Types référence et gestion de la mémoire

Virginia Aponte

CNAM-Paris

8 septembre 2020

Rappels : types des données en Java

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

La mémoire d'exécution

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)

1. Répresentation des données

Représentation mémoire des types primitifs

Si `x` est déclarée de type primitif :

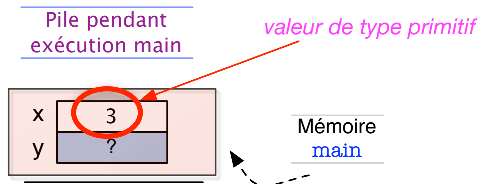
```
int x = 5;
```

- (quelque part) un emplacement mémoire est associé à `x` :
- 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];
```

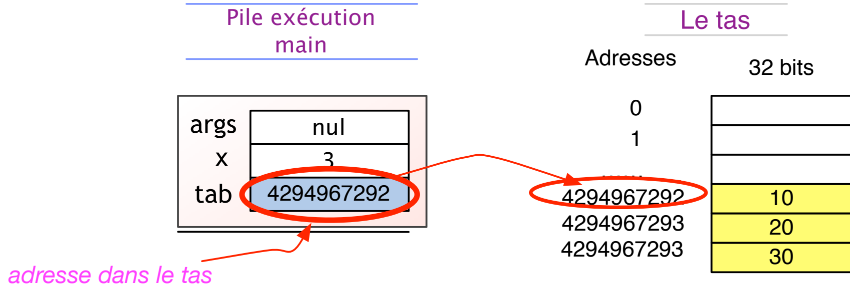
- (quelque part) un emplacement mémoire est associé à `x` :
- que contient-il ? une adresse mémoire du Tas
  - à cette adresse se trouve un espace d'au moins 5x32 bits pour stocker les 5 entiers du tableau, initialisés à 0.

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 = pointeurs = types référence

Données non primitives $\Rightarrow$ représentées de manière indirecte :

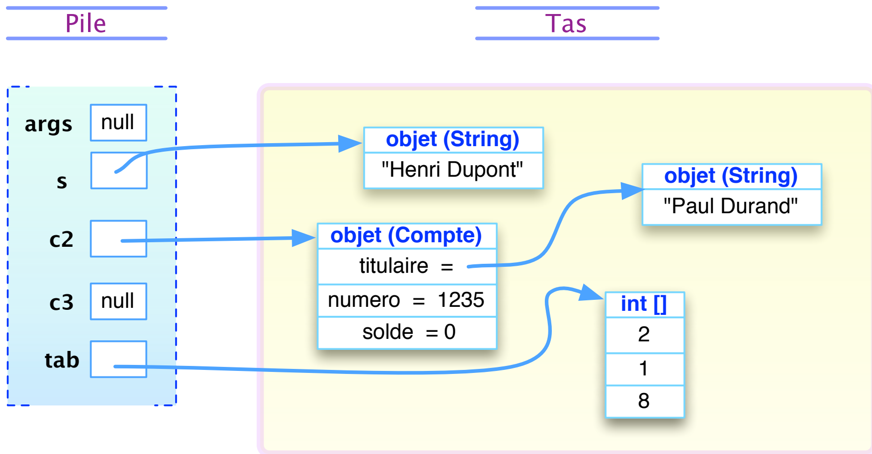
- mises dans le tas (et non dans la pile) ;
- les variables non primitives *ne contiennent pas leurs données* (mais plutôt l'adresse dans le tas où celles-ci se trouvent)

Pointeurs (types référence)

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

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);
}
```


Pile
Tas
args
null
s
c2
c3
null
tab
objet (String)
"Henri Dupont"
objet (String)
"Paul Durand"
objet (Compte)
titulaire =
numero = 1235
solde = 0
int []
2
1
8

2. Affecter, comparer des références

Affectation entre variables référence = partage de données

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

Affectation entre variables référence (de types compatibles)

