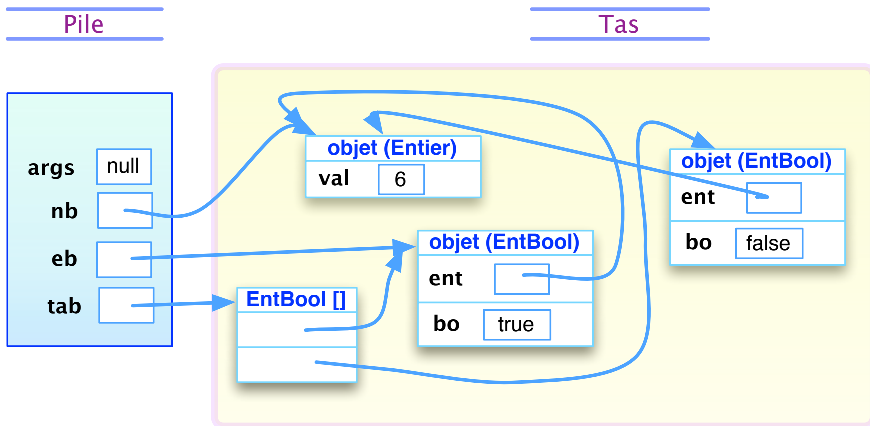
Suite exemple 2

```
public static void main(String[] args){
    Entier nb = new Entier(6);
    EntBool eb = new EntBool(nb,true);
    EntBool[] tab = new EntBool[2];
    tab[0]= eb;
    tab[1] = new EntBool(nb,false);
    nb.val = 7;
    eb.ent.val = 8;       // chemin d'acces
    tab[0].ent.val = 9; // chemin d'acces
}
```

Après exécution :

- Contexte ⇒ 3 variables (nb, eb, tab) ;
- Tas ⇒ 4 espaces alloués (4 créations).

Exemple 2 : création en mémoire (dessin)



Chemin d'accès à une case mémoire

Chemin accès pour une valeur pointée

Chemin à parcourir depuis un pointeur jusqu'à la case contenant une valeur pointée, et qui peut réquerir de traverser plusieurs « flèches ».

2 notations pour suivre une flèche :

- si elle arrive sur un objet ⇒ point + nom de la case ;
- si la flèche arrive sur un tableau ⇒ crochets + numéro case du tableau

Exemple : 4 chemins d'accès différents jusqu'au 6 de l'objet `Entier`

- `nb.val`
- `eb.ent.val`
- `tab[0].ent.val`
- `tab[1].ent.val`

Demos 2 et 3 : objets, tableaux, références dans références

7. Passage de paramètres avec références

Rappels : le passage de paramètres

Le passage de paramètres en Java se fait « par valeur » :

⇒ lors d'un appel `m(x)`, on passe à m :
la valeur contenue dans la variable x.

- x de type primitif : on passe sa valeur, entier, booléan, etc.
- x de type référence : on passe sa valeur, qui est une adresse.

Exemple 1 : passer en argument un tableau

```
static void m(int [] t){
  t[1] = 53;
}
public static void main(String [] args){
    int [] tab = {1,2,3};
    m(tab);
    for (int i=0; i< tab.length; i++){
     Terminal.ecrireString(tab[i] + "␣");
    }
}
```

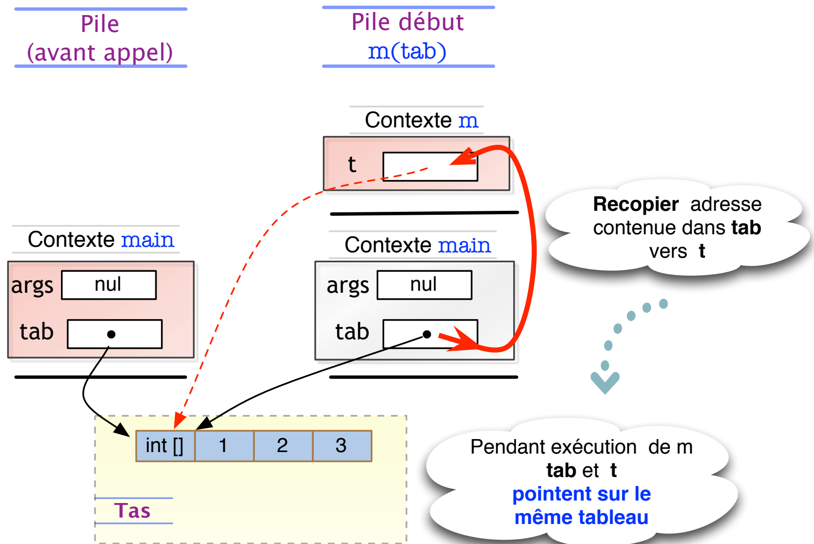
Qu'affiche ce programme ?

