Support de cours

Cours:

Initiation à la programmation (en Java)

Vidéo:

Init-JAVA-06-3-Arguments-pt2

Concepts (extraits des sous-titres générés automatiquement) :

Valeur différente. Nature du type. Référence v. Valeur distincte. Données de types élémentaires. Sortir de la méthode. Possibilités de modification. Moment de l'appel. Zone mémoire. Tableau d'entiers. Méthode m. Programme principal. Zone locale. Valeur précédente. Données de types évolués.

vers la recherche de séquences vidéo
(dans Initiation à la programmation (en Java).)

vers la vidéo

EPFL

Fonctions : passage des arguments
(Partie 2)

Initiation à la programmation (Java)

Jamila Sam, Vincent Lepetit et Jean-Cédric Chappelier

...

notes

résumé

0m 0s

Le passage des arguments (1)

Considérons la situation suivante (pseudo-code) :

```
static void methode(Type v) {
  // traitement modifiant v
}

// ailleurs, dans le programme principal,
// par exemple:
Type v1 = .. ; // initialisation de v1
methode(v1);
// v1 EST-ELLE MODIFIEE ICI OU NON???
```

En programmation de façon générale, on dira que :

- L'argument `v` est **passé par valeur** si `methode` ne peut pas modifier `v1` : `v` est une **copie locale** de `v1`.
- L'argument `v` est **passé par référence** si `methode` peut modifier `v1`

Mais que veut dire «modifier `v`» ?

En Java, nous allons nous poser cependant la question de que signifie

notes

résumé

0m 1s

Le passage des arguments (2)

Java ne manipule pas les types élémentaires comme les types évolués :

modifier v concrètement ? Car la réponse à cette question va dépendre de la nature du type.

notes

résumé

0m 6s

Le passage des arguments (2)

Java ne manipule pas les types élémentaires comme les types évolués :

Modifier `v` pour un type élémentaire n'a qu'une seule interprétation possible :

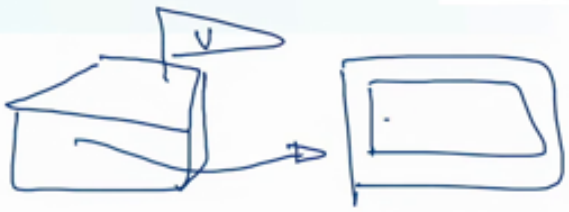
Lors d'une séquence antérieure sur les types, nous avons vu qu'en Java les données de types élémentaires n'étaient pas manipulées de la même façon que les données de types évolués. Donc types évolués, on peut par exemple penser aux tableaux ou aux chaînes de caractères.

notes

résumé

0m 17s

Le passage des arguments (2)



Il y a donc deux questions à poser au lieu d'une :

```
static void methode(Type v) { // Type :type EVOLUÉ
// traitement modifiant l'objet référencé par v
// traitement modifiant v lui même (référence)
}
// ailleurs:
Type v1 = ..; // initialisation de v1
methode(v1);
//1. v1 est-elle modifiée ici?
//2. l'objet référencé par v1 est-il modifié
//   ici?
```

Par conséquent, si je suis en train de manipuler une donnée de type élémentaire, modifier v n'a qu'une seule interprétation possible. Si ici je tente de modifier v en lui affectant une valeur différente, il est clair que désormais la zone mémoire nommée v va contenir une valeur distincte de ce qui était sa valeur précédente, la valeur dix en l'occurrence ici. Par contre, si je suis en train de travailler avec des données de type évolué le discours est différent. Nous avons vu qu'en effet les valeurs de type évolué étaient manipulées via des références, via des indirections. Si je manipule une donnée de type string par exemple, je ne manipule pas directement la chaîne de caractères, mais je manipule une référence vers la chaîne de caractères. Par conséquent, lorsque je parle de modifier v, qu'est-ce que ça signifie exactement ? Suis-je en train de modifier la référence? Suis-je en train de modifier la zone mémoire pointée par la référence ? Plaçons-nous maintenant dans la situation où notre méthode prend en paramètre une donnée de type évolué. Par exemple, on peut imaginer que type corresponde à un tableau d'entiers, dans ce cas la référence v est une indirection, une référence vers un tableau d'entiers que l'on peut schématiser de cette façon-là.

