

En cours

Polymorphisme

Résolution
dynamique des
liens

Héritage et
comparaison
d'objets

Héritage :
conseils

Le modificateur
`final`

Introduction à la Programmation : Polymorphisme Modificateurs

Laboratoire d'Intelligence Artificielle
Faculté I&C

Objectifs du cours de la semaine

- ▶ Introduire un peu plus formellement la notion de **polymorphisme**
- ▶ Expliquer les techniques de mise en oeuvre du polymorphisme en Java :
 - ▶ Héritage des types.
 - ▶ Résolution dynamique des liens.
 - ▶ Classes et méthodes **abstraites**.
- ▶ Revenir à la comparaison d'objets dans le contexte de l'héritage
- ▶ Introduire le modificateur `final`.

Pendant l'heure de cours

- ▶ Petit rappel des points importants
- ▶ Approfondissements :
 - ▶ A propos de `instanceof` transparent 45
 - ▶ Comparaison dans le cadre de l'héritage transparent 49
 - ▶ A quoi s'applique la résolution dynamique des liens ?
 - ▶ Quid du polymorphisme sur les paramètres de méthodes ?
 - ▶ (Héritage et tableaux)

Polymorphismes

En programmation, on distingue deux types de polymorphismes :

- ▶ le **polymorphisme des traitements** (ou ad hoc)
 - ☞ mécanisme de **surcharge** des fonctions/méthodes : le même identificateur est utilisé pour désigner des séquences d'instructions différentes
- ▶ le **polymorphisme des données** (ou universel)
 - ▶ le polymorphisme **d'inclusion** :
le même code peut être appliqué à des données de types différents liés entre eux par une relation de sous-typage
 - ☞ hiérarchies de classes
 - ▶ le polymorphisme **paramétrique** :
le même code peut être appliqué à n'importe quel type (généricité)
 - ☞ Cours sur les *modèles de classes* au semestre de printemps

Héritage (rappels)

Rappel du dernier cours :

- ▶ Dans une hiérarchie de classes, la *sous-classe hérite de la super-classe* :
 - ▶ tous les attributs/méthodes (sauf constructeurs et destructeur)
 - ▶ le type
- ▶ L'héritage est transitif

Héritage du type :

- ▶ on peut affecter un objet de type sous-classe à une variable de type super-classe

```
Rectangle r = new Rectangle(12.5, 2.0);  
FigureGeometrique fig = r;
```

Héritage : « est-un » : héritage du type

```
class A {
public
    A(int x )
        {a = x;}
    int getA() ...
private int a;
}
```

```
class B extends A {
public
    B(int x, int y)
        {super(x); b = y;}
private int b;
}
```

```
static void affiche(A unA)
    System.out.print( "Valeur: ");
    System.out.println(unA.getA());
}
```

```
public static void main(...) {
    B bb = new B(4, 3);
    A aa; aa=bb;
    affiche(aa);
    affiche(bb);
}
```

Valeur: 4
Valeur: 4

Les instances de `B` sont substituables aux instances de `A`

👉 **Compatibilité ascendante des types**

Polymorphisme d'inclusion

En POO, le **polymorphisme d'inclusion** est le fait que :

- ▶ les instances d'une sous-classe soient **substituables** aux instances des classes de leur ascendance, *en argument d'une méthode ou lors d'affectations*, tout en **gardant leurs nature/propriétés propre(s)**.

La mise en œuvre se fait par :

- ▶ les mécanismes de l'**héritage** dans les hiérarchies de classes,
- ▶ la **résolution dynamique des liens** :
 - ▶ le choix des méthodes à invoquer se fait *lors de l'exécution du programme* en fonction de la *nature réelle des instances* concernées

Résolution des liens

Une instance de sous-classe **B** est substituable à une instance de super-classe **A**.



