

Support de cours

Cours:

Introduction à la programmation orientée objet (en Java)

Vidéo:

W14-02-polymcomplements-JAVA-pt4

Concepts (extraits des sous-titres générés automatiquement) :

Méthode equals. Type d'objet. Méthodes de l'api de java. Nouveau false. Exemple des méthodes allant. Objet de type string. Difficulté principale. Petite difficulté. Classe rectangle. Nouveaux écueils. Largeur de l'objet courant. Classe de l'instance courante. Largeur de autreobjet. Façon possible. Comparaison d'un rectangle.



[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/>

Héritage et polymorphisme : compléments

(Partie 4)

Introduction à la programmation orientée objet (en Java)

Jamila Sam, Jean-Cédric Chappelier et Vincent Lepetit

...

notes

résumé

0m 0s



L'entête proposée pour la méthode `equals` dans une séquence précédente était :

```
public boolean equals(UneClasse arg)
```

Rectangle

or l'entête de la méthode `equals` dans `Object` est :

```
public boolean equals(Object arg)
```

- ✎ Nos définitions de `equals` constituaient jusqu'ici des **surcharges** et non pas des **redéfinitions** de la méthode `equals` de `Object` !

Dans la plupart des cas, utiliser une surcharge fonctionne sans problème, mais il est recommandé de **toujours procéder par redéfinition**.

Définir la méthode `equals` par surcharge, comme nous l'avons fait dans les séquences précédentes, peut parfaitement fonctionner, simplement en Java, il est plutôt conseillé de procéder par redéfinition, et ce, parce que certaines méthodes de l'API de Java, typiquement des méthodes qui travaillent sur ce qu'on appelle des collections de données, par exemple des méthodes allant chercher une valeur dans une collection, vont utiliser implicitement la méthode `equals` qui est exactement cette entête. Si elle n'est pas présente dans une classe à vous et que vous utilisez cette méthode, elle va utiliser la méthode par défaut héritée de `Object`

notes

résumé

0m 1s



Attention ! si l'on redéfinit `equals` pour la classe `Rectangle`, on doit pouvoir comparer un `Rectangle` avec n'importe quel autre objet :
`unRectangle.equals("toto")` devrait retourner `false`.

```
class Rectangle {
    //...
    public boolean equals(Object autreObjet) {
        if (autreObjet == null)
            { return false; }
        else {
            if (autreObjet.getClass() != getClass())
                { return false; }
            else {
                Rectangle r = (Rectangle)autreObjet;
                return (largeur == r.largeur &&
                    hauteur == r.hauteur);
            }
        }
    }
}
```

boolean equals (Rectangle
 r_1)
 $\{ \dots \}$
 ~~$r_1.equals("toto")$~~
 $r_1.equals("toto")$

qui n'est pas satisfaisante dans la plupart des cas. Alors comment procéder si l'on veut vraiment redéfinir la méthode `equals` héritée de `Object` plutôt que de surcharger cette méthode, nous vous présentons une façon possible de redéfinir la méthode `equals`, une façon assez typique, que vous pouvez rencontrer dans la littérature. Sachez qu'il existe d'autres façons de rédiger cette méthode `equals`, c'est une variante parmi d'autres. La difficulté principale que nous avons à affronter lorsque nous voulons redéfinir la méthode `equals` est que désormais, elle peut prendre en paramètre n'importe quel type d'objet. Lorsque nous utilisons la surcharge, le paramètre était de même type que la classe dans laquelle on définissait la surcharge, donc ici ceci et donc l'invocation de la méthode `equals` sur autre chose qu'un `Rectangle`, par exemple une chaîne de caractères, ferait réagir le compilateur s'il n'y avait que la surcharge : le compilateur vous dirait : " j'attends une chaîne de caractères, et vous me donnez un rectangle ". En revanche, avec une redéfinition, cette écriture devient tout à fait licite, le compilateur ne va rien dire, va parfaitement l'accepter.

notes

résumé

0m 37s



Attention ! si l'on redéfinit `equals` pour la classe `Rectangle`, on doit pouvoir comparer un `Rectangle` avec n'importe quel autre objet :
`unRectangle.equals("toto")` devrait retourner `false`.

```
class Rectangle {  
    //...  
    public boolean equals(Object autreObjet) {  
        if (autreObjet == null)  
            { return false; }  
        else {  
            if (autreObjet.getClass() != getClass())  
                { return false; }  
            else {  
                Rectangle r = (Rectangle)autreObjet;  
                return (largeur == r.largeur &&  
                        hauteur == r.hauteur);  
            }  
        }  
    }  
}
```

