IUT d’Orsay - DUT Informatique - Jean-Claude MARTIN – JEAN-CLAUDE.MARTIN@U-PSUD.FR

Programmation Orientée Objet (POO)  
COURS 4 : Héritage

Programmation structurée MONOLITHIQUE versus programmation orientée objet MODULAIRE avec plusieurs classes

Extrait de « Head First Java », Sierra & Bates

|  |  |
| --- | --- |
| SPEC. DE DEPART |  |
| PROGRAMMATION MONOLITHIQUE  **A EVITER**   * **PAS BIEN** |  |
| PROGRAMMATION **MODULAIRE avec des** **CLASSES**   * **BIEN** |  |
| OUPS … les spécifications changent ! |  |
| PROGRAMMATION MONOLITHIQUE  **A EVITER :**  **Modifier un code déjà testé, if…else en chaine / imbriqués**   * **PAS BIEN DU TOUT** | queasy = hasardeux |
| PROGRAMMATION MODULAIRE avec des **CLASSES : il suffit d’ajouter une nouvelle classe**   * **BIEN : INTERET DE LA POO** |  |

ET S’IL Y A DES ATTRIBUTS / METHODES IDENTIQUES DANS PLUSIEURS CLASSES ?

=> BESOIN DE POUVOIR CENTRALISER DES DECLARATIONS ET DU CODE POUR EVITER LES **REDONDANCES**

Intérêts de l’héritage

* éviter de dupliquer du code (attributs et méthodes) dans différentes classes qui partagent des caractéristiques communes
  + facilite les **modifications** futures du code commun :   
    …………………………………………………………………………………………………………………………………..
  + facilite **l’extension** à de nouveaux “cas” :   
    …………………………………………………………………………………………………………………………………..
* représenter explicitement une **logique** d’héritage des concepts du domaine (relation “est-un”)

Syntaxe

* Le mot-clé “extends”
* Exemple : une classe A avec un attribut a public
* Une classe B qui étend A et ajoute un attribut b public
* Déclarations et état de la mémoire après la creation de qq objets :

SCHEMA UML

CODE JAVA

SCHEMA DE LA MEMOIRE MODE RUBAN

SCHEMA DE LA MEMOIRE MODE BOITES ET FLECHES

Héritage des attributs

* Pour la gestion d’une bibliothèque on nous demande d’écrire une application traitant des documents de natures diverses : des livres, des revues, des dictionnaires, etc. Les livres, à leur tour, peuvent être des romans ou des manuels.
* Tous les documents ont un numéro d’enregistrement (un entier) et un titre (une chaîne de caractères).
* Les livres ont, en plus, un auteur (une chaîne) et un nombre de pages (un entier).
* Les romans ont éventuellement un prix littéraire (un entier conventionnel, parmi : GONCOURT, MEDICIS, INTERALLIE, etc.)
* Les manuels ont un niveau scolaire (un entier).
* Les revues ont un mois et une année (des entiers)
* Les dictionnaires ont une langue (une chaîne de caractères convenue, comme "anglais", "allemand", "espagnol", etc.).
* Tous les objets en question ici (livres, revues, dictionnaires, romans, etc.) doivent pouvoir être manipulés en tant que documents.
* Schéma des classes

UML sans le détail des attributs

* ATTENTION : Il faut vérifier la logique de votre schéma notamment qu’il est possible de dire par exemple   
  *un Roman est-un Livre*
* Déclaration en Java des classes
* ATTENTION : ne pas redéclarer dans une sous classe les attributs hérités dans la classe mère !
* Allocation des objets dans la mémoire : allocation automatique des attributs hérités