Exemple 1 (2)

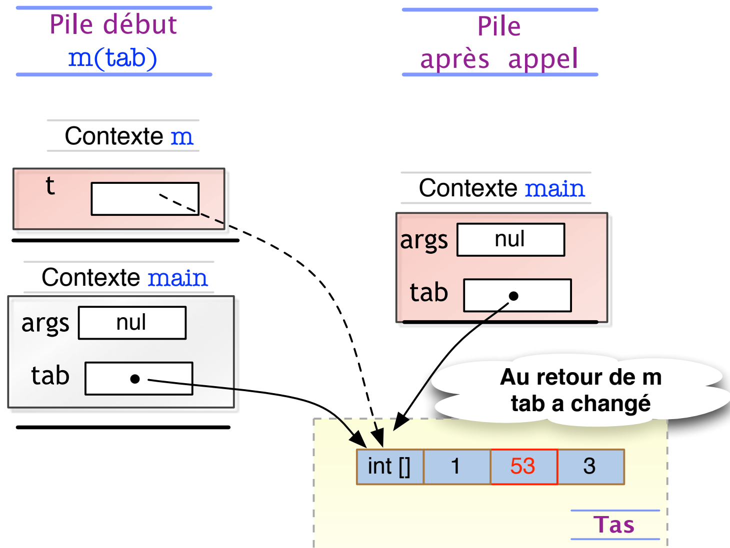
```
static void m(int [] t){
  t[1] = 53;
}
public static void main(String [] args){
    int [] tab = {1,2,3};
    m(tab);
    for (int i=0; i< tab.length; i++){
     Terminal.ecrireString(tab[i] + "␣");
    }
}
```

- variable `tab` de `main` est un tableau (adresse) ;
- `main` appelle `m(tab)` ⇒
  - recopie adresse dans `tab` vers paramètre `t` du contexte de `m`,
  - au retour, la valeur de `tab` a-t-elle changé ?

Exemple 1 (3)



Exemple 1 (4)



Exemple 2 : passer en argument un objet

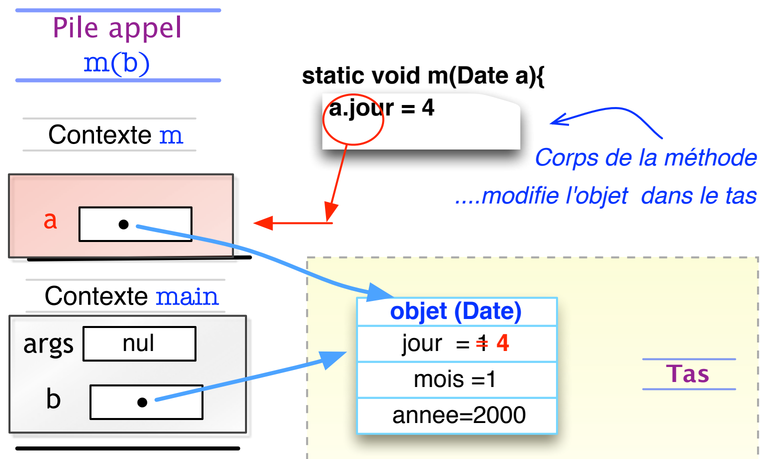
```
static void m(Date a){
    a.jour=4;
}
public static void main(String [] args){
    Date b = new Date(1,1,2000);
    m(b);
```

Exécution de `m(b)` :

1. `a.jour = 4` ⇒ modifie la variable `jour` dans le tas.
2. Retour au main. La variable `a` n'existe plus. Dans le tas, l'objet référencé par `b` **a été modifié**.

`b.afficherDate()` ⇒ affiche `4 / 1 / 2000`

Exemple 2 suite : avec un dessin

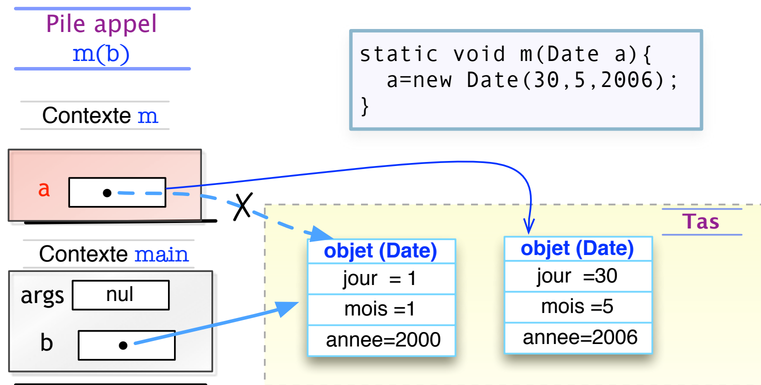


Exemple 2 (3)

```
static void m(Date a){
    a =  new Date(30,5,2006);
}
public static void main(String [] args){
    Date b = new Date(1,1,2000);
    m(b);
    b.afficherDate();
```

Et qu'affiche celui-ci ?

