La mémoire d'exécution (partie 1)

Virginia Aponte

CNAM-Paris

23 octobre 2020

La représentation des données en mémoire

Pendant l'exécution d'un programme, ses données et ses variables sont stockées dans la mémoire vive, où elles sont manipulées et modifiées.

Dans ce cours nous étudions :

1. Comment les données et les variables d'un programme sont agencées en mémoire pendant l'exécution ;
2. Comment les instructions du programme modifient/utilisent les données en mémoire.

Le but est de comprendre le comportement des instructions Java afin d'écrire des meilleurs programmes : corrects et robustes.

1. Les données et les variables en mémoire

Deux familles de types de données en Java

- Types Primitifs : comprend les données simples et indivisibles (entier, caractère, etc) ;
- Types Référence : comprend les agrégats de plusieurs données. Peuvent être manipulées comme un tout, ou séparément.
  - **tableaux** : plusieurs données (même type) ;
  - **String** : chaînes de caractères (suite de caractères mis dans une chaîne) ;
  - **objets** : plusieurs données (types $\neq$).

Les variables, l'exécution et la mémoire

- Un programme déclare des variables pour stocker ses données.
- A l'exécution, on assigne à chaque variable un emplacement de la mémoire vive :
  - on y met les valeurs successives prises par chaque variable au cours de l'exécution ;
  - selon son type, chaque donnée et chaque variable est représentée en mémoire de manière différente : le nombre d'octets alloués et la région de la mémoire où elles sont placées pourront différer.

La mémoire d'exécution

La mémoire d'exécution comporte deux grandes zones :

- La Pile (*stack*)
  - sert à stocker *les variables déclarées par les sous-programmes* en cours d'exécution (sous forme d'empilement de *contextes d'exécution*) ;
  - quelques informations de contrôle (où retourner après un appel, ou s'il y a une erreur)
- Le Tas (*heap*)
  - sert à créer *à la volée* et à stocker les nouvelles données composites nécessaires à l'exécution (tableaux, chaînes, objets) ;

2. Contexte d'exécution d'une méthode

Les méthodes et leurs variables

```
static void main (String [] args){
  char c = 'a';
  int y = 7;
  y = y - 5;
  double m = y;
}
```

Les variables d'une méthode sont les variables qu'elle déclare :

- les paramètres déclarés dans son entête (ici, `args`)
- les variables déclarées dans son corps (ici, `c, y, m`)

Les variables d'une méthode sont rangées dans une mémoire locale d'exécution, utilisée par les instructions de la méthode pendant leur exécution.

Contexte d'exécution d'une méthode

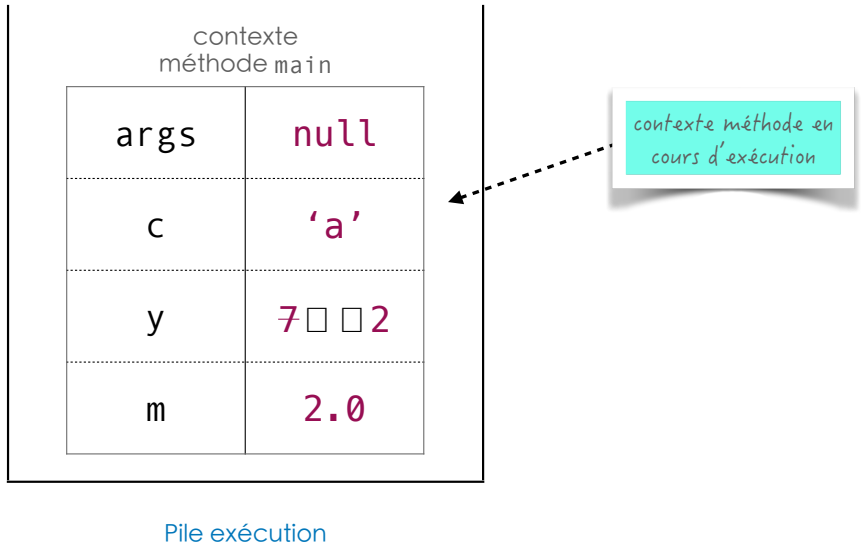
Contexte d'exécution d'une méthode

C'est la mémoire locale dont a besoin une méthode pour s'exécuter.