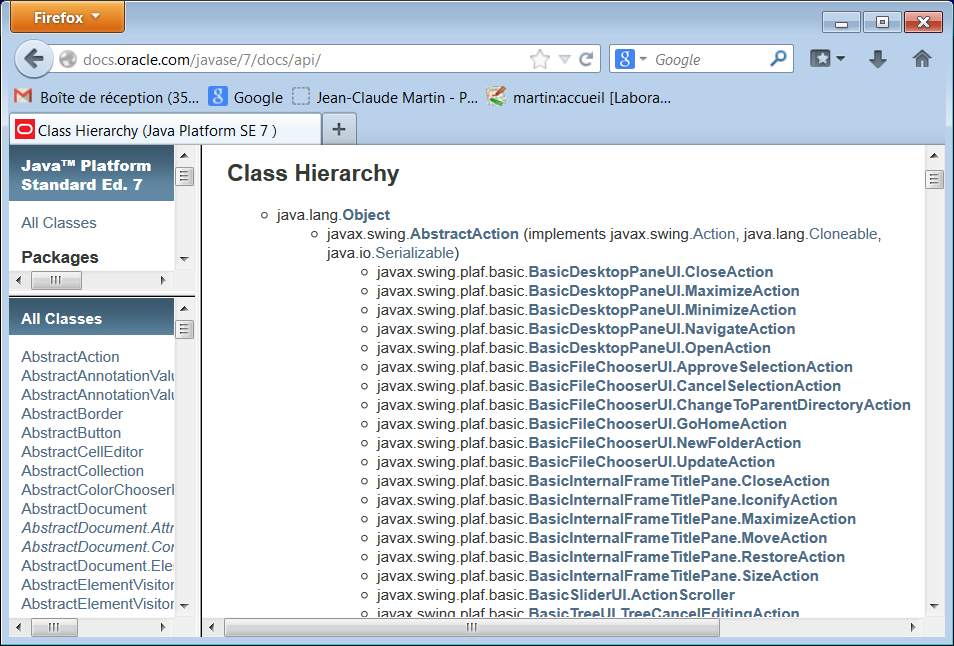
Vocabulaire

Class … extends …..

* La classe FILLE étend / hérite de la classe MERE
* classe-mère / super-classe : **. ……………………………………………………………..**
* sous-classe / classe dérivée : **……………………………………………………………..**

Hiérarchie des classes du JDK

* parcourir l’arborescence des classes dans la javadoc : “tree”
* la classe mère de toutes les classes en Java est : **………………………………………………………………..**



Modificateur d’accès et héritage : protected

* Un attribut ou une méthode protected d’une classe C est accessible par les méthodes
* 1) des sous-classes de la classe C même dans des paquetages différents de celui de C
* 2) des classes du même paquetage que celui de C

Package p1 ;

Class A {

private x

protected y

…

Package p2 ;

Class B extends A

* Est-ce que B peut accéder à l’attribut x : **………………………………………………………………..**
* Est-ce que B peut accéder à l’attribut y : **………………………………………………………………..**
* Est-ce qu’une classe C n’héritant pas de A peut accéder à x : **………………………………………………..**
* Est-ce qu’une classe C n’héritant pas de A et n’étant pas dans le même paquetage peut accéder à y : **………………………………………………………………..**
* Est-ce que ces accès peuvent être en lecture mais pas en modification : **………………………………………………………………..**

Méthodologie pour définir la hiérarchie des classes

* Déterminer les attributs et comportements communs à plusieurs classes
* Définir une classe représentant ces attributs et comportements communs
* Décider si une sous-classe a besoin de comportements spécifiques (redéfinition ou ajout de méthodes)
* ATTENTION : ne pas confondre “est-un” (HERITAGE) et “a-un” (ATTRIBUTS)

Héritage multiple

* Dans certains langages, une classe peut hériter de plusieurs classes
* En Java : est-ce que l’héritage multiple est possible : NON
* Cf cours à venir sur les interfaces

Héritage de **méthodes**

* But : réutilisation des fonctionnalités / comportements de la classe héritée

class Forme {

int x ;

int y ;

int couleur ;

void afficher () {

System.out.println ("x = " + x + " , y = " + y + ", couleur = " + couleur);

}

}

* + Cercle (sans méthodes)
    - attribut spécifique : ………………………………………………………….

Class Cercle ………………………………………..

}

class Dessin {

public static void main (String args[]) {

// Créer une forme et un cercle

…………………………………………………………………………………………………………..

…………………………………………………………………………………………………………..

// Afficher leurs attributs avec un println

…………………………………………………………………………………………………………..

…………………………………………………………………………………………………………..

// Appeler la méthode afficher sur une instance de la classe Forme

…………………………………………………………………………………………………………..

// Appeler la méthode afficher sur une instance de la classe Cercle :

// Cette méthode est HERITEE !

…………………………………………………………………………………………………………..

