**Introduction à la programmation orientée objet**

Partie 0 : Présentation et organisation

**PROGRAMME**

* **POO : Programmation Orientée Objet**
  + Objet, classe
    - Programmation structurée par les données
    - Types de Données Abstrait (TDA)
    - Réalisation et utilisation
    - Gestion des exceptions
  + Structuration
    - Composition / agrégation
    - Interfaces / héritage / classes abstraites
      * Généricité
      * Typage dynamique et polymorphismes
* **Modélisation UML**
  + Diagramme de classe
* **Outils JAVA**
  + API, packages, jar, javadoc
  + Collections
    - TDA séquentiel ; listes
    - Utilisation des packages de l'API Java : ArrayList ?
* **Outils TP**
  + Linux, distribution Ubuntu : <http://www.ubuntu-fr.org>
  + Editeurs, Outils de développement
    - Emacs : "télécharger emacs"
    - Eclipse : <http://www.eclipse.org/>

**ORGANISATION**

* **Volume horaire**
  + CM 16h : 8 x 2h
  + TD 18h : 9 x 2h
  + TP 20h : 10 x 2h
* **Emploie du temps**
  + CM début semaine 3 : 2 séances puis 1 séance / semaine (lundi)
  + TD debut semaine 3 : 1 séance / semaine (mardi ou mercredi)
  + TP début semaine 4 : 1 séance / semaine (mercredi, jeudi ou vendredi)
* **Travail personnel à fournir toutes les semaines !!!**
  + Pendant les CM : distribution des feuilles de TD

**🡺** Indication des exercices à réaliser pour le TD

* + TP : énoncés complets

🡺 Finir le TP quand il n'est pas fini pendant la séance

* **Informations**
  + Cours INFO0201 sur le Moodle de l'URCA

BV : Mes cours > Cours en ligne > Allez sur Moodle > INFO0201

* + - Support de cours
    - Enoncés de TD
  + <http://cyril-rabat.fr> >Enseignement > INFO0201
    - Enoncés de TP et réalisation (TP itératifs)
  + Site de la licence INFO <https://www.licenceinfo.fr/>
    - Suivi du présentiel
    - Notes de contrôle continu
  + Par mail sur la messagerie universitaire de l'URCA
* Evaluation
  + DS1 1h30 semaine 6 à la place du CM 20 pts
  + DS2 1h30 semaine 10 à la place du CM 20 pts 70 % "papier"
  + DST 2h semaine 14 ? 30 pts
  + ITP 1h semaine 17 ? 20 pts

30% "ordi"

* + CRTP évaluation des TP déposés sur le site 10 pts

100 pts

* + Session 2 (ssi nécessaire)
    - EET 2h semaine 26 70 pts "papier"
    - Conservation de la partie pratique 30 pts "ordi"

Partie 1 : Une classe, des objets …

**PLAN**

* Préambule
* Analyse d'un exemple simple
* COO : Conception Orienté Objet
* La vie des objets
* Contrôle d'accès
* Les méthodes
* Diagramme de classes
* Compléments techniques
* Conception d'un exemple complet

**PREAMBULE**

* Info0101 : objectif = algorithmique
  + On "pense" traitement
    - Fonctions + procédures
  + Classe = moyen d'exécuter un code (tester un algo)
    - Une seule calsse
    - Main, qui appelle des fonctions/procédures
* Info0201 : objectif = données
  + On "pense" données
    - Les données caractérisant un élément
    - Lorsque les éléments sont structurés
  + Classe = description des données (de la structure d'une catégorie d'éléments)
    - On décrit des éléments, on les utilise => 2 classes
    - Il peut donc y avoir des traitements (bien sûr !) : des traitements de données

**ANALYSE D'UN EXEMPLE DE CLASSE**

* Exemple : une carte bancaire

Qu'est-ce qu'une carte bancaire ? qu'est-ce qui la caractérise ?