Que se passe-t-il lorsque **B** redéfinit une méthode de **A** ?

```
class A {  
    public  
    void afficher() {  
        System.out.println("A");  
    }  
}
```

```
class B extends A {  
    public  
    void afficher() {  
        System.out.println("B");  
    }  
}
```

```
public.. main(..) {  
    A aa;  
    B bb = new B();  
    aa = bb;  
    aa.afficher();  
}
```

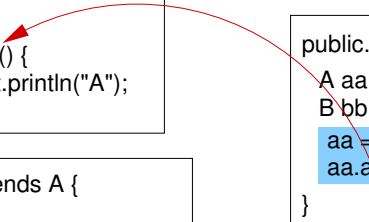
Résolution statique des liens

Première possibilité : c'est le **type de la variable** qui **détermine la méthode** à exécuter :

```
class A {  
    public  
    void afficher() {  
        System.out.println("A");  
    }  
}
```

```
class B extends A {  
    public  
    void afficher() {  
        System.out.println("B");  
    }  
}
```

```
public.. main(..) {  
    A aa;  
    B bb = new B();  
    aa = bb;  
    aa.afficher();  
}
```



Résolution statique des liens

Résolution dynamique des liens

Deuxième possibilité : le choix de la méthode se fait selon à la *nature réelle de l'instance* contenue dans la variable :

```
class A {  
    public  
    void afficher() {  
        System.out.println("A");  
    }  
}
```

```
class B extends A {  
    public  
    void afficher() {  
        System.out.println("B");  
    }  
}
```

```
public.. main(..) {  
    A aa;  
    B bb = new B();  
    aa = bb;  
    aa.afficher();  
}
```

☞ Java met systématiquement en oeuvre la **résolution dynamique des liens**

Rappel : les avatars rencontrent le joueur

```
public static void main(String[] args)
{
    Joueur leJoueur = new Joueur(...);
    Avatar[] lesAvatars = new Avatar[3];

    lesAvatars[0] = new Elfe(...); // Correct ?
    lesAvatars[1] = new Orc(...);
    lesAvatars[2] = new Sorcier(...);

    for (int i=0; i < lesAvatars.length; ++i)
    {
        lesAvatars[i].rencontrer(leJoueur);
    }
}
```

- 👉 le tableau `lesAvatars` est une *collection hétérogène* d'`Avatar`. La méthode `rencontrer(Joueur)` s'adapte automatiquement à la nature réelle de l'`Avatar` grâce à la résolution dynamique des liens

Méthodes et classes abstraites

Au sommet d'une hiérarchie de classe, il n'est pas toujours possible de :

- ▶ donner une définition générale de certaines méthodes, compatibles avec toutes les sous-classes,
- ▶ ...même si l'on sait que toutes ces sous-classes vont effectivement implémenter ces méthodes

Méthodes et classes abstraites (2)

Exemple :

```
class FigureFermee {  
    //....  
  
    // difficile a definir a ce niveau !..  
    // comment ecrire le corps?  
    public double surface() {///  
        }  
  
    // ...pourtant la methode suivante en aurait besoin !  
    public double volumeCylindre const (double hauteur) {  
        return (hauteur * surface());  
    }  
    //....  
}
```



Comment définir `surface` pour des figures géométriques quelconques ?

Définir `surface` de façon arbitraire sachant que l'on va la redéfinir plus tard n'est pas une bonne solution (source d'erreurs) !

👉 Il faut déclarer la méthode `surface` comme **abstraite**

Méthodes abstraites

Une méthode *abstraite*, est *incomplètement spécifiée* :

- ▶ elle n'a *pas de définition* dans la classe où elle est introduite (pas de corps)
- ▶ elle sert à signaler aux sous-classes qu'elles **doivent redéfinir** la méthode virtuelle héritée

Syntaxe :

```
abstract Type nom_methode(liste d'arguments);
```

Exemple :

```
abstract class FigureFermee {  
    public abstract double surface();  
    public abstract double perimetre();  
    //...  
}
```

Classes abstraites

Règles :

1. Une méthode abstraite doit être déclarée dans une classe abstraite
2. Les classes et méthodes abstraites sont déclarées au moyen du modificateur `abstract`
3. le modificateur `abstract` signale :
 - ▶ que la méthode ne peut être appelée elle-même mais doit exister dans toutes les sous-classes que l'on veut pouvoir instancier
 - ▶ qu'une classe ne peut être instanciée

Les classes et méthodes abstraites permettent de :

- ▶ rendre compilable un `programme incomplet` sans fausser le modèle
- ▶ clarifier les intentions du programmeur

Note : une classe abstraite peut tout à fait hériter d'une classe non abstraite.