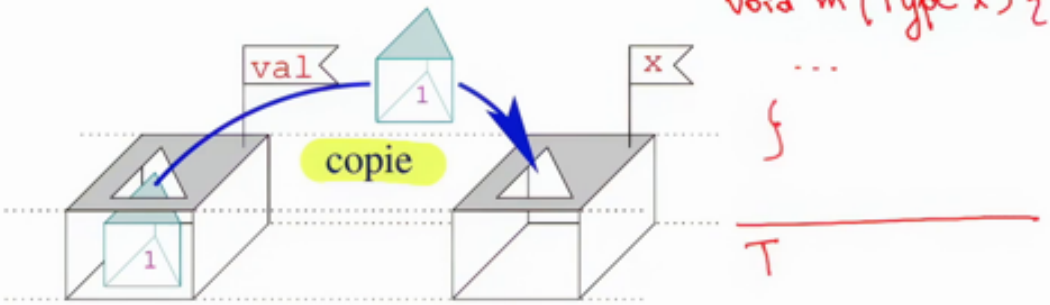
notes

résumé

0m 31s

Passage d'argument par valeur : schéma

En Java, il n'existe que le passage par valeur : une méthode travaille toujours sur une copie de la valeur qui lui est passée en paramètre



Il y a donc deux possibilités de modification ici. Celle qui vient peut-être le plus naturellement à l'esprit consiste à mettre une valeur alternative dans v, ce qui veut dire qu'on est en train de modifier la référence elle-même, ce qui aurait pour incidence, puisqu'ici je change la référence elle-même, de faire pointer v vers un autre tableau. Deuxième possibilité de modification, je peux via la référence modifier l'objet référencé, par exemple écrire ceci, ce qui aura pour conséquence d'altérer la seconde entrée du tableau de cette façon-là. Il y a donc en Java deux questions à se poser au lieu d'une. Première question : si ma méthode altère la référence, est-ce que la variable, la référence, passée en argument se trouve modifiée au sortir de la méthode ? Deuxième question : si je modifie l'objet référencé par v1 via v, est-ce que l'objet référencé par v1 est modifié ? Alors il faut savoir qu'en Java, il n'existe en réalité que le passage par valeur, ce qui signifie concrètement qu'une méthode travaille toujours systématiquement qu'avec une copie de l'argument qui lui est passé au moment de l'appel. Dans le cas des types évolués c'est donc bien une référence qui est copiée. Si je me place ici dans une situation concrète où j'ai une méthode m, ici sans type de retour, qui prend en paramètre une donnée d'un type quelconque nommée x, supposons par exemple que dans un programme principal j'invoque cette

notes

résumé

1m 49s

Passage par valeur : type élémentaire

```
static void methode(Type v) {
  // traitement modifiant v
}

// ailleurs, dans le programme principal,
// par exemple:
Type v1 = .. ; // initialisation de v1
methode(v1);
// v1 EST-ELLE MODIFIEE ICI OU NON???
```

☞ Si `Type` est un type élémentaire
la réponse à la question dans le code est NON !!

méthode de la façon suivante où je déclare une donnée val du bon type ensuite j'appelle ma méthode en lui passant en argument val, il faut savoir qu'à ce moment-là, la valeur de val est copiée dans une zone locale à la méthode.