- c'est une région composée d'emplacements pour chaque variable de la méthode ;
- cette région est dans la pile ;
- pendant l'exécution, la méthode lit/modifie les valeurs de ses variables via cette mémoire.
- ⚠ utilisée uniquement pendant 1 exécution (après, le contexte est sorti de la pile)

Exemple : contexte d'exécution pour méthode main

```
public static void main (String [] args) {
    char c = 'a';
    int y = 7;
    y = y-5;
   double m = y;
}
```



3. La pile d'exécution (premier aperçu)

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).

⚠ *Comme nous ne savons pas encore écrire des méthodes autres que* `main`*, pour le moment notre pile ne contiendra que le contexte de la méthode* `main`.

4. Répresentation des données

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();
```

Répresentation des données en mémoire

Un variable `x` est déclarée de type `T` et initialisée correctement :

```
T x = valeur ;
```

- (quelque part en mémoire) un espace est associé à `x` :
  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 :

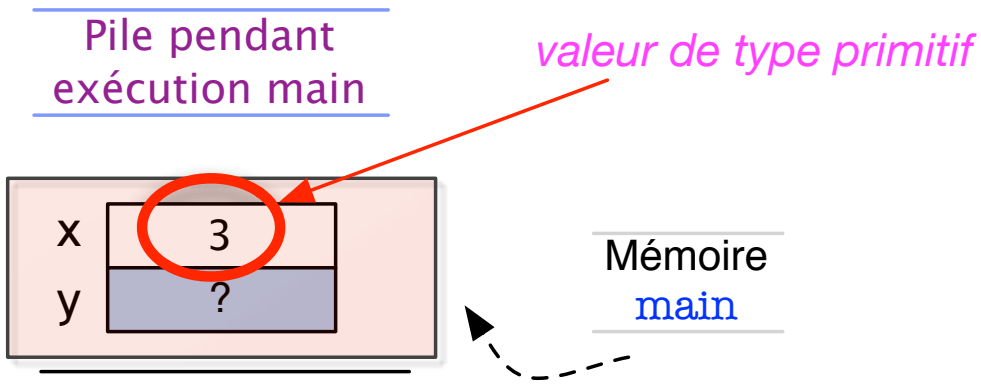
```
int x = 5;
```

- (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 valeurs de types primitifs (int, char, double, etc.) sont représentés en mémoire dans un espace de taille fixe, qui dépend du type de la 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];
```

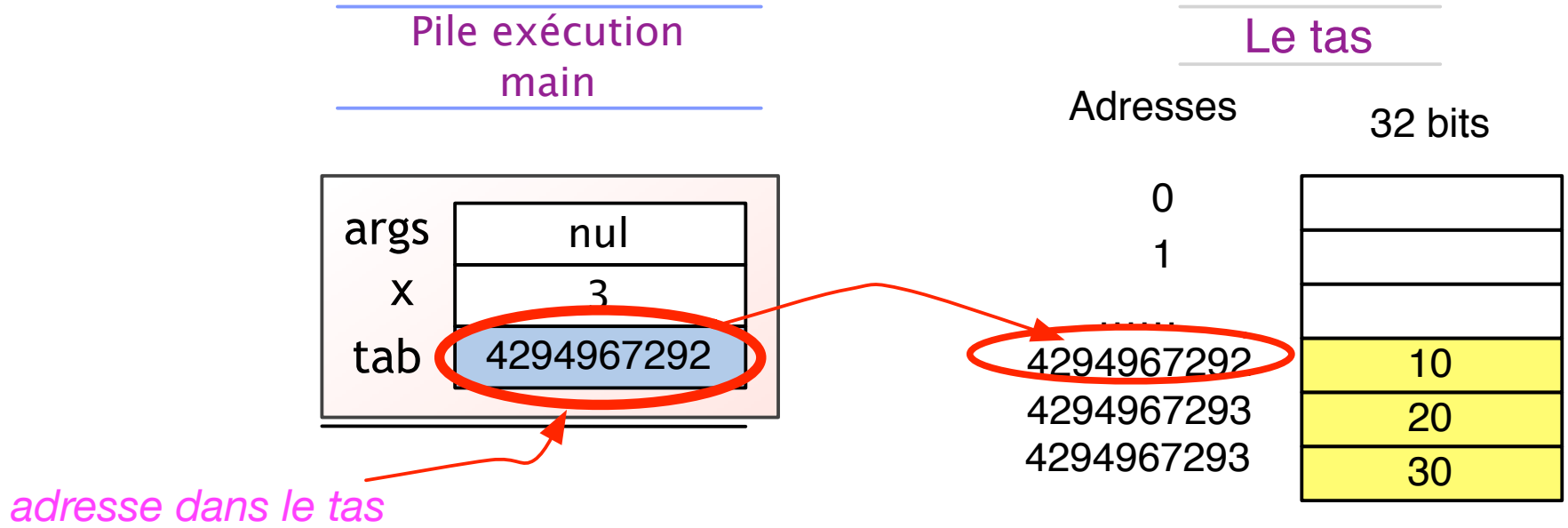
- (quelque part) un emplacement mémoire est associé à `x` :
  1. la taille de cet emplacement ? 32 ou 64 bits.
  2. que contient-il ? une adresse mémoire du Tas
     - à cette adresse se trouve un espace d'au moins 5x32 bits pour stocker les 5 entiers du tableau, initialisés à 0.