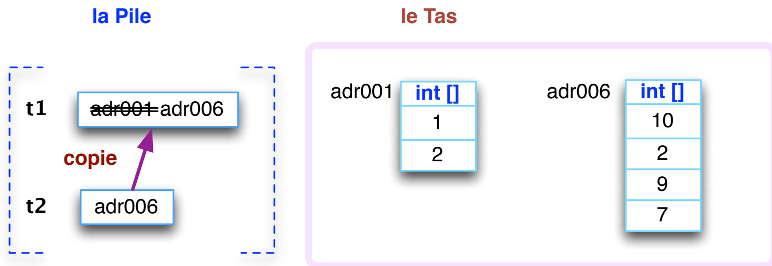
Comportement :

- copie du contenu (une adresse) d'une variable vers l'autre ;
- après coup, elles contiennent la même adresse.

t1, t2 partagent la mêmes donnée ! $\Rightarrow$ si on change une composante dedans, les deux variables « voient » ce changement.

Dessin affectation références (adresses explicites)

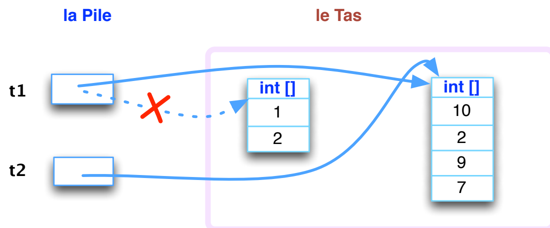
```
int [] t1, t2;
t1 = {1,2};
t2 = {10,2, 9, 7};
t1 = t2;
```



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

Dessiner ce qui change après `t1[2] = 100;`

Affectation entre variables objet

Le même phénomène de partage sur les objets :

```
Compte c1 = new Compte("Dupont", 218, 100);
Compte c2 = new Compte("Durand", 51, 200000);
c1.affiche(); // affiche le solde, numero et titulaire
c2.affiche();
```

```
> java testBis
60 , Dupont ,218
60 , Dupont ,218
```

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

Dans le cas d'adresses, cela teste 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){ System.out.println("t1==t2"); }
if (t1==t3){ System.out.println ("t1==t3"); }
else {System.out.println ("t1!=t3");  }
```

Affichages :

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

Egalité == sur objets

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");   }
```

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

Comparer tableaux, Strings, objets

- Tableaux, Strings et autres objets **sont des adresses**.
- L'opérateur == utilisé pour les comparer, **compare ces adresses**, autrement dit, cela teste s'il s'agit du même espace référencé 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.

3. 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

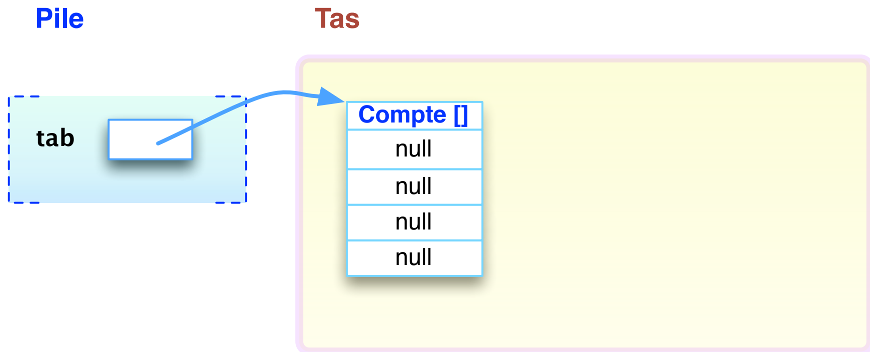
Création d'un tableau avec :

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

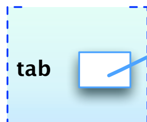
```
Compte [] tab = new Compte [4];
for (int i=0; i<4; i++) {
  tab[i] = new Compte("1235"+i, 0);
}
```

Le tableau de comptes (après création)

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



Le tableau de comptes (après la boucle)



4. Contexte d'exécution d'une méthode

Le contexte d'exécution d'une méthode

Mémoire locale à chaque méthode avec emplacements pour :

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

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 des paramètres passés par 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 3 passée pour le paramètre x ($3 \mapsto x$)

Contexte exécution appel plusUn (3)

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

x	3
res	?