Classe/méthode abstraite

- ▶ A quoi sert une **classe abstraite** ?
 - ▶ Le compilateur rappellera le programmeur de ses intentions initiales
 - ▶ Modèle plus clair
 - ▶ Indispensable si l'on souhaite utiliser des méthodes abstraites
- ▶ A quoi sert une **méthode abstraite** ?
 - ▶ Impose des conditions sur les sous-classes
 - ▶ Rend une classe compilable même si l'implémentation d'une méthode particulière manque (son corps)
 - ▶ Indissociable de la notion de polymorphisme

Exemple : classe abstraite (1)

Attention : Si une classe est déclarée `abstract`, il est impossible de l'instancier !

```
abstract class Avatar
{
    private int energie;
    private int dureeVie;
    public Avatar() {
        // initialisation des attributs
        // avec des valeurs par défaut
    }

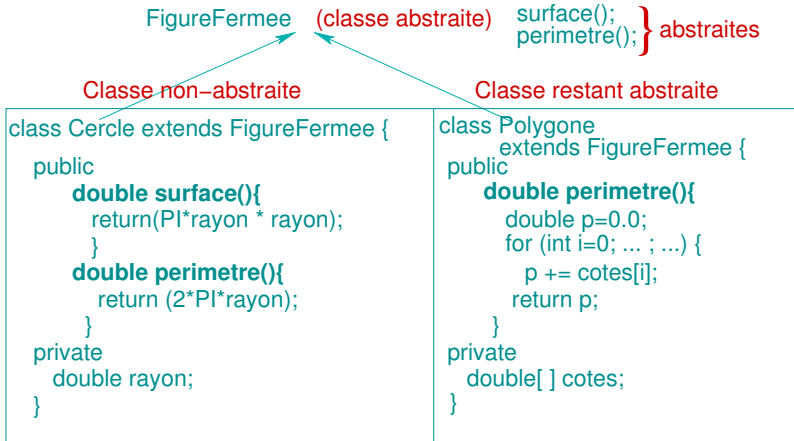
    // ailleurs:

    Avatar unAvatar = new Avatar();
}
```

Exemple de message d'erreur :

```
Avatar unAvatar = new Avatar();
Avatar is abstract; cannot be instantiated
```

Exemple : classes abstraites (2)



Exemple commercial

Vos fins de mois sont difficiles et vous décidez de rentabiliser votre jeu.

- ▶ **But** : vendre 2 classes destinées à programmer un jeu **sans divulguer les sources**

1. **Jeu** :

- ▶ Classe pour gérer le jeu
- ▶ Se contente ici d'afficher les personnages (avatars)
- ▶ L'acheteur appellera les méthodes de la classe, pas d'accès aux variables d'instance

2. **Avatar** :

- ▶ Classe de base pour les avatars
- ▶ L'acheteur ajoutera des sous-classes à sa guise

Matériel à vendre

La classe `Jeu` vendue sous forme compilée :

☞ S'occupe de gérer un tableau d'avatar et de les afficher

```
class Jeu {  
    private Avatar[] avatars;  
    public Jeu(int taille) {  
        avatars = new Avatar[taille];  
    }  
  
    public void ajouterAvatar(Avatar a) {  
        // ajoute un avatar au tableau  
    }  
  
    public void afficher() {  
        for (int i = 0; i < avatars.length; i++)  
            avatars[i].afficher();  
    }  
}
```

Matériel à vendre (2)

La classe `Avatar` vendue sous forme compilée :

- ▶ Super-classe pour les avatars
- ▶ A étendre par l'acheteur

```
abstract class Avatar {  
    String nom;  
    int energie;  
    int dureeVie;  
    Avatar (String unNom, int) {  
        nom = unNom;  
        // reste des initialisations  
    }  
  
    // M'ethode afficher(..) ???  
}
```

☞ La classe `Avatar` est abstraite, l'utilisateur devra ajouter ses propres sous-classes



Comment imposer que la méthode `afficher` soit définie dans les sous-classes introduites par l'utilisateur ?