}

}Redéfinition de méthodes dans les sous-classes :   
LE **POLYMORPHISME** (par sous-typage)

* Un des buts de l’héritage : réutilisation dans une classe fille des fonctionnalités d’une classe mere tout en apportant certaines variations spécifiques de la classe fille : par exemple une **implementation différente de la même méthode**
* Exemple :
  + Redéfinition de la méthode afficher() dans la classe Cercle et affichage de l’attribut spécifique aux cercles

**class** Forme {

**int** x ;

**int** y ;

**int** couleur ;

**void** afficher () {

System.*out*.println ("x = " + x + " , y = " + y + ", couleur = " + couleur);

}

}

**class** Cercle

}

public class Dessin {

public static void main (String args[]) {

// Créer une forme et un cercle

Forme f = new Forme (); f.x = 10 ;

Cercle c = new Cercle (); c.rayon = 5 ;

// Afficher leurs attributs

System.*out*.println (f.x + "," + f.y + "," + f.couleur);

System.*out*.println (c.x + "," + c.y + "," + c.couleur + "," + c.rayon);

// Appeler la méthode afficher DE LA CLASSE FORME sur une instance de la classe Forme

f.afficher();

// Appeler la méthode afficher sur une instance de la classe Cercle :

c.afficher (); => appelle la méthode afficher de la classe Cercle

}

}

AFFICHAGE :

* POLYMORPHISME des méthodes :
  + une méthode (et ses paramètres) peut être définie dans différentes classes
  + ce sont différentes **formes** d’une même méthode (POLY-MORPHE)
  + la méthode se comporte différement selon l’objet sur lequel elle est appellée (car l’implémentation (caf les instructions) sont différentes)
  + le nom et les types et ordre des paramètres doivent être identiques
  + le type retourné peut être différent !
  + Surcharge vs. polymorphe
    - Une méthode **surchargée** diffère de la méthode originale par le nombre ou le type des paramètres qu'elle prend en entrée. => dans la même classe
    - Une méthode **polymorphe** a un squelette identique à la méthode de base, mais traite les choses différemment. Cette méthode se trouve dans une autre classe.

Résolution d’appel de méthodes  
QUELLE METHODE APPELER (QUAND IL Y A PLUSIEURS VERSIONS / FORMES DE LA MEME METHODE)

Forme f ;

f = new Cercle (); => AUTORISE : f PEUT PRENDRE PLUSIEURS FORMES

f.afficher (); // Qu’est-ce qui est affiché ? :;

* Possibilités
  + Résolution **statique** d’appel de méthode : C’EST LE COMPILATEUR QUI CHOISIT LA METHODE QUI DOIT ETRE APPELLEE : ici ce serait la méthode afficher de la classe Forme
  + Résolution **dynamique** d’appel de méthode : C’EST L’INTERPRETEUR QUI CHOISIT A L’EXECUTION LA METHODE A APPELER : ici ce serait la méthode afficher de la classe Cercle
  + **En java ce qui a été choisi par les concepteurs de Java :   
    Résolution ……………………………………………………...d’appel de méthode**
  + C’est la version la plus spécifique de la méthode qui est appelée   
    (la plus basse dans la hiérarchie en commençant à partir de la classe de l’objet)

Ajout d’une méthode agrandir() dans la classe Cercle

**class** Forme {

**int** x ;

**int** y ;

**int** couleur ;

**void** afficher () {

System.*out*.println ("x = " + x + " , y = " + y + ", couleur = " + couleur);

}

}

**class** Cercle **extends** Forme {

**int** rayon ;

**public** **void** afficher() {

System.*out*.println("Rayon : " + rayon);

}

**……………………**

}

public class Dessin {

public static void main (String args[]) {

// Créer une forme et un cercle

Forme f = new Forme (); f.x = 10 ;

Cercle c = new Cercle (); c.rayon = 5 ;

// Afficher leurs attributs

System.*out*.println (f.x + "," + f.y + "," + f.couleur);

System.*out*.println (c.x + "," + c.y + "," + c.couleur + "," + c.rayon);

c.agrandir(2); // 2

f.agrandir(2); //

System.*out*.println (f.x + "," + f.y + "," + f.couleur);

System.*out*.println (c.x + "," + c.y + "," + c.couleur + "," + c.rayon);

