

Support de cours

Cours:

Introduction à la programmation orientée objet (en Java)

Vidéo:

W12-04-complements-JAVA-pt2

Concepts (extraits des sous-titres générés automatiquement) :

Manipulation des objets. Variable de type rectangle. Ligne de code. Constructeur de copie. Valeurs particulières. Programme java. Copie distincte. Séquences précédentes. Objets distincts. Point suivant. Méthode clone. R1. Exemple concret. Champs largeur. Rôle de notre objet.



[vers la recherche de séquences vidéo](#)

(dans Introduction à la programmation orientée objet (en Java).)



[vers la vidéo](#)

Center for Digital Education. Plus de matériel de soutien pédagogique ici :

<https://www.epfl.ch/education/educational-initiatives/cede/educational-technologies-gallery/boocs-en/>

Fin de vie, affectation, affichage et comparaison d'objets

(Partie 2)

Introduction à la programmation orientée objet (en Java)

Jamila Sam, Jean-Cédric Chappelier et Vincent Lepetit

...

notes

résumé

0m 0s





Passons maintenant au point suivant, la manipulation des objets

notes

résumé

0m 1s



Supposons que l'on souhaite :

- ▶ créer un objet **b** à partir d'un autre objet **a** du même type ;
- ▶ assurer que **a** et **b** soient deux objets **distincts** en mémoire

|

via des références, et les implications que cela peut avoir. A chaque fois que nous créons un objet dans un programme Java, nous avons vu qu'en réalité cet objet est manipulé via une référence. Lorsque je déclare une variable de type Rectangle la situation en mémoire est la suivante.

notes

résumé

0m 5s



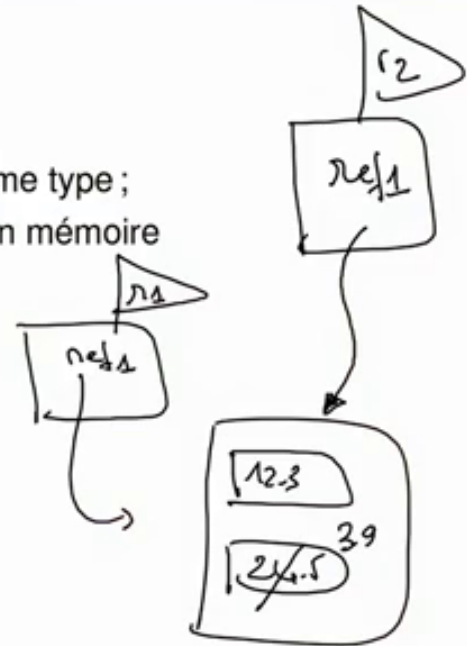
Supposons que l'on souhaite :

- ▶ créer un objet **b** à partir d'un autre objet **a** du même type ;
- ▶ assurer que **a** et **b** soient deux objets **distincts** en mémoire

```
Rectangle r1 = new Rectangle(12.3, 24.5);
Rectangle r2 = r1;
```

r1 et **r2** référencent ici le **même objet**
et non pas deux copies distinctes du même objet :
toute modification via **r2** modifiera également **r1**.

r2.setHauteur(3.9);



Ma variable de type Rectangle, appelons-la « rect » est en réalité une référence vers un objet. Avec ces différents attributs. On s'intéresse ici au problème suivant. On suppose que l'on souhaite créer un objet « b » à partir d'un objet « a » du même type avec la contrainte que « a » et « b » doivent être pour nous deux objets distincts en mémoire. Comment le faire proprement en Java sachant que les objets sont manipulés via des références ? Prenons un exemple concret. Ici le rôle de notre objet « a » est tenu par un rectangle « r1 ». On souhaite à partir de ce rectangle « r1 » créer un autre rectangle, donc du même type, mais on souhaite que les deux objets « r2 » et « r1 » soient deux objets distincts en mémoire. Alors en Java, créer un objet « r2 » à partir d'un objet « r1 » en utilisant en utilisant l'affectation ne va pas nous permettre d'atteindre l'objectif d'avoir deux objets distincts en mémoire. Voyons concrètement pourquoi. Lorsque j'exécute cette ligne de code, J'ai donc en mémoire la situation suivante, au terme de l'exécution, J'ai dans ma variable « r1 » la référence vers un objet rectangle qui a des valeurs particulières pour ses champs largeur et hauteur. Lorsque ensuite j'exécute cette ligne de code, Je vais créer une variable « r2 ». Supposons que l'on note ici « ref1 » la référence contenue dans « r1 » Lorsque l'on réalise cette affectation, on se contente de copier dans la variable « r2 » la valeur contenue dans « r1 » à savoir la référence au même objet. Ce qui veut dire concrètement que l'on va pointer ici sur le même objet en mémoire « r1 » et « r2 » ne désignent pas deux objets distincts en mémoire. Ce qui veut