Exemple 2 suite : avec un dessin



la méthode ne modifie pas la référence passée mais un objet pointé localement.

Passage de paramètres

```
static void m(<un-type> p){...
 }

static void main... {
 <un-type> x = ...;
 m(x);    // appel de methode
```

- Lors d'un appel de méthode `m(x)`, on passe à m **la valeur contenue dans la variable** x.
- Cette valeur est recopiée dans la variable `p` du contexte de m. Elle est utilisée pendant l'exécution de m.
- Si m réalise une affectation sur p, cela **n'a aucune incidence** sur la valeur de x, et cela quelque soit le type de x (primitif ou référence).

Demos 4 : appel méthode statique avec références

8. Appel de méthode d'instance

Exécuter un appel de méthode d'instance

Méthodes statiques ou d'instance ⇒ contextes d'exécution ≠.

```
class Compte{
  int solde=0;
  void depot(int x){ this.solde = this.solde + x; }
}
public class ExempleMetRef{
 public static void main(String[] args){
   Compte c1 = new Compte();
   c1.depot(50); // <-- appel methode d'instance
   Terminal.ecrireIntln(c1.solde);
 }
}
```

Appel méthode d'instance (2)

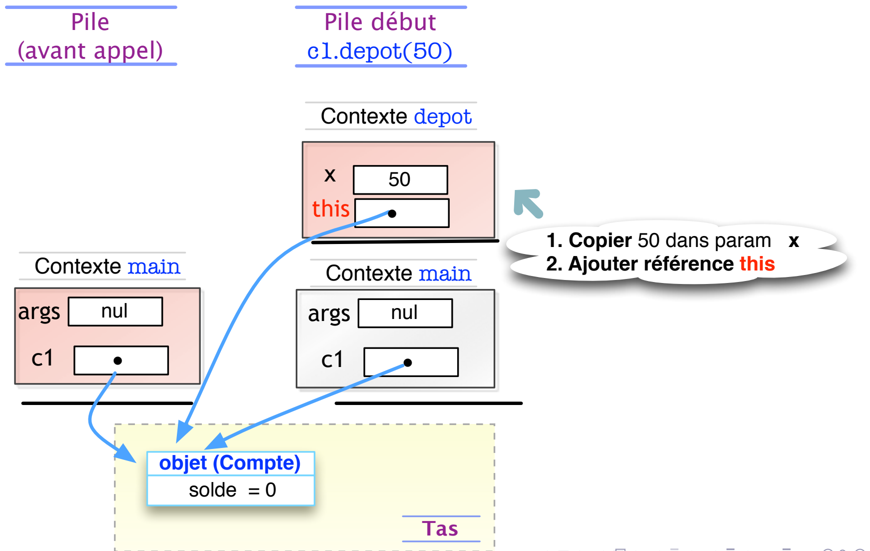
```
Compte c1 = new Compte();
c1.depot(50); // <-- appel methode non statique
```

- on commence par empiler le contexte de la méthode (comme avant) ;
- mais, dans ce contexte *il y a toujours* un emplacement pour la variable `this`, qui référence l'objet sur lequel se fait l'appel.

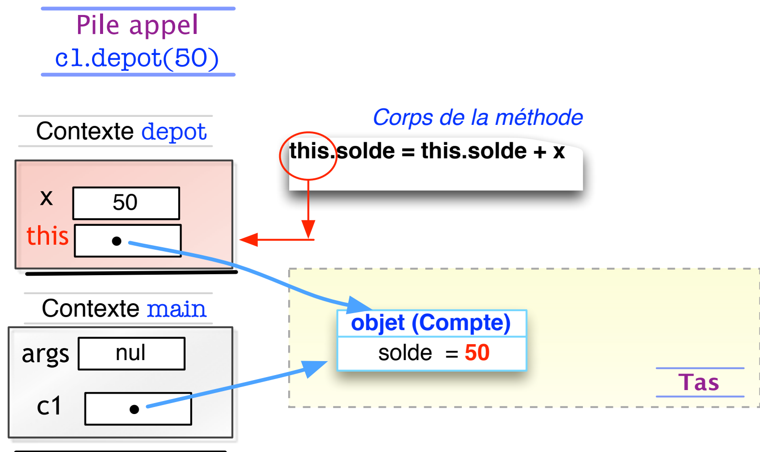
Contexte de méthode non statique

Contient variable `this` pointant vers l'objet sur lequel se fait l'appel.

Appel méthode d'instance (dessin)



Exécution ⇒ suivre pointeur sur this dans le corps de la méthode.



Demo 5 : appel méthode instance sur objet