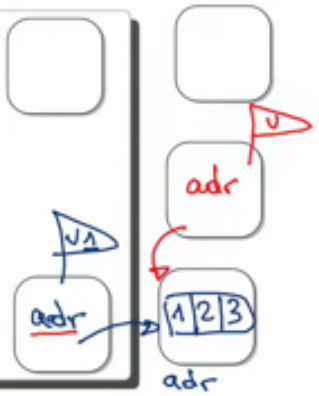
notes

résumé

3m 37s

Passage par valeur : type évolué

```
static void methode(Type v) { // Type :type EVOLUÉ
// traitement modifiant l'objet référencé par v
// traitement modifiant v lui même (référence)
}
// ailleurs:
Type v1 = ..; // initialisation de v1
methode(v1);
//1. v1 est-elle modifiée ici?
//2. l'objet référencé par v1 est-il modifié
//   ici?
```



- On a toujours un passage par valeur donc la référence `v` est une copie de `v1`.
- Cependant, `Type` étant évolué, l'argument qui est donné à `methode` lors de l'appel `methode(v1)` est une copie de la référence à `v1` (son adresse) : l'objet pointé par `v` est le même que l'objet pointé par `v1`. **Toute modification faite sur l'objet référencé via `v` est donc visible via `v1`** !

☞ La réponse à la question 2 est donc OUI (et reste non pour la question 1)

Donc, reprenons notre question de départ, en nous intéressant au cas où type est un type élémentaire. Par exemple, on peut imaginer que type correspond au type entier, int, et donc le traitement modifiant v pourrait être simplement ici une incrémentation de cette nature-là. Dans notre programme principal, v1 serait déclaré de type entier, en lui affectant une valeur d'origine, et on invoquerait la méthode sur v1. Donc ici en l'occurrence, nous nous trouvons avec une variable v1 qui contient la valeur trois et au moment où cet appel est réalisé, v1 est en fait copié dans une zone locale à la méthode qui s'appelle v, donc ici on aurait une copie de la valeur. Si on exécute ensuite le corps de la méthode, on altérerait le contenu de v puisque on est entrain de travailler sur v et on voit qu'à nul moment la valeur v1 n'est concernée par cette modification. Donc dans le cas où le type est élémentaire, la réponse à cette question est non en raison du passage par valeur. Donc, plaçons-nous maintenant dans la situation où notre méthode travaille avec une donnée de type évolué. Prenons en guise de type évolué par exemple un tableau d'entiers, donc ici dans notre programme principal notre écririons quelque chose qui ressemble à ceci. Ici, nous nous trouvons en mémoire dans la situation suivante ; donc nous avons une variable v1 qui contient une référence en quelque sorte l'adresse vers le tableau un, deux, trois. Donc au moment où est réalisé cet appel de méthode, nous avons vu qu'en Java on ne connaît que le passage par valeur, ce qui signifie que v1 va être copié dans une zone locale v, locale à la méthode. Donc ici nous avons une zone mémoire v, locale à la méthode, et dans laquelle nous allons copier le contenu de v1, c'est-à-dire la référence vers le tableau. On voit bien ici que

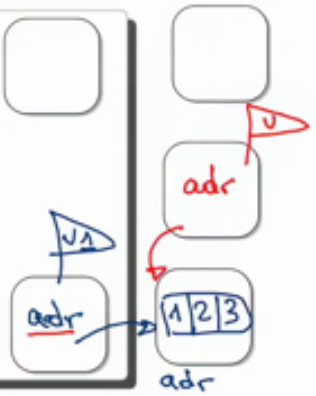
notes

résumé

3m 53s

Passage par valeur : type évolué

```
static void methode(Type v) { // Type :type EVOLUÉ
// traitement modifiant l'objet référencé par v
// traitement modifiant v lui même (référence)
}
// ailleurs:
Type v1 = ..; // initialisation de v1
methode(v1);
//1. v1 est-elle modifiée ici?
//2. l'objet référencé par v1 est-il modifié
//   ici?
```



- On a toujours un passage par valeur donc la référence `v` est une copie de `v1`.
- Cependant, `Type` étant évolué, l'argument qui est donné à `methode` lors de l'appel `methode(v1)` est une copie de la référence à `v1` (son adresse) : l'objet pointé par `v` est le même que l'objet pointé par `v1`. **Toute modification faite sur l'objet référencé via `v` est donc visible via `v1`** !