(param)

valeur passée en paramètre

La pile d'exécution (*Heap*)

Rappel :

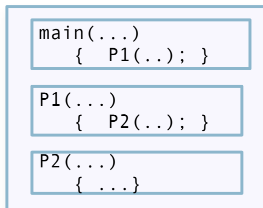
- une méthode P1 peut invoquer P2, qui peut invoquer P3, etc.
- les variables d'une méthode lui sont locales (inaccessibles pour les autres méthodes) ;

Pile d'exécution (*stack*)

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

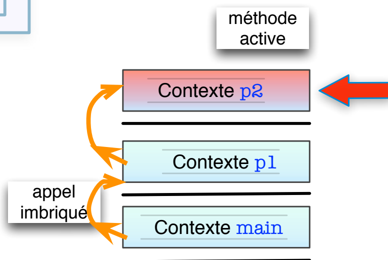
- contient les contextes de toutes les méthodes appelées et non encore terminées ;
- Haut de la pile : contexte de la méthode active (qui s'exécute actuellement).

Fonctionnement de la pile (dessin)



Code exécuté:

main →
p1(..) →
p2(..) → exécution p2

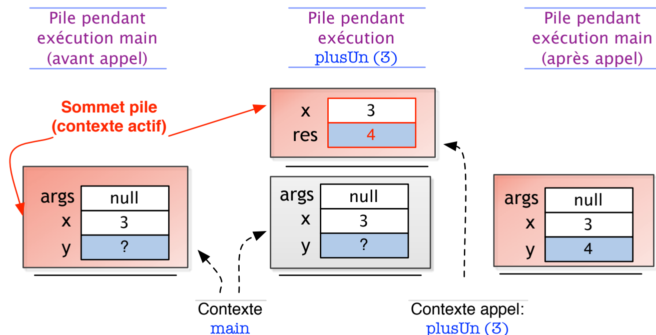


Fonctionnement pile d'exécution (2)

- 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.

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 (PythonTutor) contexte + pile d'exécution, appels imbriquées (méthodes statiques)

5. 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] + "␣");
    }
}
```

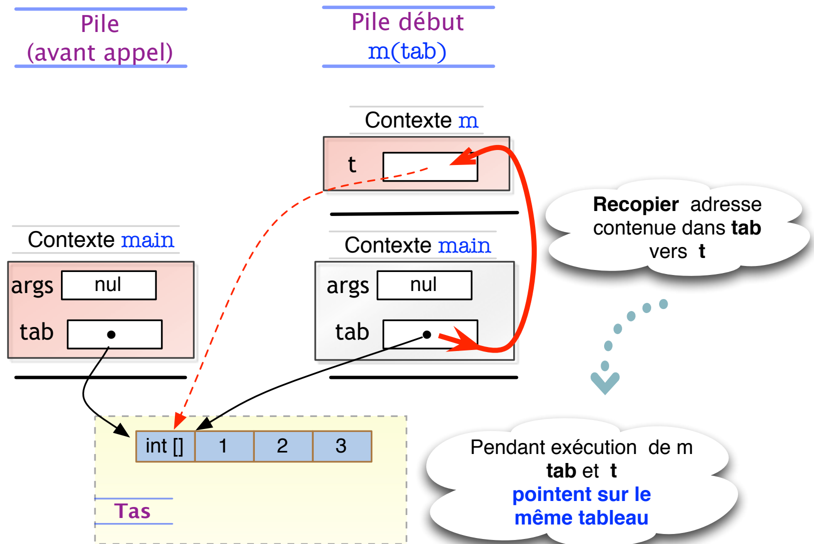
Qu’affiche ce programme ?

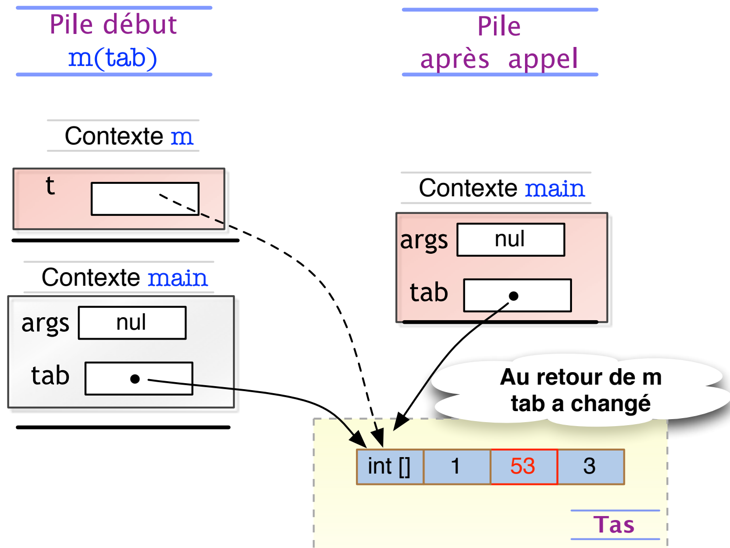
Exemple 1 (2)

