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

**Programmation Orientée Objet (POO)  
COURS 3 : Paquetages, modificateur d’accès, encapsulation**

# Normes de programmation OBLIGATOIRES

|  |  |
| --- | --- |
| * Un nom de classe commence par  **………une majuscule…………………**. * Un nom de méthode commence par  **………une minuscule…………………**. * Un nom d’attribut commence  **……………une minuscule…………**   ……**un underscore \_taille**   * Une constante s’écrit en  **en majuscules** * Une classe (PUBLIC) C1 se trouve dans un fichier qui s’appelle ………**C1.java**………………………………. * Indentation **OBLIGATOIRE** * **Un saut de ligne avant chaque bloc et on commence par un commentaire**   **// Parcourir tous les article**  **for (int i = …**  **{**  **….**  **}**  **// Afficher le total des …**   * Commentaires :  1 avant chaque méthode, 1 avant chaque bloc * numClient, numCompte  => utiliser le même nom d’attribut : "numero" dans toutes les classes : c’est de l’OO * Nombre de blocs imbriqués :  MAX 3 : au delà écrire des sous-programmes |  |

# Paquetages

Définition : un paquetage (“package” en anglais) est un regroupement de classes qui ont une fonction similaire ; par exemple classes permettant de gérer les entrées-sorties (java.io).

* Intérêt : …RANGER………………………………………………….

Utilisation d’un paquetage existant

Exemple : la classe File permet de récupérer les noms des fichiers du dossier courant

La classe File est dans le paquetage java.io

Pour l’utiliser : 2 solutions

Solution 1

class ExPaquetageExistant {

public static void main (String arg[]) {

System.out.println ("Affichage du dossier ");

java.io.File f = new java.io.File (".");

String dossier [] = f.list ();

// Boucle for each

for (String s : dossier) {

System.out.println ("-- " + s);

}

}

}

Solution 2

**import java.io.File ;** OU **import java.io.\*;**

class ExPaquetageExistant {

public static void main (String arg[]) {

System.out.println ("Affichage du dossier ");

**File** f = new File (".");

String dossier [] = f.list ();

…

Pour utiliser une classe qui est dans un package donné, il faut :

* Soit **…utiliser le nom long de la classe avec le nom du paquetage devant le nom de la classe**
* Soit **…importer la classe au début du fichier…………………………………………**

Le mot clé import PERMET D’UTILISER LE NOM COURT

Exception : La classe System est dans le paquetage java.lang

Pourquoi peut-on tout de meme l’utiliser ? LE PAQUETAGE **java.lang EST IMPORTE PAR DEFAUT**

Exemple :

Rien à mettre car Math est dans java.lang

class ExPaquetageExistant {

public static void main (String arg[]) {

System.out.println ("Utilisation d'un paquetage");

System.out.println ("2 puissance 3 = " + **Math**.pow (2,3));

}

Un paquetage peut contenir

**des classes**

**des sous paquetages**

Import des sous-paquetages : **l’import d’un paquetage n’importe pas les sous-paquetages**

Les paquetages du JDK : exemples

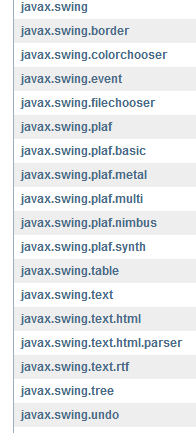
java.lang : classes de base (System, String, Math, …)

java.util : classes utilitaires (Random, Vector, Calendar, Arrays, ArrayList…)

java.io : entrées-sorties (fichiers, ...)

javax.swing : interfaces graphiques

javax.swing.**event** : événements (clics de souris, …)



“Companies use their reversed Internet domain name to begin their package names—for example, **com.example.mypackage** for a package named mypackage created by a programmer at example.com.”

Spécifier un paquetage pour une nouvelle classe

Mettre en début de fichier : exemple

**package** mobilier ;

* Classes Table, Chaise, …

Dans Eclipse :

Créer un paquetage

Dossier src : un dossier par paquetage => structure des dossiers = structure des paquetages

Si pas d’instruction package : default

Exemple d’utilisation de paquetage : le modèle MVC

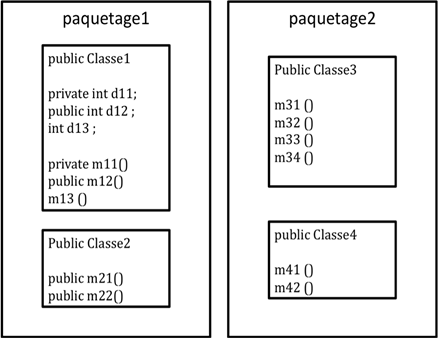
* Exemple de paquetages différents : le modèle MVC
  + modeles : les données de l’application (comptes, clients, …) et leurs traitement (recherche, tri, ajout d’un client, …)
  + vues : montrent un aspect de l'état du modèle (par exemple la liste des comptes d'un client donné)
  + contrôleurs : appellent des méthodes sur le modèle (par exemple ajout d'un compte) et demande à la vue correspondante de se mettre à jour

# Modificateurs d’accès

Visibilité / accessibilité des attributs et des méthodes d’une classe

* Pour un attribut, accessible = on peut consulter ou modifier sa valeur
* Pour une méthode, accessible = on peut appeler cette méthode
* **par défaut : accessible dans les méthodes des classes du même package**
* **public** : accessible dans les méthodes de toutes les classes
* **private** : accessible seulement dans les méthodes de cette classe (et les classes imbriquées)
* protected : accessible par cette classe et les sous-classes (cf cours sur l’héritage)