Pour commencer on va se contenter de :

* + Un propriétaire
  + Un numéro de carte
  + Un code secret
  + Un plafond de paiement
* Modèle

Chaque caractéristique sera enregistrée comme un **attribut** de la classe avec un nom, un type, un niveau d'accès :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Propriétaire | N° de carte | Code secret | Plafond de paiement |
| Nom attribut | Nom | Num | Code | Plafond |
| Type | ch. De caractères | Entier | Entier | Réel |
| Niveau d'accès | Privé | Privé | Privé | Privé |

**ANALYSE D'UN EXEMPLE DE CLASSE**

* Modélisation

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Propriétaire | N° de carte | Code secret | Plafond de paiement |
| Nom attribut | Nom | Num | Code | Plafond |
| Type | ch. De caractères | Entier | Entier | Réel |
| Niveau d'accès | Privé | Privé | Privé | Privé |

Diagramme de classe Code JAVA

Class CB {

//attributs

Private String nom;

Private int num;

Private int code;

Private double plafond;

}

|  |
| --- |
| **CB** |
| - nom : String  - num : int  - code : int  - plafond : double |

* De la classe à l'objet

Nous avons caractérisé la carte bancaire, il est maintenant temps d'en créer une, voici ce que l'on souhaite :

**Code JAVA**

Par exemple dans une classe de test avec juste la procédure principale

//déclaration de la variable

CB premiereCB;

//instanciation de la classe

permiereCB = new CB("JONQUET", 49787456, 1234, 300.0)

|  |
| --- |
| **Ma première CB**  Nom : "JONQUET"  Num : 4978 7456  Code : 1234  Plafond : 300 € |

**Pour cela il faut un constructeur dans notre classe qui va créer l'objet !!**

On complète donc le code de notre classe :

**Code JAVA :**

Class CB {

//attributs

Private String nom;

Private int num;

Private int code;

Private double plafond;

//constructeur par initialisation;

Public CB(String n, int nu, int c, double p){

Nom=n;

Num=nu;

Code=c;

Plafond=p;

}

}

**Accès direct aux attributs dans la classe**

**Permet d'affecter une valeur à chaque attribut de la classe pour l'objet que l'on crée**

Voyons la représentation mémoire :

nom

"JONQUET"

(String)

String

num

49787456

int

code

plafond

double

300.0

int

1234

(CB)

premiereCB

CB

L'**instanciation** permet de créer l'objet dans un espace réservé de la mémoire et renvoie sa **référence** qui sera stockée dans la **variable** premiereCB de **type** CB.

**On a créé notre première classe et notre premier objet mais dont l'utilité est, pour le moment, limité…**

**Ajoutons quelques fonctionnalités.**

* Comportement du plafond de la CB

Qu'est-ce que le plafond d'une carte bancaire ? Est-il consultable ? modifiable ? en dehors de la classe.

Notre choix :

* Consultable hors de la classe
* Modifiable hors de la classe

Attribut privé => il faut donc ajouter des méthodes pour y accéder et le modifier

**Code JAVA**

…

//accès

Public double getPlafond(){

Return plafond; //retourne al valeur de l'attribut plafond de l'objet courant

}

//modification

Public void setPlafond(double p){

Plafond = p; //modification de la valeur de l'attribut de l'objet courant ici sans vérification…

}

* Et le code de la CB ?

Est-il consultable ? modifiable ? en dehors de la classe ?

Notre choix :

* Non-consultable en dehors de la classe
* Non-modifiable en dehors de la classe
* Mais il faut pouvoir le vérifier !
* Ajout d'une méthode verifCode : **code JAVA**

…

//vérification du code

Public boolean verifCode (int c){

Return (code==c); //retourne le résultat de la comparaison (vrai ou faux)

}

**Et le diagramme de classes ?**

* Nouveau diagramme de classes

|  |
| --- |
| **CB** |
| **-** nom : String  - num : int  - code : int  - plafond : double |
| + getPlafond() : double  + setPlafond(double) : -  + verifCode(int) : boolean |