Méthode afficher

Afin de vendre `Jeu` sous forme compilée :

- ▶ La classe `Avatar` doit avoir une méthode `afficher`
- ▶ Sinon message d'erreur :

```
Cannot resolve symbol  
symbol : method afficher()  
avatar[i].afficher();
```

- ▶ On aimerait que l'utilisateur ajoute ses propres méthodes `afficher` à ses sous-classes
- ▶ La résolution dynamique des liens garantira que c'est la méthode de la sous-classe qui sera exécutée

Mauvaise solution

Solution 1 : Ajouter une méthode quelconque

```
abstract class Avatar {  
    String nom;  
    int energie;  
    int dureeVie;  
    Avatar (String unNom, int) {  
        nom = unNom;  
        // reste des initialisations  
    }  
  
    void afficher() {  
        return;  
    }  
}
```

C'est une **très mauvaise idée**

- ➡ Affichage incorrect si une sous-classe "oublie" de redéfinir la méthode (avatars fantômes !!)
- ➡ Mauvais modèle de la réalité

Bonne solution

Solution 2 : Signaler que la méthode doit exister dans *chaque* sous-classe

☞ Déclarer la méthode comme **abstraite**

Syntaxe :

- ▶ `abstract` + en-tête + ;
- ▶ Pas d'instructions, même pas d'accolades

```
abstract class Avatar {  
    String nom;  
    int energie;  
    int dureeVie;  
    Avatar (String unNom) {  
        nom = unNom;  
        // reste des initialisations  
    }  
    abstract void afficher();  
}
```

Méthode abstraite (résumé)

Si l'on ajoute `abstract` à une méthode :

1. La méthode ne peut pas avoir d'instructions
2. La méthode sert seulement à signaler aux sous-classes qu'elles doivent redéfinir la méthode abstraite héritée
3. La classe qui contient la méthode abstraite doit elle-même être abstraite
4. Une sous-classe qui ne redéfinit pas une méthode abstraite héritée doit elle-même être abstraite

Une méthode abstraite `n'est jamais appelée` :

1. Elle se trouve toujours dans une classe abstraite qui n'a donc pas d'instances
2. Si elle est héritée par une classe non-abstraite, cette classe doit redéfinir la méthode
3. Elle n'a de toute façon pas d'instructions

Un client (si si il y en a) :

1. achète les classes compilées
`Avatar` et `Jeu`,
2. ajoute ses propres sous-classes d'`Avatar` : `Guerrier`,
`Cavalier`, etc
3. écrit sa propre méthode `main` :
classe `JeuClient`

Ajout de la classe Cavalier

Votre client ajoute une sous-classe `Cavalier` à `Avatar` :

```
class Cavalier extends Avatar {  
    Cheval monture;  
    Cavalier(String unNom) {  
        super(unNom);  
        // reste des initialisations  
    }  
    void afficher() {  
        // code pour afficher specifiquement  
        // le cavalier  
    }  
}
```

Message d'erreur en cas d'oubli de la méthode `afficher` :

```
class Cavalier should be declared abstract;  
it does not define afficher() in Avatar
```

Le client fait la même chose pour toutes ses sous-classes d'avatars.

Classe JeuClient

Cette classe contient la méthode `main` de votre client :

- ▶ Construction d'un objet `Jeu`
- ▶ Remplissage du tableau par les différents avatars
- ▶ Pas d'accès direct au tableau (car privé)
- ▶ Utilisation de l'API (les méthodes fournies)

Classe JeuClient (2)

Code :

```
class JeuClient {  
    public static void main(String[] args) {  
        Jeu monJeu = new Jeu(2);  
        monJeu.ajouterAvatar(new Cavalier(...));  
        monJeu.ajouterAvatar(new Guerrier(...));  
  
        // a la place de afficher on peut imaginer  
        // des methodes plus sophistiquee comme la gestion  
        // de la rencontre avec un joueur etc..  
        monJeu.afficher();  
    }  
}
```

Résumé de l'exemple

1. Classes fournies sous forme compilée :
 - ▶ `Jeu` assure la logistique de base du jeu
 - ▶ `Avatar` pour modéliser des avatars abstraits
2. Classes ajoutées par l'acheteur :
 - ▶ `Guerrier`, `Cavalier`, etc pour représenter ses propres avatars selon le modèle de la classe `Avatar`
 - ▶ `JeuClient` pour créer son instance spécifique de la classe `Jeu`
 - ▶ Ecriture du code facilitée (des parties difficiles et/ou fastidieuses ont été achetées)
3. La partie vendue n'aurait pas compilé sans `abstract` pour `afficher` et `Avatar`

Constructeurs et polymorphisme

Un constructeur est une méthode spécifiquement dédiée à la construction de l'**instance courante** d'une classe, il n'est pas prévu qu'il ait un comportement polymorphique.