~~largeur == autreObjet.largeur~~

Pourquoi ? Parce que un objet de type String est un Object par héritage Et du coup, ceci est parfaitement licite, je peux une String dans un Object. À noter que si l'on ne redéfinit pas la méthode `equals` dans la classe `Rectangle` au moment de faire ce genre d'invocation, on utiliserait la méthode `equals`, telle que définie dans la classe `Object`, laquelle compare uniquement les références et qui ne fait pas forcément les traitements que l'on souhaiterait voir mis en œuvres pour la comparaison d'un rectangle avec un autre objet. C'est donc au programmeur de la méthode `equals` de définir correctement le corps de sa méthode de sorte à ce que la comparaison avec d'autres types d'objets que des rectangles se fasse correctement, typiquement retourne `false`. On a envie qu'un rectangle soit comparable à un autre rectangle, et pas à un objet d'un autre type. Pour garantir qu'un rectangle ne puisse être égal qu'à un autre rectangle strictement parlé, il faut tester si l'objet passé en paramètre est de la même classe que la classe `Rectangle`. Ceci peut se faire au moyen d'une méthode `getClass` à nouveau héritée de `Object`. Par cette tournure, on peut tester si la classe de l'autre objet est la même que la classe de l'instance courante. Et si tel n'est pas le cas, on retourne `false`. Pour résumer, lorsqu'on veut redéfinir la méthode `equals`, on commence, comme nous le faisons pour la surcharge, par tester si le paramètre ne vaut pas la valeur `null` : s'il vaut `null`, on retourne `false`, sinon, là est la nouveauté, on teste si l'autre objet est de même classe que l'objet courant, si tel n'est pas le cas, on retourne à nouveau `false`, Et sinon, s'il est garanti que `autreObjet` n'est pas `null`, que le rectangle passé

notes

résumé

2m 1s



Attention ! si l'on redéfinit `equals` pour la classe `Rectangle`, on doit pouvoir comparer un `Rectangle` avec n'importe quel autre objet : `unRectangle.equals("toto")` devrait retourner `false`.

```
class Rectangle {  
    //...  
    public boolean equals(Object autreObjet) {  
        if (autreObjet == null)  
            { return false; }  
        else {  
            if (autreObjet.getClass() != getClass())  
                { return false; }  
            else {  
                Rectangle r = (Rectangle)autreObjet;  
                return (largeur == r.largeur &&  
                        hauteur == r.hauteur);  
            }  
        }  
    }  
}
```

~~largeur == autreObjet.largeur~~

en paramètre est bien de classe `Rectangle`, on peut procéder à la comparaison attribut par attribut, comme nous le faisons dans le cadre de la surcharge. Mais là, nous avons de nouveau une petite difficulté à affronter : en effet, si je veux comparer attribut par attribut, il faut que je teste si la largeur de l'objet courant vaut la même valeur que la largeur de `autreObjet`, et que la hauteur de l'objet courant vaille la même valeur que la hauteur de l'autre objet. Or, `autreObjet` est de type `Object`, ce qui ne garanti pas la présence des attributs `largeur` et `hauteur`. Nous savons bien que `autreObjet` contient bel et bien un `Rectangle`, la précaution que nous avons prise ici nous le garanti. Cependant, si l'on écrit quelque chose comme ceci, donc si l'on essaie de comparer la largeur de l'objet courant avec la largeur de l'autre objet, en utilisant directement cette tournure, le compilateur va émettre un message d'erreur, en vous disant que `autreObjet` est de type `Object`, lequel ne contient pas de champs `largeur`.

notes

résumé



Ici, nous convertissons autreObjet en Rectangle, on parle de transtypage, pour garantir au compilateur qu'il est possible d'accéder au champs largeur. Ceci va parfaitement fonctionner ici, parce que nous avons bel et bien garanti dans la chaîne d'exécution que autreObjet va contenir un objet de type rectangle. Si l'on essaie de faire un transtypage, alors que autreObjet ne contient pas un Rectangle, on aurait un message d'erreur au moment de l'exécution. Mais ça n'est pas le cas ici. Donc, globalement voici comment on peut s'y prendre pour redéfinir la méthode equals avec les nouveaux écueils qui se présentent lorsqu'on fait de la redéfinition plutôt que de la surcharge. Et ceci termine notre petite séquence de complément.

notes

résumé

4m 49s