}

}

super

* Permet d’accéder explicitement (par exemple en cas d’ambiguité) à des attributs ou des méthodes de la classe mère
* Exemple : appel de la méthode afficher de Forme dans la méthode afficher de Cercle :

**class** Cercle **extends** Forme {

**int** rayon ;

**public** **void** afficher() {

**super**.afficher(); //

System.*out*.println("Rayon : " + rayon);

}

}

public class Dessin {

public static void main (String args[]) {

// Créer une forme et un cercle

Forme f = new Forme (); f.x = 10 ;

Cercle c = new Cercle (5);

// Appeler la méthode afficher sur une instance de la classe Forme

f.afficher();

// Appeler la méthode afficher sur une instance de la classe Cercle :

c.afficher (); // APPELLE LA METHODE afficher de Cercle (en commençant par appeler la méthode afficher de Forme)

}

}

AFFICHAGE :

x = 10 , y = 0, couleur = 0

x = 0 , y = 0, couleur = 0

Rayon : 5

Constructeur d’une classe héritée

* Exemple : class B extends A …
* Chaque sous-classe peut définir son constructeur
* Super : obligatoirement la 1ère instruction du constructeur de la classe dérivée
* Si le constructeur de B n’appelle pas le constructeur de A cela est fait automatiquement   
  (appel du constructeur sans paramètre de la classe A)

**class** Cercle **extends** Forme {

**int** rayon ;

Cercle (**int** r) {

// INSERTION AUTOMATIQUE DE L’APPEL DU CONSTRUCTEUR Forme ()

rayon = r ;

}

}

* Appel d’un constructeur de la classe mère avec paramètres :

class A {

A (int i, int j) {…}

…}

class B extends A {

B () {

super (10, 20) ; // ON APPELLE EXPLICITEMENT LE CONSTRUCTEUR DE LA CLASSE MERE

}

Méthodologie pour définir la hiérarchie des classes

* Déterminer les attributs et comportements communs à plusieurs classes
* Définir une classe représentant ces attributs et comportements communs
* Décider si une sous-classe a besoin de comportements spécifiques (redéfinition ou ajout de méthodes)
* ATTENTION : ne pas confondre “est-un” (HERITAGE) et “a-un” (ATTRIBUTS)

**VERSION INCORRECTE (car ce n’est pas logique de dire “un cercle a-une forme”)**

Class Cercle {

Forme f ; // Permet d’avoir des attributs x, y, couleurs … MAIS PAS D’HERITER DES METHODES DE FORME !

Int rayon ;

}

Opérateur instanceof

* Permet de déterminer la classe d’un objet

**public** **class** TestInstanceOf {

**public** **static** **void** main(String[] args) {

Forme f ;

f = **new** Forme ();

**if** (f **instanceof** Forme)

System.*out*.println ("f instance de Forme ") ;

**if** (f **instanceof** Cercle)

System.*out*.println ("f instance de Cercle ") ;

Forme c ;

c = **new** Cercle (2) ; // Permis grace au polymorphisme !

**if** (c **instanceof** Forme)

System.*out*.println ("c instance de Forme") ;

**if** (c **instanceof** Cercle)

System.*out*.println ("c instance de Cercle ") ;

}

}

AFFICHAGE

Tableaux d’objets

* Application du polymorphisme à chaque case du tableau
  + Forme formes [] ;
  + formes[i].afficher () // APPELLE LA METHODE AFFICHER DE LA SOUS-CLASSE : ex CERCLE, RECTANGLE, …
  + => AVANTAGE DU POLYMORPHISME : ON TRAITE DE MANIERE HOMOGENE DES OBJETS QUI ONT DIFFERENTES “FORMES” / QUI SONT INSTANCES DE DIFFERENTES SOUS CLASSES
* Si on veut appeler une méthode d’une sous-classe :
  + Tester avec instanceof
  + “caster” l’objet

**class** Cercle **extends** Forme {

**int** rayon ;

**int** diametre () {

**return** rayon \* 2 ;

}

}

...

if (formes[i] instanceof Cercle) {

// Appel d’une méthode spécifique de la classe Cercle

float d = ((Cercle) formes[i]).diametre() ;

}

Downcasting

* Force le compilateur à considérer une référence à une instance classe C  
  comme une référence à une instance d’une sous-classe de C
* Obligatoire lorsque l’on veut appeler une méthode spécifique à une sous-classe qui n’existe pas dans la classe Mere.