Il est cependant possible d'invoquer une méthode polymorphique dans le corps d'un constructeur

- ☞ Ceci est cependant **déconseillé** : la méthode agit sur un objet qui n'est peut-être alors que partiellement initialisé !

Un exemple ...

Constructeurs et polymorphisme (2)

```
abstract class A {
    public abstract void m();
    public A() {
        m(); // m'ethode invocable de maniere polymorphique
    }
}

class B extends A {
    private int b;
    public B() {
        b = 1; // A() est invoquee implicitement juste avant
    }
    public void m() { // d'efinition de m pour la classe B
        System.out.println("b vaut : " + b);
    }
}

// .... dans le programme principal:
B b = new B(); // la m'ethode m invoquee dans A() est
// ici celle de la classe B (polymorphisme)
// affiche: b vaut 0
// (au moment o'u m() est invoquee, b = 1
// dans B() n'est pas encore executee !)
// voir : ordre d'appel des constructeurs (cours 7)
```

La super-classe `Object`

Il existe en Java une super-classe commune à toutes les classes : la classe `Object` qui constitue **le sommet de la hiérarchie**

Toute classe que vous définissez, si elle n'hérite d'aucune classe explicitement, dérive de `Object`

Il est donc possible d'affecter une instance de n'importe quelle classe à une variable de type `Object` :

```
Object v = new UneClasse (...); // OK
```

La super-classe `Object` (2)

La classe `Object` définit, entre autres, les méthodes :

- ▶ `toString` : qui affiche juste une représentation de l'adresse de l'objet ;
- ▶ `equals` : qui fait une comparaison au moyen de `==` (comparaison des références) ;
- ▶ `clone` : qui permet de copier l'instance courante (nous y reviendrons !)

Dans la plupart des cas, ces définitions par défaut ne sont pas satisfaisantes quand vous définissez vos propres classes

- ☞ Vous êtes amenés à les **masquer** (redéfinir) pour permettre un affichage, une comparaison (et plus tard un “clonage”) corrects de vos objets

[C'est en fait (presque) ce que nous faisons dans le cours 7 dans nos exemples avec `equals` et `toString`]

La classe `String` redéfinit aussi évidemment ces méthodes.

La méthode `equals` dans le contexte de l'héritage

Pour une définition correcte de la méthode `equals`, un certain nombre de précautions doivent être prises.

Leur mise en oeuvre nécessite quelques connaissances sur la notion de `transtypage` dans une hiérarchie de classes.

Nous allons donc commencer par un petit détour sur cette notion

...

Typage fort

Java est un langage à typage fort :

- ▶ Il faut donc toujours respecter les types des variables (affectations, arguments de méthodes, accès aux membres d'une classe)

- ▶ Exemple de l'affectation :

`type à gauche = type à droite`

1. Affectation permmissible grâce à l'héritage de type :

`variable super-classe = variable sous-classe;`

2. Affectation non-permissible, même si la variable de type super-classe contient un objet de type sous-classe :

`variable sous-classe = variable super-classe; //NON!!`

Typage fort (2)

Exemple :

- Extraction d'un objet `Elfe` du tableau `avatars` :

```
for (int i = 0; i < avatars.length; i++)  
    if (avatars[i] instanceof Elfe) {  
        Elfe uneElfe = avatars[i]; // illicite!!  
        //... manipulation de uneElfe  
    }
```

- Message d'erreur lors de la compilation :

Incompatible types

found : Avatar

required: Elfe

```
Elfe uneElfe = avatars[i];
```



Pourquoi donc ?

