Ecriture des classes (suite)
Les méthodes polymorphes

Sommaire :

1 Introduction

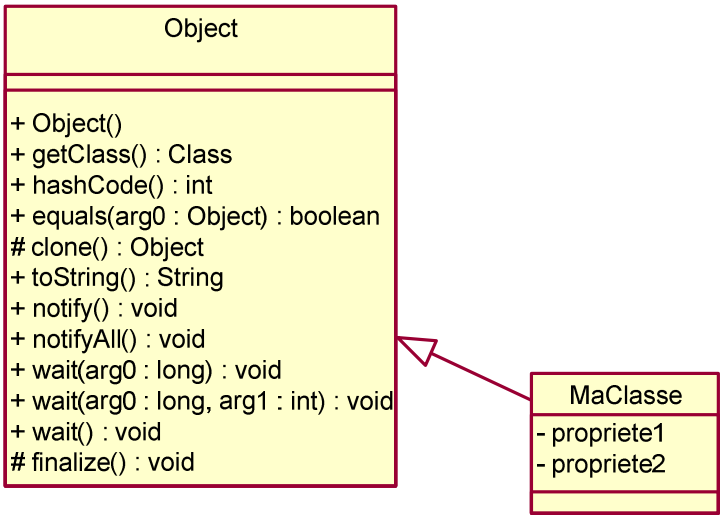
Manipulation des textes, des dates, calcul mathématique, gestion des ensembles d'objets: les classes présentées ici illustrent les choix réalisés par les inventeurs de Java pour ses concepts fondamentaux.
La bibliothèque Java contient des classes qu'il faut absolument connaître, telle java.lang.Object sur laquelle repose toute la hiérarchie des classes Java, java.lang.String qui représente des chaînes de caractères, ou java.lang.System utilisée pour communiquer avec la MV. Outre ces classe incontournables, les classes utilitaires pour gérer la date, l'heure et les ensembles d'objets s'avèrent indispensables pour programmer des applications.

2 La classe java.lang.Object

Java utilise une hiérarchie unique pour toutes les classes. La classe Object est la super-classe de cette hiérarchie.
En java, la relation d'héritage avec Object est implicite :

`class MaClasse` équivaut à écrire : **`class MaClasse extends Object`**

Le diagramme de classes suivant présente la classe Object (J2SE 1.3):



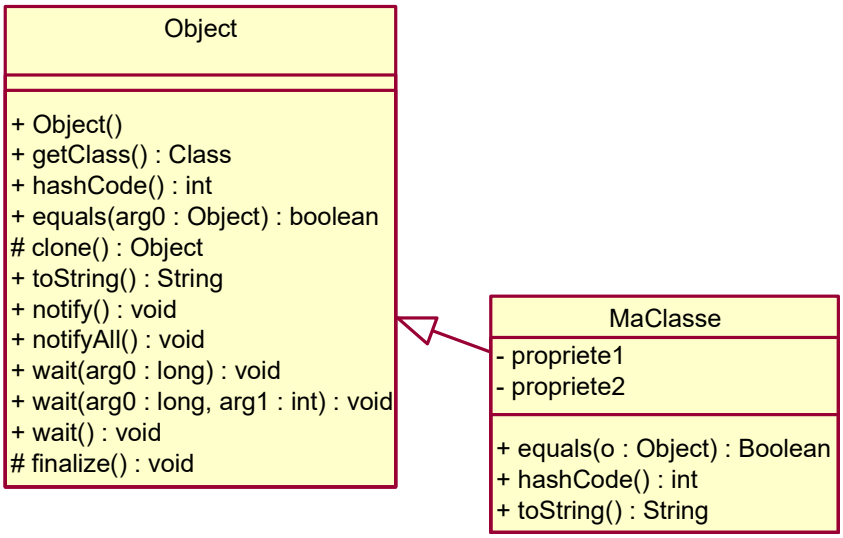
2.1 Les méthodes d'instance polymorphes

Dans une classe quelconque, on peut utiliser telles quelles, les méthodes d'Object (héritage) ou on peut en modifier l'implémentation c'est-à-dire redéfinir (overriding)ces méthodes. Dans ce dernier cas, si on utilise une référence de type Object , le polymorphisme entrera en jeu : ces méthodes sont dites « polymorphes ».

Les méthodes les plus souvent redéfinies sont les suivantes :

Méthode polymorphe	explication
toString	renvoie une chaîne de caractères décrivant l'état de l'objet courant
equals	renvoie true si deux objets distincts sont égaux
hashCode	renvoie un code de « hachage » entier

Les méthodes equals, hashCode et toString sont souvent redéfinies dans les classes de l'API Java (String, java.lang.Float, ...) et c'est en les utilisant que vous percevrez mieux pourquoi les concepteurs de Java ont choisi de déclarer ces méthodes dans la classe java.lang.Object. Notamment, equals et hashCode sont redéfinies pour comparer les objets ou les retrouver dans une collection d'objets.
Le diagramme de classes suivant présente les méthodes polymorphes (J2SE 1.3):



2.2 Autres méthodes d'Object

méthode	explications
getClass	renvoie une instance de la classe java.lang.Class qui conceptualise la classe d'un objet
wait et notify	utilisées pour la synchronisation des threads
clone	crée une copie d'un objet
finalize	redéfinie dans une classe pour décrire les traitements spécifiques à effectuer à la destruction d'un objet par le ramasse-miettes

3 Ecriture de vos classes en Java

Lorsque vous développerez vos propres classes, vous redéfinirez les méthodes equals, hashCode et toString ; si nécessaire, vous implémenterez l'interface Comparable.