```
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)




Pile début
m(tab)
Contexte m
t
Contexte main
args
nul
tab
Pile
après appel
Contexte main
args
nul
tab
Au retour de m
tab a changé
int []
1
53
3
Tas

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

Pile appel
m(b)

Contexte m

a

Contexte main

args	nul
b	•

static void m(Date a){
a.jour = 4

Corps de la méthode

....modifie l'objet dans le tas

objet (Date)
jour = ~~1~~ 4
mois =1
annee=2000

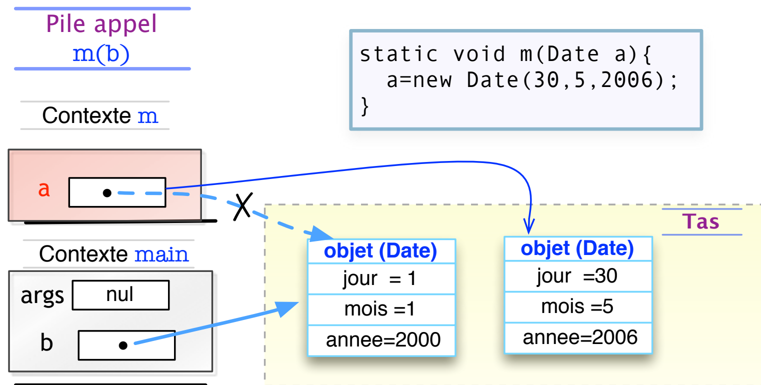
Tas

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).

6. 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)

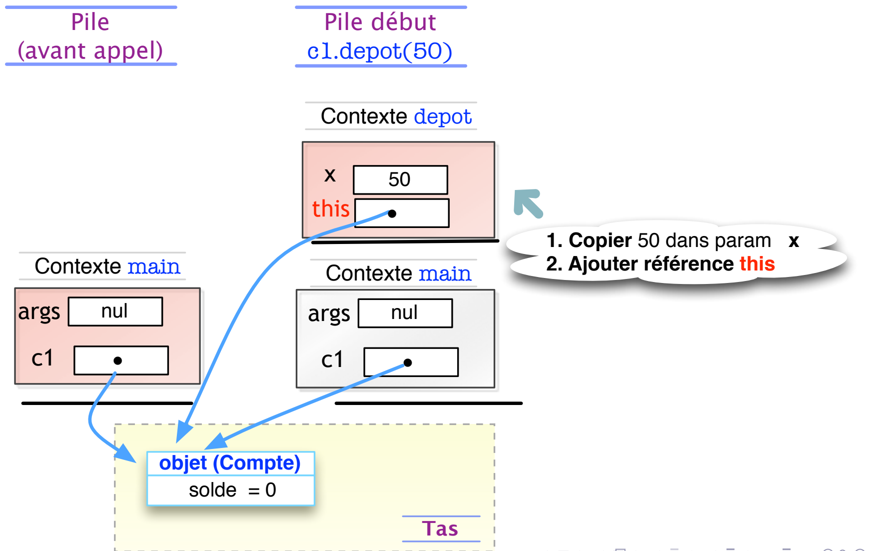
```
Compte c1 = new Compte();
c1.depot(50); // <-- appel methode non statique
```

- 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 méthode d'instance

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