Une valeur de type non primitif est représenté par une adresse mémoire dans le tas. À cette adresse sont stockées les données de la variable.

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 appellées 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

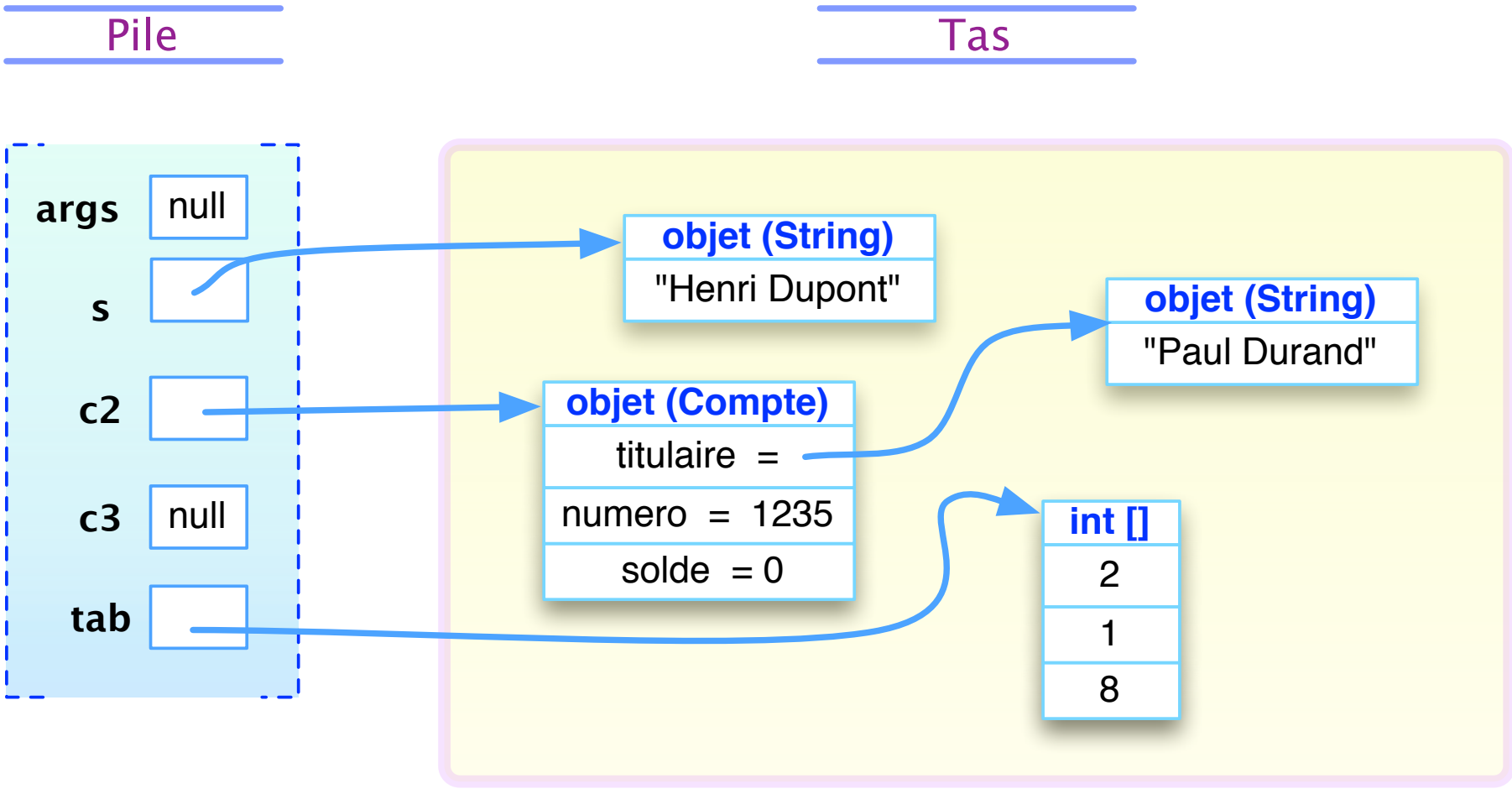
La valeur contenue dans une variable est soit