notes

résumé

0m 25s



Affectation & copie d'objets

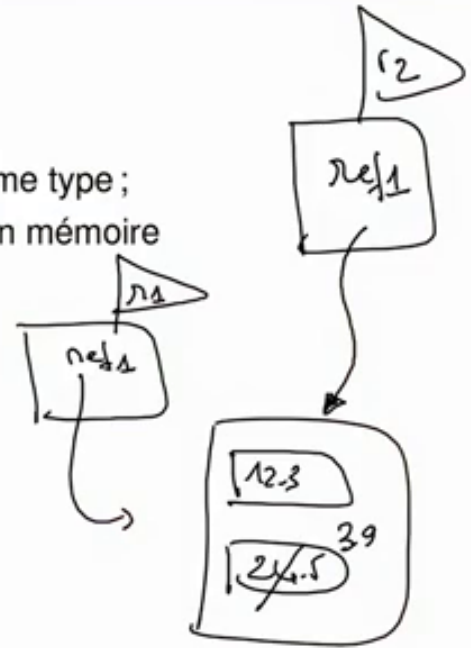
Supposons que l'on souhaite :

- ▶ créer un objet **b** à partir d'un autre objet **a** du même type ;
- ▶ assurer que **a** et **b** soient deux objets **distincts** en mémoire

```
Rectangle r1 = new Rectangle(12.3, 24.5);
Rectangle r2 = r1;
```

r1 et **r2** référencent ici le **même objet**
et non pas deux copies distinctes du même objet :
toute modification via **r2** modifiera également **r1**.

r2.setHauteur(3.9);



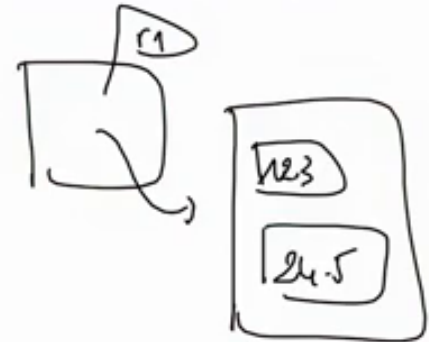
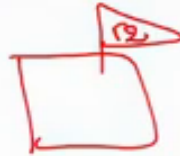
dire concrètement que toute manipulation faite sur l'objet au travers de « r1 » sera également visible au travers de « r2 » et vice-versa. Donc par exemple, si je change la valeur de la hauteur de « r2 », ce sera aussi visible pour « r1 ».

notes

résumé

Si l'on veut que l'objet référencé par **r2** soit une copie distincte de celui référencé par **r1**, il ne faut pas utiliser l'opérateur ~~X~~ mais plutôt un **constructeur de copie** (ou la méthode **clone** qui sera vue plus tard) :

```
Rectangle r1 = new Rectangle(12.3, 24.5);
Rectangle r2 = new Rectangle(r1);
```



Concrètement, ceci veut dire que si l'on souhaite que « r2 » soit une copie distincte de « r1 » alors en Java on n'utilisera pas l'opérateur d'affectation. On utilisera plutôt un constructeur de copie ou alors la méthode clone qui sera vue un petit peu plus tard dans le cadre de ce cours. Alors nous en parlons parce que ce n'est pas systématiquement le cas dans tous les langages. Dans certains langages on peut parfaitement créer deux copies distinctes d'objets au travers de l'affectation. Nous avons vu dans des séquences précédentes comment écrire dans une classe, un constructeur de copie. C'est exactement un constructeur de copies qui est utilisé ici. Donc un constructeur qui crée un rectangle à partir des valeurs d'un autre rectangle. Donc ici si j'exécute cette ligne de code, je suis dans la même situation que tout à l'heure. « r1 » est une référence vers un objet Rectangle particulier. Lorsque j'exécute la seconde ligne, Je vais créer un autre objet Rectangle. Donc un objet distinct en mémoire où je vais simplement recopier, champ-à-champ, les valeurs du premier rectangle. Donc on voit bien ici que l'on n'est pas dans la même situation que tout à l'heure. Nous avons bien « r2 » et « r1 » comme étant deux objets distincts en mémoire. étant deux objets distincts en mémoire.

notes

résumé

2m 49s