Transtypage et affectation

Examinons ce qui s'est passé avec l'instruction :

```
Elfe uneElfe = avatars[i];
```

- ▶ Affectation `sous-classe = super-classe`
- ▶ Le *compilateur* voit que `avatars[i]` est déclarée comme un `Avatar`
- ▶ C'est seulement à l'*exécution* qu'on peut voir qu'il s'agit d'un avatar de type `Elfe`
- ▶ Si `avatars[i]` est un `Orc`, le programme causera une erreur à l'exécution
- ▶ Le compilé doit être aussi sûr que possible,
👉 l'affectation ne peut donc pas être permise

Transtypage et affectation (2)

Solution au problème précédent :

- ▶ Le programmeur prend la responsabilité du type de l'objet
- ▶ Il doit assurer au compilateur que `avatars[i]` est vraiment un objet de type `Elfe`

Transtypage explicite : spécification explicite du type d'une variable existante

Syntaxe : `(type) variable`

Transtypage et accès aux membres

Une situation similaire à la précédente :

- ▶ Accès à une variable/méthode à l'aide d'une variable de type super-classe

Exemple :

```
if (avatars[i] instanceof Orc)
    avatars[i].frapper(pauvreJoueur); // illicite!!
                                     // meme si frapper est
                                     // une methode de Orc
```

Un message d'erreur lors de la compilation :

```
Cannot resolve symbol
symbol   : method frapper
location: class Avatar
    avatars[i].frapper(pauvreJoueur);
```



Pourquoi ça ne marche pas ?

- ▶ Le *compilateur* voit que `avatars[i]` est déclarée comme un `Avatar`
- ▶ Il n'est donc pas sûr que `frapper` existe pour l'objet

Transtypage et accès aux membres (2)

Code correct :

```
if (avatars[i] instanceof Orc)
    ((Orc) avatars[i]).frapper(pauvreJoueur);
```

Note : le transtypage explicite n'est nécessaire que si le compilateur ne peut pas être sûr de l'existence de la variable/méthode accédée

Exemple :

- ▶ Accès à l'attribut `dureeVie` via un objet de type `Orc`
- ▶ Aucun problème de compilation, car `dureeVie` est déclarée dans la super-classe `Avatar`

Mise en garde

Les exemples figurant sur les deux transparents suivants illustrent dans quelles situations un transtypage s'avère formellement nécessaire.

Il ne constituent en aucun cas **des règles d'utilisation à suivre sans discernement** (nous y reviendrons plus tard).

Résumé

```
class SuperClasse {
    SuperClasse() {
    }
}

class SousClasse extends SuperClasse {
    int v;
    SousClasse() {
        super();
    }
    void m1() {
    }
}

class ExplicitCast {
    public static void main(String[] args) {
        SuperClasse sup = new SousClasse();
        // Affectation d'une variable SuperClasse (objet
        // SousClasse) 'a une variable SousClasse:
        SousClasse sous;
        sous = sup; // faux
        s = (SousClasse)sup; // OK
    }
}
```

Résumé (2)

```
// Variable SuperClasse(objet SousClasse)
// apparî'ee avec un param'etre SousClasse:
    m2(sup); // faux
    m2((SousClasse) sup); // OK
// Acc'es 'a une variable d'instance d'eclar'ee
// dans SousClasse mais pas dans SuperClasse:
    System.out.print(sup.v); // faux
    System.out.print(((SousClasse) sup).v); // OK
// Acc'es 'a une m'ethode d'instance d'eclar'ee
// dans SousClasse mais pas dans SuperClasse:
    sup.m1(); // faux
    ((SousClasse) sup).m1(); // OK
}
static void m2 (SousClasse sous2) {
}
}
```

Au sujet du test de types

Attention : si l'on teste le type d'un objet, par exemple avec `instanceof`, pour choisir quelle méthode lui appliquer :

```
if (forme instanceof Cercle)
    (Cercle) forme.afficherCercle();
else if (forme instanceof Rectangle)
    (Rectangle) forme.afficherRectangle();
//....
```

C'est que l'on n'a pas compris que le polymorphisme est l'un des avantages principaux de la POO !

Sauriez-vous dire pourquoi (dans le cas de cet exemple) ?

- ☞ (A propos du test de types, voir aussi un certain nombre d'exercices proposés en TP.)

Façon correcte de définir `equals` (1)

Revenons maintenant à la définition de la méthode `equals`.

L'entête proposée pour la méthode `equals` dans le cours 7 était :

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

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

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

☞ Nos définitions de `equals` constituaient jusqu'ici des **surcharges** (overloading) et non pas des **redéfinitions** (overriding) de la méthode `equals` de `Object` !