☞ La réponse à la question 2 est donc OUI (et reste non pour la question 1)

autant v que v1 pointent vers le même tableau en mémoire. Ceci signifie que si par exemple, je tente maintenant de modifier l'objet référencé via v, donc si j'essaie dans la méthode de faire ceci, donc ici je modifie le tableau via ce lien, mais puisque autant v que v1 pointent vers le même emplacement, les modifications faites via v sont visibles via v1. Ceci donc signifie que dans le cas où je réalise un traitement dans la méthode qui modifie l'objet référencé, la réponse à cette question est oui. Intéressons-nous maintenant à la seconde situation, essayons de réaliser un traitement qui modifie, dans la méthode, la référence elle-même. Donc ici, du fait du passage par valeur, au moment de cet appel nous avons toujours une copie de v1 dans v, ce qui signifie que nous sommes dans cette situation-là où nous avons copié la référence dans v, la référence dans le tableau dans v.

notes

résumé

Exemple de passage par valeur (type élémentaire)

```
public static void main(String[] args) {
  int val = 1;
  m(val);
  System.out.println(" val=" + val);
}

static void m(int x) {
   x = x + 1;
   System.out.print(" x=" + x);
}
```

L'exécution de ce programme produit l'affichage :

`x=2 val=1`

Ce qui montre que les modifications effectuées à l'intérieur de la méthode `m()` ne se répercutent pas sur la variable extérieure `val` associée au paramètre `x` et passée par valeur.

Maintenant, tentons de modifier cette référence, donc ceci peut se faire par exemple de cette façon-là, je déclare un autre tableau t, qui a d'autres valeurs, et je modifie la référence elle-même donc je réalise cette affectation-là. Donc ici, je suis dans cette situation-là en mémoire, j'ai donc une variable t qui contient une référence vers un autre tableau, j'ai donc ce lien, et je fais cette affectation qui signifie que j'altère la référence elle-même, ce qui veut dire que je casse ce lien, ce lien n'existe plus, désormais ma variable locale pointe vers le tableau référencé par t. On voit donc que par le biais de cette manipulation, comme on a modifié la référence sur une copie locale, v1 n'est nullement affecté par cette modification, lorsque je termine d'exécuter ma méthode, donc à ce point-là j'ai toujours au sein de v1 la valeur adresse. Ce qui signifie concrètement que si je réalise un traitement qui modifie la référence elle-même, la réponse à cette question demeure non, toujours en raison du passage par valeur. Illustrons maintenant cette discussion sur des cas concrets, des exemples concrets. Donc, nous avons ici à faire à un petit programme principal qui commence par déclarer une variable val, cette variable est de type élémentaire, de type int, ce qui signifie que la valeur qui lui est associée est directement stockée dans val, il n'y a pas d'indirection, pas de référence. La seconde instruction invoque une méthode m en lui passant en argument val. Nous avons vu qu'en Java il n'existe que le passage par valeur, ce qui signifie que

notes

résumé

7m 13s

Type évolué : modification de la référence

```
public static void main(String[] args) {
   int[] tab = {1};
   m(tab); // tab (référence)
           // PASSAGE PAR VALEUR AUSSI
   System.out.println(" tab[0]= " + tab[0]);
}
static void m(int[] x) {
   int[] t = {100};
   x = t; //Modification de la référence
          //(on met une autre adresse dans x)
   System.out.print("x[0]= " + x[0]);
}
```

la valeur de val est copiée dans une zone locale à la méthode qui s'appelle ici x. Donc nous nous trouvons dans la situation où x contient la même valeur que val, mais il s'agit d'une autre zone en mémoire. Ensuite s'exécute la première instruction du corps de la méthode qui a pour rôle d'incrémenter la valeur de x, nous nous trouvons avec la valeur deux pour x, et lorsqu'on fait afficher la valeur de x on voit bien qu'il s'agit bien de la valeur deux. Ensuite, lorsque la méthode a terminé son exécution, si on veut faire afficher la valeur de val, on réalise bien que cette valeur est demeurée inchangée. Elle n'a pas été modifiée en raison du passage par valeur. Tout ceci pour dire que les modifications dans ce cas de figure qui sont effectuées à l'intérieur de la méthode ne se répercutent pas sur la variable extérieure val.