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

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



5. 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

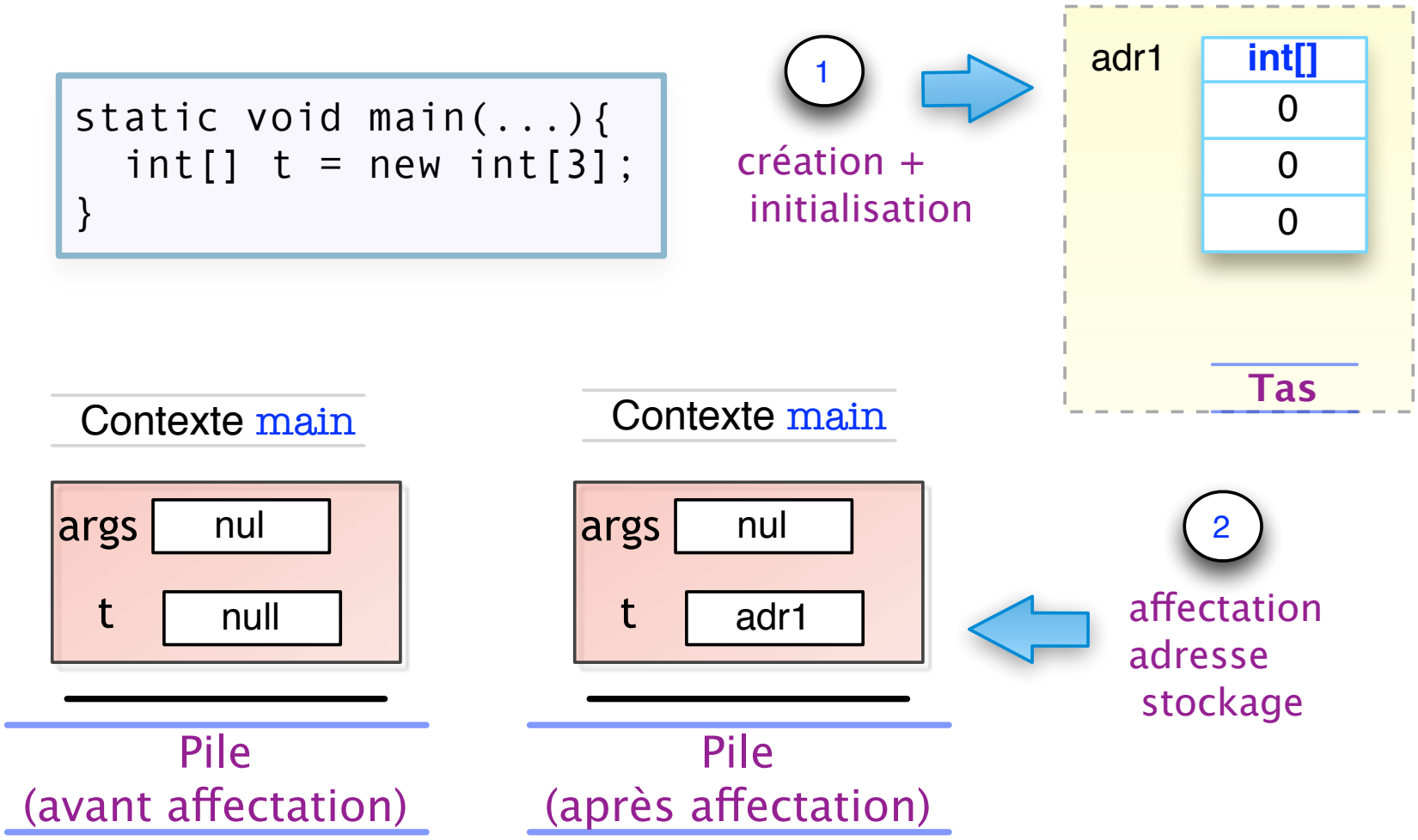
Se fait (sauf pour les types objet prédéfinis) via la syntaxe :