**class** Forme {

**int** x ;

**int** y ;

**int** couleur ;

**void** afficher () {

System.*out*.println ("x = " + x + " , y = " + y + ", couleur = " + couleur);

}

}

**class** Cercle **extends** Forme {

**int** rayon ;

**void** afficher () {

System.*out*.println ("x = " + x + " , y = " + y + ", " +

"couleur = " + couleur + " rayon : " + rayon);

}

**void** agrandir (**int** coef) {

rayon = rayon \* coef ;

}}

**public** **class** Editeur {

**public** **static** **void** main (String args[]) {

Cercle c ;

c = **new** Cercle (1);

c.agrandir(2); //

c.afficher();

Forme f ;

f = **new** Forme ();

//f.agrandir (2); //:

//

//Forme d ;

//d = new Cercle ();

//d.agrandir (2); //

Forme d ;

d = **new** Cercle (1);

**if** (d **instanceof** Cercle)

((Cercle) d).agrandir (2); //

}

}

Upcasting Synonymes : transtypage ascendant / généralisation

Un upcasting explicite n’a pas d’impact sur le fait que la résolution d’appel d’une méthode se fait dynamiquement : le polymorphisme se fonde sur une signature et un type de retour définis à la compilation (et qui ne seront donc pas remis en question lors de l’exécution).

**class** Animal

{

**public** **void** parler()

{

System.*out*.println("parler Animal");

}

}

**class** Chien **extends** Animal

{

**public** **void** parler ()

{

System.*out*.println("parler Chien");

}

}

**public** **class** Zoo

{

**public** **static** **void** main (String [] args)

{

Chien c = **new** Chien();

// Upcasting

Animal a = (Animal) c;

c.parler(); // APPEL DE LA METHODE DE LA CLASSE

a.parler ();// APPEL DE LA METHODE DE LA CLASSE

}

}

AFFICHAGE :

* Un upcasting est fait implicitement lors que l’on passe en paramètre à une méthode un objet instance d’une classe C2 alors que la méthode attends en paramètre un objet instance d’une classe plus générale
  + il n’y a pas besoin de faire un cast explicite lors de l’appel

class A {}

class B extends A {}

public class Test {

static void method(A a) {

System.out.println("Method A");

}

public static void main(String[] args) {

B b = new B();

method(b); // Upcasting implicite

method((A) b); // Upcasting explicite

}

}

* Un upcasting a un impact lors du choix entre plusieurs versions d’une méthode surchargée ; nécessite un cast explicite vers la classe mère

public class Animalerie

{

static void acheter (Animal a) {

System.*out*.println("Achat d'un Animal");

}

static void acheter (Chien c) {

System.*out*.println("Achat d'un Chien");

}

public static void main (String [] args) {

Chien c = new Chien();

*acheter* (c);

*acheter* ((Animal) c); // UPCASTING QUI A UN EFFET SUR LE CHOIX DE LA METHODE

}

}

AFFICHAGE :

Polymorphisme en Java =

* Compatibilité par affectation entre un type classe et un type ascendant
* Résolution dynamique d’appel de methode
* Valable avec un nombre quelconque de classes

*Résolution dynamique* d'appel de méthode

* Synonymes = *association tardive* entre l'appel d'une méthode et les instructions   
  / late binding / ligature dynamique

Pour aller plus loin

* <http://www.youtube.com/watch?v=9JpNY-XAseg>
* <http://fr.wikipedia.org/wiki/Polymorphisme_(informatique)>
* <http://www.siteduzero.com/tutoriel-3-10601-apprenez-a-programmer-en-java.html?all=1#ss_part_10353>
* <http://www.faqs.org/docs/think_java/TIJ309.htm>
* page 26 de <http://bigbozoid.free.fr/CoursMASTER1/IPA/COURS/Penser%20en%20Java%20(version%202.4).pdf>
* <http://forum.codecall.net/topic/50451-upcasting-downcasting/#axzz27gCwTOeZ>
* Downcasting & Run-time Type Identification (RTTI). <http://bruce-eckel.developpez.com/livres/java/traduction/tij2/?chap=8&page=2>