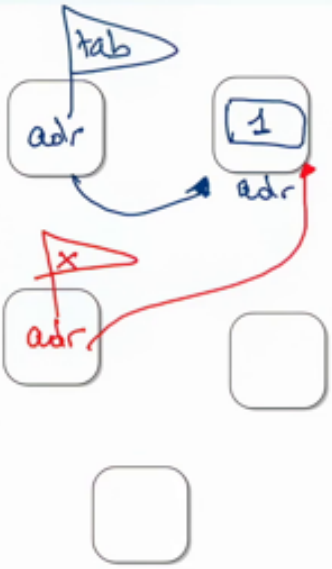
notes

résumé

9m 1s

Type évolué : modification de la référence

```
public static void main(String[] args) {
    int[] tab = {1};
    m(tab); // tab (référence)
            // PASSAGE PAR VALEUR AUSSI
    System.out.println(" tab[0]= " + tab[0]);
}
static void m(int[] x) {
    int[] t = {100};
    x = t; //Modification de la référence
           //(on met une autre adresse dans x)
    System.out.print("x[0]= " + x[0]);
}
```



Autre situation, celle de la modification d'une référence dans le corps d'une méthode. ici, nous nous trouvons dans la situation où nous avons un programme principal qui déclare une variable tab qui est de type évolué, c'est un tableau d'entiers, donc les types évolués sont manipulés via des interactions, nous nous trouvons donc en mémoire dans cette situation-là, où tab pointe vers un tableau qui en l'occurrence contient une seule cellule contenant un. Ensuite, il y a l'appel de la méthode m qui était exécuté, et nous avons vu qu'il n'y a que du passage par valeur en Java, ce qui signifie que tab est copié dans une zone locale à la méthode qui est x. Donc nous nous trouvons maintenant dans cette situation où nous avons copié la valeur de tab dans x et donc x pointe vers le même tableau en mémoire. En exécutant le corps de la méthode,

notes

résumé

9m 54s

Type évolué : modification de la référence

```
public static void main(String[] args) {
   int[] tab = {1};
   m(tab); // tab (référence)
           // PASSAGE PAR VALEUR AUSSI
   System.out.println(" tab[0]= " + tab[0]);
}
static void m(int[] x) {
   int[] t = {100};
   x = t; //Modification de la référence
          //(on met une autre adresse dans x)
   System.out.print("x[0]= " + x[0]);
}
```

L'exécution de ce programme produit l'affichage :

```
x[0]= 100 tab[0]= 1
```

Les modifications faites dans la méthode sur la référence elle-même ne sont pas visibles à l'extérieur de la méthode !

la première instruction rencontrée est celle créer un nouveau tableau. Ce dernier va donc pointer vers le nouveau tableau en mémoire avec une autre référence, qui en l'occurrence contient une seule et unique cellule valant cent. Et la seconde instruction tente d'altérer la référence elle-même, donc on affecte à x l'adresse du nouveau tableau, ce qui a pour conséquence de casser ce lien et d'en créer un nouveau vers ce tableau. Si maintenant on fait afficher la valeur de la première cellule du tableau x, x de zéro étant donné le lien nouvellement créé, vaut clairement cent. Lorsque la méthode a terminé son exécution et qu'on fait afficher la valeur de la première cellule du tableau tab, on voit que l'altération faite sur la référence elle-même n'est nullement visible sur l'argument en raison du passage par valeur et du coup la valeur de tab de zéro reste inchangée et vaut toujours donc un.