`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];
}
```



6. 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

Possible si leurs types sont compatibles. Ex : 2 Comptes, 2 tableaux, etc.

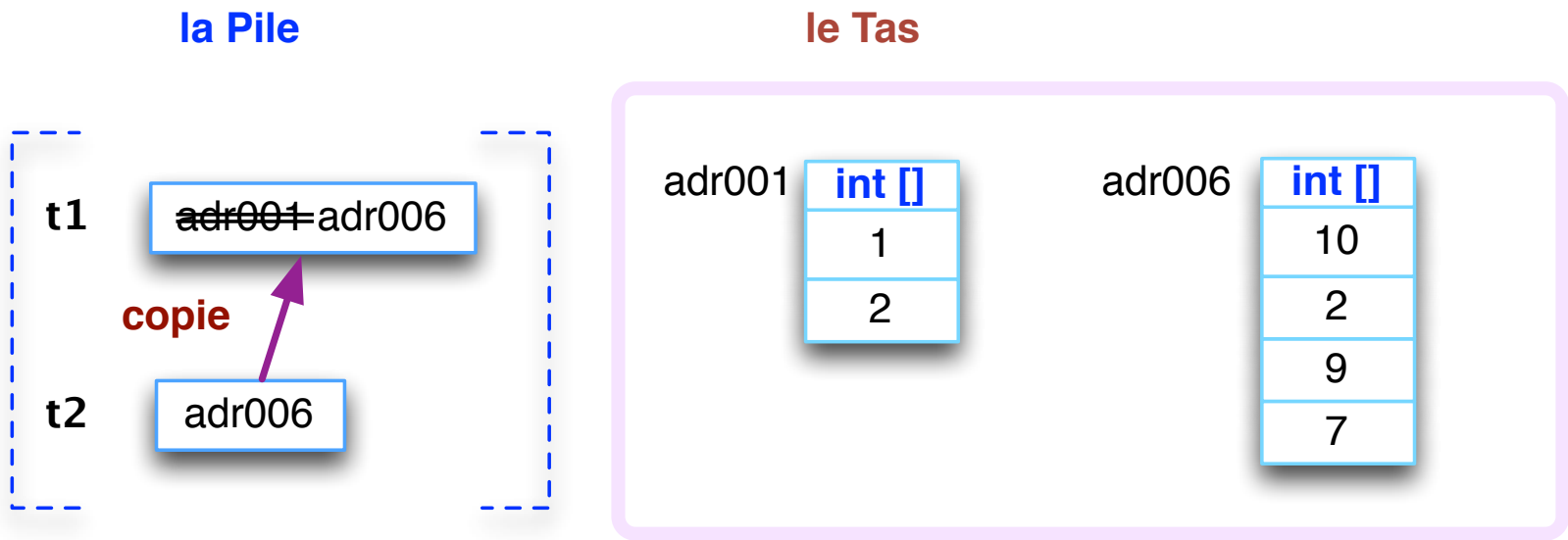
Comportement :

- 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)

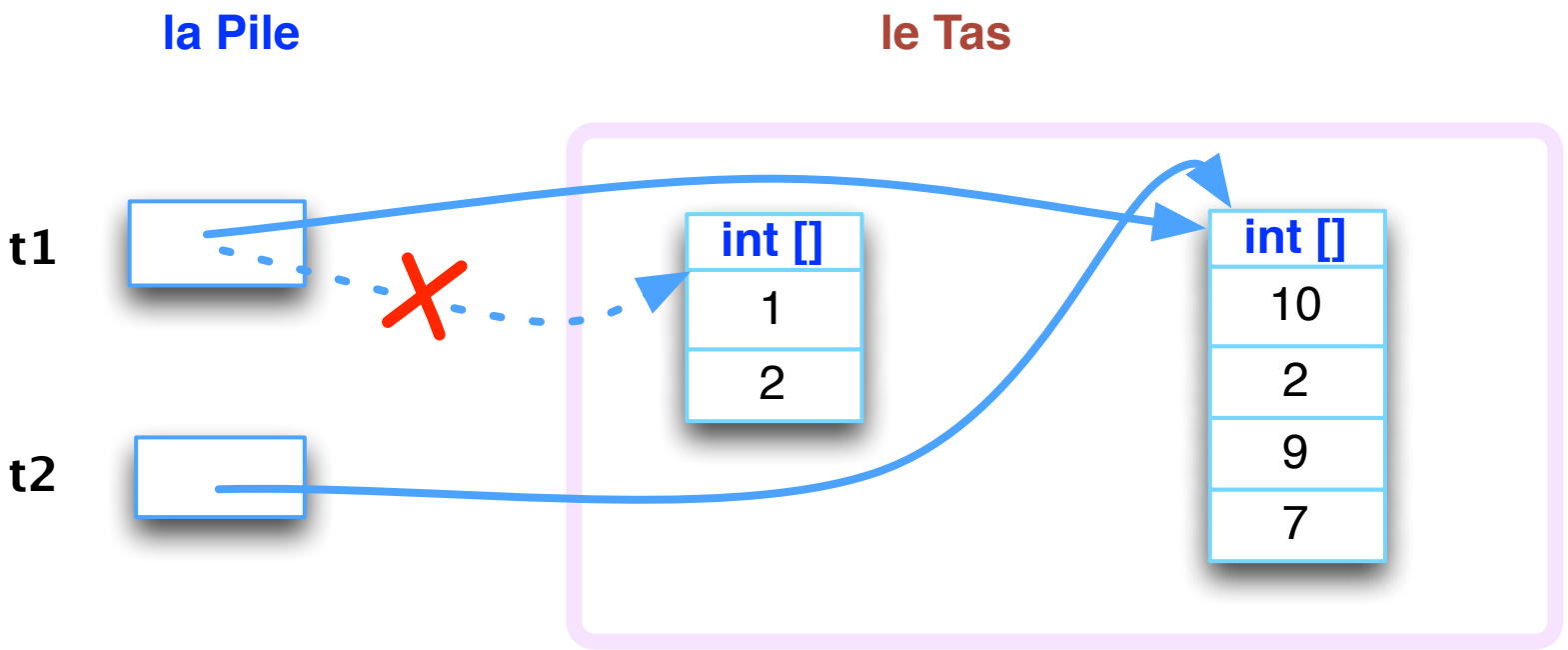
```
int [] t1, t2;
t1 = {1,2};
t2 = {10,2, 9, 7};
t1 = t2;
```



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**.
  ⇒ t1 et t2 contiennent la **même adresse**.
- Elles pointent vers le même emplacement physique de la mémoire.
  ⇒ 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])` ⇒ affiche 50

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
```