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

Surcharge vs Redéfinition (résumé)

On redéfinit ("override") une méthode d'instance si les types des paramètres sont identiques :

```
public boolean equals(Object o)
```

Sinon, on l'a surcharge ("overload") :

```
public boolean equals(UneClasse uc)
```

Façon correcte de définir `equals`(2)

Exemple revisité : 1ère tentative

```
class Avatar {  
    //...  
    // Ici on redefinit equals (et non pas surcharge)  
    public boolean equals(Object autreAvatar) {  
        if (autreAvatar == null) {  
            return false;  
        }  
        else  
            return (nom.equals(((Avatar) autreAvatar).nom));  
    }  
}
```

👉 Le transtypage de `autreAvatar` au type `Avatar` est devenu nécessaire ici !

Attention ! on souhaite maintenant pouvoir comparer un `Avatar` avec n'importe quel autre objet : `unAvatar.equals("toto")` devrait retourner `false`. Or, ces instructions compilent mais produiront une erreur à l'exécution !

Façon correcte de définir `equals`(3)

Il faudrait donc s'assurer que les objets à comparer sont bien des objets de la même classe !

👉 méthode `getClass()` de la classe `Object`

`getClass()` retourne une représentation de la classe de `this` (de telles représentations sont comparables avec `==` ou `!=`)

Exemple revisité : 2ème (et bonne) tentative

```
class Avatar {  
    //...  
    public boolean equals(Object autreAvatar) {  
        if (autreAvatar == null)  
            return false;  
        else  
            if (autreAvatar.getClass() != getClass())  
                return false;  
            else  
                return (nom.equals((Avatar) autreAvatar.nom));  
    }  
}
```

getClass et instanceof

Plus précisément, `getClass` retourne (une représentation de) la classe dont le constructeur a été utilisé pour créer `this` :

```
Avatar a1 = new Avatar(...);
Avatar a2 = new Elfe(...); // Elfe est une
                           // sous-classe de Avatar

System.out.println((a1 instanceof Avatar)
                   && (a2 instanceof Avatar)); // true
System.out.println((a1.getClass() == a2.getClass())); // false
```

getClass et instanceof (2)

Utiliser `instanceof` au lieu de `getClass` dans l'implémentation de `equals` :

```
....  
// dans equals de Avatar :  
if (! (autreAvatar instanceof Avatar))  
  
....  
// et  
  
....  
// dans equals de Elfe :  
if (! (autreAvatar instanceof Elfe))
```

est déconseillé car cela rendrait des instances de sous-classes possiblement “égales” à des instances de super-classes :

```
Avatar a1 = new Avatar(...);  
Elfe a2 = new Elfe(..);  
a1.equals(a2); // peut retourner true  
a2.equals(a1); // retourne toujours false.
```

Héritage : quelques conseils de conception

- ▶ Placer les méthodes et attributs communs dans les super-classes
- ▶ Utiliser l'héritage pour modéliser des relations de type **"est-un"**
- ▶ N'utiliser l'héritage que si les méthodes héritées ont un sens pour les sous-classes
- ▶ Ne faites pas de tests sur le type des objets mais **utilisez le polymorphisme**
- ▶ Evitez l'utilisation d'attributs protégés

Le modificateur `final`

Ce modificateur permet d'indiquer les éléments du programme qui ne **doivent pas être modifiés**
Possible pour les classes, variables et méthodes.

Variables finales

Si l'on ajoute `final` à une variable d'instance, une variable locale ou un paramètre :

- 👉 il devient impossible de lui affecter une valeur plus d'une fois

Exemple : variable stockant la constante π ...



A quoi ça sert ?