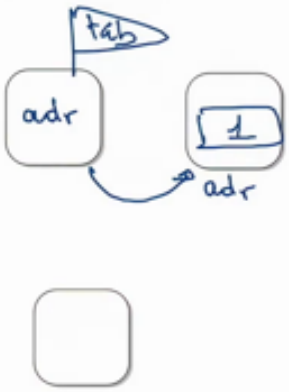
notes

résumé

10m 49s

Type évolué : modification de l'objet référencé

```
public static void main(String[] args) {
   int[] tab = {1};
   m(tab);
   System.out.println(" tab[0]= " + tab[0]);
}
static void m(int[] x) {
   x[0] = 100; // modification de l'objet
               // referencé par x
   System.out.print("x[0]= " + x[0]);
}
```



Donc le résultat de l'exécution de ce programme nous montre que les modifications faites dans la méthode sur la référence elle-même ne sont pas visibles à l'extérieur de la méthode, toujours en raison du passage par valeur.

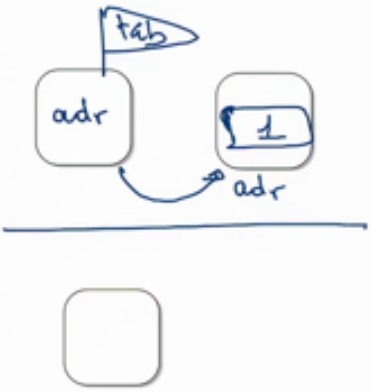
notes

résumé

12m 4s

Type évolué : modification de l'objet référencé

```
public static void main(String[] args) {
   int[] tab = {1};
   m(tab);
   System.out.println(" tab[0]= " + tab[0]);
}
static void m(int[] x) {
   x[0] = 100; // modification de l'objet
               // referencé par x
   System.out.print("x[0]= " + x[0]);
}
```



Dernière situation, celle de la modification de l'objet référencé dans le corps d'une méthode. Donc nous nous trouvons dans la même situation que l'exemple précédent où le programme principal crée un tableau contenant une seule cellule valant un.

notes

résumé

12m 22s

Type évolué : modification de l'objet référencé

```
public static void main(String[] args) {
   int[] tab = {1};
   m(tab);
   System.out.println(" tab[0]= " + tab[0]);
}
static void m(int[] x) {
   x[0] = 100; // modification de l'objet
               // referencé par x
   System.out.print("x[0]= " + x[0]);
}
```

L'exécution de ce programme produit l'affichage :
`x[0]= 100 tab[0]= 100`

Les modifications faites dans la méthode sur l'objet référencé restent visibles à l'extérieur de la méthode !
(on a copié dans `x` la référence `tab` : `x` et `tab` pointent sur le même tableau.)

Au moment de l'appel de la méthode m en lui passant l'argument tab, en raison toujours du passage par valeur, tab est copié dans x, x qui est une zone locale à la méthode m. Nous voyons bien qu'ici autant tab que x référence le même tableau. Donc ici en exécutant le corps de la méthode m, la première instruction rencontrée va altérer la première cellule du tableau pointé par x, ce qui aura pour effet de modifier cette cellule. Et nous voyons que la modification faite sur la première cellule du tableau x est également visible dans tab. Donc l'exécution de cette instruction d'affichage va clairement afficher que x zéro a pour valeur cent. Lorsque nous terminons l'exécution de la méthode et passons à l'affichage suivant, étant donné que aussi bien tab que x pointent vers le même tableau, l'affichage de la valeur de tab de zéro sera également cent.

notes

résumé

12m 37s

Donc la situation ici est que la modification faite à l'intérieur de la méthode est également visible à l'extérieur de la méthode en dépit du passage par valeur. Pourquoi ? Parce que aussi bien la zone locale à la méthode que celle de l'argument pointe en réalité sur le même tableau. pointe en réalité sur le même tableau.

notes

résumé

13m 53s