3.1 Redéfinir la méthode toString

A la base, la méthode toString d'Object renvoie une chaîne de caractères contenant la classe de l'objet courant suivi du caractère @ et de la valeur en hexadécimal renvoyée par hashCode (par exemple org.gnu.tests.Test@6a55ra).

Dans les classes métiers, cette méthode est souvent redéfinie pour renvoyer un texte (debug) décrivant l'état de l'objet c'est-à-dire un texte avec la valeur de ses champs.

Rq: Pour une concaténation de chaîne (opérateur « + »), si l'un des opérandes est un objet alors toString est appelée pour placer la forme textuelle de cet objet.

Voici la signature de toString:

```
public String toString ()
```

3.2 Redéfinir la méthode equals

La méthode equals renvoie « true » si deux objets distincts sont considérés comme « égaux ». A la base, la méthode equals d'Object ne fait que comparer deux objets à l'aide de l'opérateur = = (toujours false pour deux objets distincts).
Dans les classes métiers, cette méthode equals est souvent redéfinie car, pour l'égalité, il existe une logique qui très souvent est basée sur l'égalité de certains champs. Par exemple pour la classe String, deux chaînes sont égales si leurs caractères sont égaux en les comparant un à un.

Après avoir écrit equals, une collections de type Set va automatiquement vous empêcher de stocker des objets égaux (doublons).

Voici la signature de equals : `public boolean equals(Object obj)`
La relation d'égalité définie entre deux objets doit être alors réflexive, commutative et transitive.

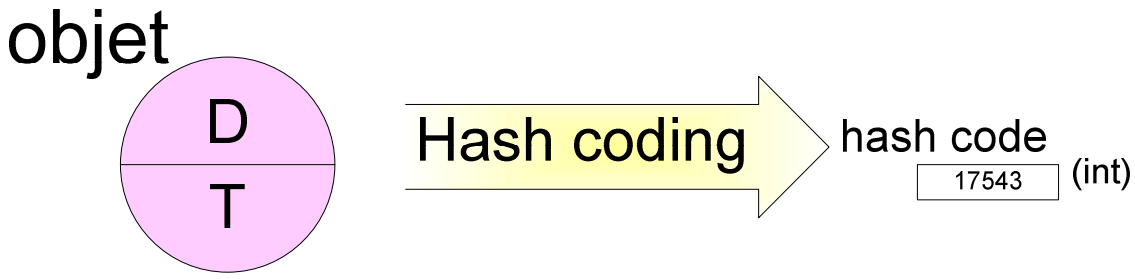
Attention : dans vos classes, si vous redéfinissez equals alors vous devez aussi redéfinir hashcode en suivant la même logique métier (Voir hashCode plus loin).

3.3 Bien redéfinir equals …

Si vous redéfinissez (overriding) la méthode equals, faites attention à sa signature. Déclarez l'unique paramètre comme type Object, même si vous savez pertinemment que seuls deux objets de la même classe sont comparables. Si le paramètre n'était pas de type Object, la méthode equals serait surchargées (overloading), ce qui n'est pas interdit, mais alors, vous auriez un disfonctionnement sur les services de stockage dans certaines collections (Set).

Concernant l'écriture de equals, le type Object vous oblige à utiliser l'opérateur instanceof pour vérifier la nature de l'objet avant de comparer les champs.

3.4 Redéfinir la méthode hashCode



La finalité de la méthode hashCode est de fournir pour chaque objet, un entier dit « code de hachage » qui est utilisé pour le stockage et la récupération dans certaines collections (HashSet, HashMap, TreeSet,...,etc).
La méthode hashCode utilise un algorithme rapide et de bonne qualité statistique.
Voici la signature de hashCode: `public int hashCode()`

Règle pour l'écriture de hashCode :

Règle: si deux objets sont égaux au sens d'equals alors hashcode doit retourner le même entier.

La condition inverse n'est pas obligatoire : deux objets peuvent renvoyer un code de hachage identique tout en étant différents par la méthode equals.
Donc, pour deux références obj1 et obj2 désignant deux objets quelconques, l'algorithme suivant doit être vérifiés : *si (obj1.equals(obj2) renvoie true) alors obj1.hashCode() == obj2.hashCode() est égal à true*

D'ou, la règle précédente nous amène à la règle suivantes :

Règle : Dans vos classes métier, si vous redéfinissez equals alors vous devez aussi redéfinir hashcode en suivant la même logique métier.

3.5 Explications supplémentaires sur le code de hachage

3.5.1 Utilisation pour les collections

Dans certaines collections, le code de hachage est utilisé pour ranger un ensemble d'objets et optimiser les recherches dans cet ensemble.
Une recherche dans ce type d'ensemble est comparable à la démarche effectuée pour consulter un dictionnaire pourvu d'onglets pour chaque lettre de l'alphabet. Si vous cherchez le mot Java vous saisirez alors directement l'onglet J pour tomber sur le premier mot commençant par J.

Pour assurer une efficacité maximale à ce système de classement, les codes de hachage des objets doivent être répartis sur l'ensemble des entiers int (voir explications suivantes).

3.6 Qualité d'un algorithme de hachage

(domaine des statistiques)

Les qualités d'un algorithme de hachage sont la vitesse et une bonne répartition de ses résultats sur l'axe des entiers.
Pour l'algorithme parfait et sur un grand échantillons d'objets, si on calcule le hashcode et si on cumule le nombre d'objets pour chaque entier trouvé, n obtiendrait le graphique suivant :