- ▶ Aide à la programmation
- ▶ Le compilateur empêchera le programmeur d'affecter une valeur par erreur
- ▶ Rend les intentions du programmeur plus claires

Note : Par convention, les variables finales sont en majuscules

- ▶ `PI` plutôt que `pi`
- ▶ `MECHANT_LOUP` plutôt que `mechantLoup`

Variables finales (2)

Exemples :

```
double rayon, surface;  
final double PI = 3.14;  
Cercle(double r) {  
    rayon = r;  
    surface = PI * rayon * rayon;  
    PI = 3.15; // illicite!!  
}  
  
class ExempleFinal {  
    public static void main(String[] args) {  
        final Cercle C;  
        C = new Cercle(2.5); // OK car 1'ere fois  
        C = new Cercle(25.7); // Faux car 2'eme fois  
        methode(1, 2);  
    }  
    static void methode(int a, final int b) {  
        a = 3; // OK car param'etre pas final  
        b = 4; // Faux car param'etre final  
    }  
}
```

Variables finales et objets référencés

Attention : `final` empêche l'affectation d'une nouvelle valeur à une variable, mais n'empêche pas de modifier l'éventuel objet référencé par cette variable :

```
class Container {
    int value;
}
class Proof {
    public static void main(String[] args) {
        Container c = new Container();
        c.value = 42;
        modify(c);
        System.out.println("La reponse est" + c.value);
    }
    /* on ne peut pas modifier la r'ef'ERENCE c elle-m^eme
       mais on peut modifier l'objet qu'elle r'ef'ERENCE */
    static void modify(final Container c) {
        c.value = -1; // va modifier l'objet r'ef'ERENC'EE par c
    }
}
```

Méthodes finales

Si l'on ajoute `final` à une méthode :

- ➡ Impossible de redéfinir (masquer) la méthode dans une sous-classe

Exemple : on aimerait que `viellir` de `Avatar` soit *toujours* appliquée

```
class Avatar
{
    ...
    final void vieillir() {
        --dureeVie;
    }
}
```

Une méthode finale s'utilise comme d'habitude :

```
Orc unOrc = new Orc(...);
orc.viellir();
```

Méthodes finales (2)

```
class Orc extends Avatar {  
    //...  
    void vieillir () { // faux si la methode  
                        // est finale dans Avatar  
                        dureeVie -=2;  
    }  
}
```

Utilité de finaliser une méthode :

- ▶ Aide à la programmation
- ▶ Rend le programme plus clair
- ▶ Simule la résolution *statique* des liens

Classes finales

Si l'on ajoute `final` à une classe :

➡ Impossible d'étendre la classe par une sous-classe

Exemple : on aimerait que la classe `Sorcier` n'ait jamais de sous-classe

```
final class Sorcier extends Avatar {  
    //...  
}
```

Une classe finale s'utilise comme d'habitude :

```
Sorcier oz = new Sorcier(...);
```

... mais **Attention** : une classe finale ne peut jamais avoir de sous-classe !

```
class MageNoir extends Sorcier { .. } // illicite!!
```

Classes finales (2)

Comme pour les variables et les méthodes, l'utilité de définir des classes finales est de :

- ▶ rendre le programme plus clair

Les méthodes et classes finales peuvent être à priori agaçantes :

- ▶ Exemple : la classe prédéfinie `String` est finale
- ▶ Aucune possibilité de définir

`class MyString extends String`

afin d'améliorer certaines méthodes par redéfinition !!

- ☞ Mais, permet de fixer une fois pour toute le comportement d'une classe

Ce que j'ai appris aujourd'hui

- ▶ Ce qu'est le **polymorphisme d'inclusion** en programmation
 - ▶ Par quel moyen ce concept puissant est mis oeuvre en Java :
 - ▶ Résolution dynamique des liens (late/dynamic binding)
 - ▶ Héritage des types
 - ▶ Classes et méthodes abstraites
 - ▶ Comment comparer des objets dans le contexte de l'héritage
 - ▶ Ce qu'est le modificateur `final`
- 👉 Je dispose maintenant d'un bon bagage pour exploiter les concepts OO de Java



Annexe : pour aller plus loin



- ▶ Sur la définition correcte de `equals` et la classe `Object` :
 - ▶ <http://download.oracle.com/javase/7/docs/api/java/lang/Object.html>
 - ▶ <http://www.artima.com/lejava/articles/equality.html>
- ▶ Sur l'usage de `instanceof` et `getClass` :
 - ▶ <http://www.artima.com/intv/bloch17.html>
- ▶ Sur la notion de covariance des types de retour (liée à l'overriding) :
http://en.wikipedia.org/wiki/Covariance_and_contravariance_%28computer_science%29#Java