Exemple :



d11 est accessible par …m11, m12, m13…………………………………………

d12 est accessible par …toutes les méthodes de toutes les classes…………………………………………

d13 est accessible par …les méthodes de Classe1 et celles de Classe2 (même paquetage)…………

Visibilité / accessibilité d’une classe

* class C (par défaut)
  + cette classe est accessible depuis toutes les autres classes de ce paquetage
* public class C :
  + cette classe est accessible depuis toutes les autres classes

Classe / méthodes

Si une classe n’est pas déclarée public alors que ses méthodes sont déclarées public,  
 celles-ci ne seront pas accessibles d’un autre paquetage

Exemple de méthode privée

class Stock {

public gererStock () {

optimiser () ;

public maj () {

optimiser () ;

**private** void optimiser () {

…

# Encapsulation

VIDEO

<https://www.youtube.com/watch?v=szYzBC89CPE>

Principe de l’encapsulation

L’encapsulation consiste à permettre à un utilisateur d’utiliser un objet sans qu’il ne se soucie des détails internes. Un objet sert alors de conteneur / capsule pour regrouper et protéger l’accès aux attributs et aux méthodes.

Capsule = protection + entrées contrôlées dans la capsule

Schéma d’une capsule

Intérêt de l’encapsulation

L’encapsulation permet de cacher certains attributs des objets. Pour accéder à ces attributs, un programmeur est alors obligé de passer par des méthodes qui contrôlent les accès.

Accesseurs

Pour permettre tout de même d’accéder à des attributs que l’on a mis en private, il faut ajouter des méthodes qui permettent d’y accéder : ce sont des accesseurs :

- getX () => récupérer la valeur de l’attribut : float getSolde()

- setX(val) => donner une valeur à l’attribut : void setSolde ()

Exemple : classe Action, mot de passe pour accéder au solde, affichage des opérations, …

CLASSE ENCAPSULEE: Action

class Action {

private double montant = 100;

void setMontant (double val) {

if (val > 10 \* montant) {

System.out.println ("Tentative de magouille ?");

}

else

montant = val ;

}

double getMontant () {

return montant ;

}

}

CLASSE QUI EST MAINTENANT OBLIGEE DE PASSER PAR LES ACCESSEURS POUR ACCEDER AUX ATTRIBUTS DE Action :

class Magouille {

public static void main (String args[]) {

Action a = new Action ();

//AVANT ENCAPSULATION

//a.montant = -10 000;

System.out.println ("Montant initial : " + a.getMontant());

// Modifier le montant

a.setMontant(10000) ;

// On sauvegarde ces objets dans les fichiers

System.out.println ("Montant apres magouille : " + a.getMontant());

}

}

L’unité d’encapsulation est la classe

Exemple de la méthode coincide (Point pt) dans la classe Point

**public class Point {**

**private double x ;**

**private double y ;**

**Point (double x, double y) {**

**this.x = x ;**

**this.y = y ;**

**}**

**public boolean coincide (Point pt) {**

**// Cette méthode peut accéder aux attributs privés du paramètre pt !**

**// car pt est de type Point (donc de la classe de cette méthode)**

**// => L’unité d’encapsulation est LA CLASSE**

**// => Tous les objets instances d’une même classe sont DANS LA MEME CAPSULE**

**return ((pt.x == x) && (pt.y==y));**

}

public static void main (String args []) {

Point p1 = new Point (1, 3);

Point p2 = new Point (2, 9);

Point p3 = new Point (1, 3);

System.*out*.println ("p1 et p2 " + **p1.coincide(p2)**); // équivalent à p2.coincide (p1)

System.*out*.println ("p1 et p3 " + p1.coincide(p3));

}

public double getX() {

return x;

}

public void setX(double x) {

this.x = x;

}

public double getY() {

return y;

}

public void setY(double y) {

this.y = y;

}

}

Les méthodes d’une classe peuvent accéder

à tous les attributs private de l’objet sur lequel est appelé la méthode

ET AUSSI

aux attributs private de tous les objets de cette classe

(par exemple les objets passés en paramètre à la méthode)

Classes internes

Définition

Une classe *interne* est définie à l’intérieur d’une autre classe.

Exemple

**class Cercle**

{

private Centre c ;

private double r ;

**class Centre // La définition de cette classe est imbriquée dans celle de la classe Cercle.**

**{ public Centre (int x, int y){**

**this.x = x ; this.y = y ;**

**}**

**public void affiche()**

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

**}**

**private int x, y ;**

**}**

public Cercle (int x, int y, double r)

{ c = new Centre (x, y) ; **// Crée une instance de la classe imbriquée**……………..

this.r = r ;

}

public void affiche ()

{ System.out.print ("cercle de rayon " + r + " de centre ") ;

c.affiche() ; // ………………………………………………………………………………..

}

public void deplace (int dx, int dy)

{ c.x += dx ; c.y += dy ; // ………………………………………………………………………………..

}

}

public class TstCercl

{ public static void main (String args[])

{ Cercle c1 = new Cercle(1, 3, 2.5) ;

c1.affiche() ;

c1.deplace (4, -2) ;

c1.affiche() ;

}

}

**cercle de rayon 2.5 de centre 1, 3  
cercle de rayon 2.5 de centre 5, 1**

Remarques :

* Permet de réduire le nombre de classes visibles
* Doit rester logique

Droits d’accès devant la classe imbriquée

* public : la classe interne est accessible partout où sa classe externe l’est en précisant par exemple   
  **new Cercle.Centre ()**
* **private** (possible sur une classe **imbriquée**, pas sur une classe externe)
* par défaut : accessible depuis les classes du même paquetage