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.

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] + "␣");
   }
}
```

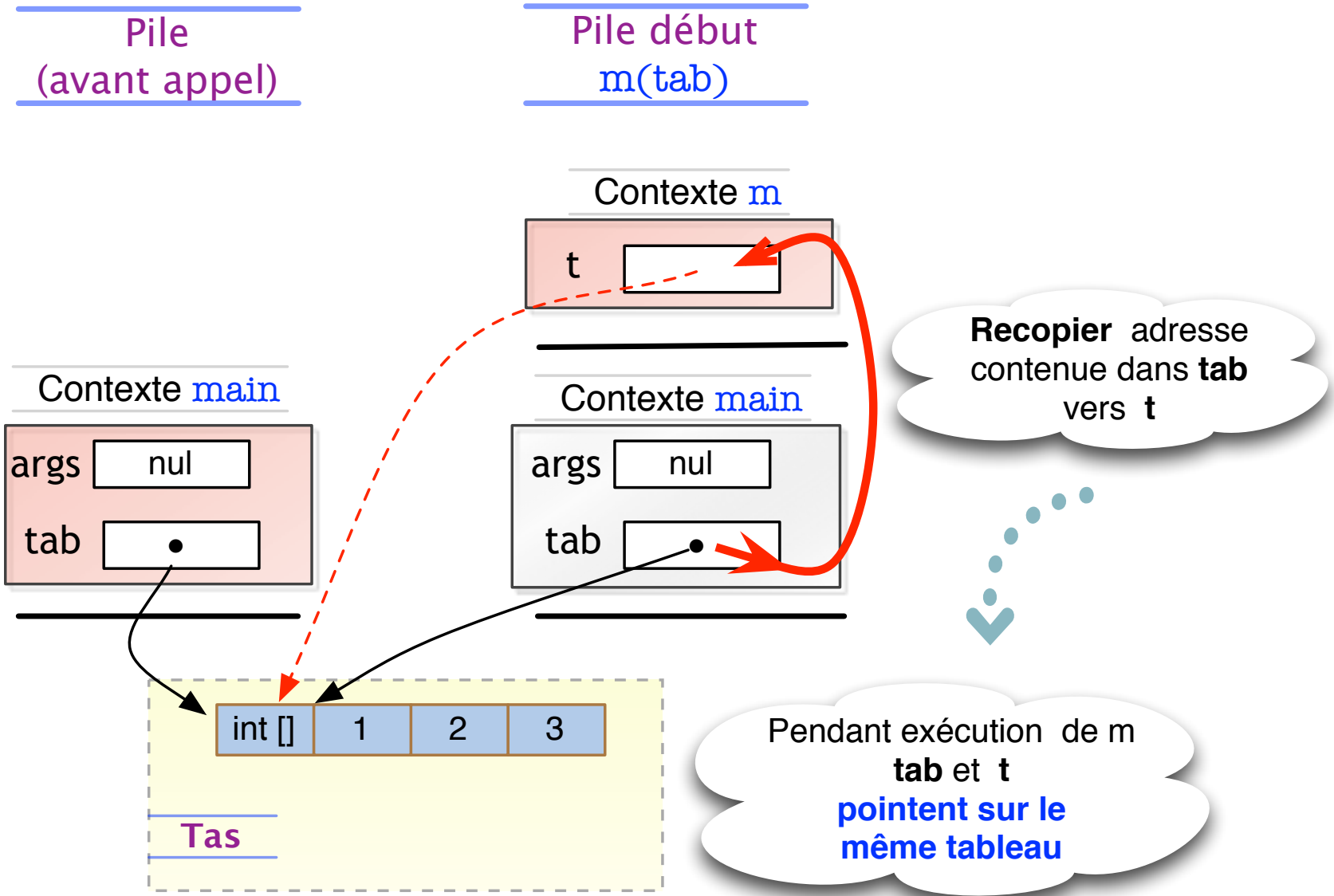
Qu'affiche ce programme ?

Exemple 1 (2)

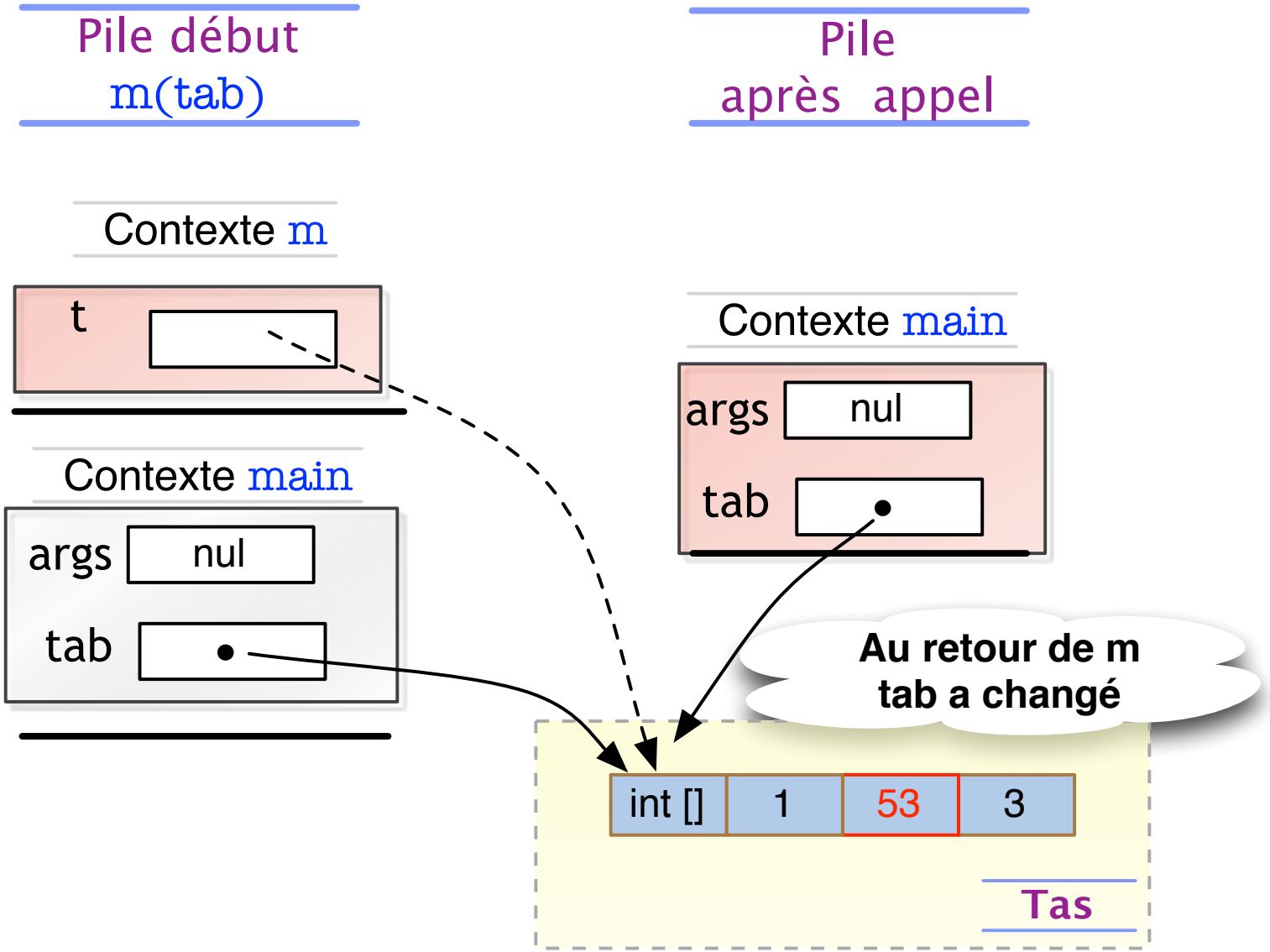
```
static void m(int [] t){
  t[1] = 53;
}
public static void main(String [] args){
   int [] tab = {1,2,3};
   m(tab);
   for (int i=0; i< tab.length; i++){
    Terminal.ecrireString(tab[i] + "␣");
   }
}
```

- 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)


Pile
(avant appel)
Pile début
m(tab)
Contexte m
t
Contexte main
args
nul
tab
Contexte main
args
nul
tab
Recopier adresse contenue dans tab vers t
int []
1
2
3
Tas
Pendant exécution de m
tab et t
pointent sur le
même tableau

Exemple 1 (4)


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