* On ne met pas les constructeurs

**On peut maintenant faire une classe de test !**

* Utilisation dans une autre classe : **code JAVA**

Class TestCB{ //juste ma procédure principale (main)

Public static vois main (String[] args){

//déclaration des varaibles

CB carte1, carte2;

//instanciations

Carte1 = new CB("JONQUET", 49787456, 1234, 300.0);

Carte2 = new CB("JAILLET", 48875540, 9998, 300.0);

//modification du plafond carte1

Carte1.setPlafond(1000);

//vérification du code de carte2

If(carte2.verifCode(1234)) System.out.println("code bon");

Else System.out.println("code faux");

}

}

**Notation pointée**

**Accès par la référence**

* Utilisation dans une autre classe **Compilation JAVA**
  + Classe CB **🡺 fichier CB.java**
  + Classe TestCB **🡺 fichier TestCB.java**
  + Compilation des deux fichiers :

$> javac CB.java

$> javac TestCB.java

**🡺** CB.class

🡺 TestCB.class

* + Exécution de la classe de test

**Attention les 2 fichiers doivent être dans le même dossier**

$> java TestCB

* Utilisation dans une autre classe **Représentation mémoire**

(CB)

"JONQUET"

String

49787456

1234

300.0 1000.0

(String)

int

int

double

plafond

code

num

nom

CB

Carte1

(CB)

"JAILLET"

String

48875540

9998

300.0

(String)

int

int

double

plafond

code

num

nom

CB

Carte2

**V**

**Plaçons bien les choses avant de poursuivre.**

**COO : CONCEPTION ORIENTEE OBJET**

1. Analyser els différents types d'éléments en présence
2. Définir leurs caractéristiques (les données les caractérisant)
3. Expliciter l'ensemble des traitements qu'ils peuvent réaliser / subir

🡺 **Programmation dirigée par les données**

* + **TDA** : type de données abstrait
    - Données membres (éléments caractéristiques) : attributs
    - Opération / traitement : méthodes
  + **UML** : Unified Modeling Language
    - Moyen de modélisation => diagramme UML
  + **Module** : entité regroupant l'ensemble des memebres
    - Attributs et méthodes => encapsulation
  + **Langage Orienté Objet**
    - Décrit les TDA
    - Organisation modulaire
* Terminologie
  + **Diagramme de classes (UML)** : représente les membres (attributs et méthodes) du modèle au sein du module
  + **Classe** : modèle décrivant les caractéristiques communes et les comportements communs d'un ensemble d'éléments (module décrit selon le langage choisi)
  + **Objet** : représentation d'une classe
    - Classe constitue un ***générateur*** d'objet / un ***modèle*** d'objet
    - Objet est une ***instance*** de cette classe
  + **Membres :**
    - **Attributs :** données membres
    - **Méthodes :** comportement des objets de la classe
  + **Instanciation**
    - Concrétisation d'une classe en un objet particulier
    - Code : utilisation de l'opérateur ***new*** avec un constructeur
  + **Caractérisation** 
    - Un objet est caractérisé par les ***valeurs*** de ses attributs
    - Son comportement est défini par les méthodes de sa classe
* ***Classe*** : concept, description
* ***Objet*** : représentant concret d'une classe
* Une classe constitue un ***générateur*** d'objets
* Un objet est une ***instance*** de cette classe

**LA VIE DES OBJETS**

* Référence
  + **Les variables**
  + Pour un type primitif : contient sa valeur

- Exemple : int a = 5;

* + Pour un tableau : contient une référence vers le tableau

- Exemple : int[]  tab  = new  int[5];

* + Pour un objet : contient une référence vers l'objet

- Exemple : CB carte1 = new CB("JONQUET", …);

**=> Attention au test d'égalité et à l'affichage des objets et tableaux**

* + S.o.p(a); => ok
  + S.o.p(tab); => ok mais [I@4b1210ee
  + S.o.p(carte1); => ok mais  CB@12a3a380
  + **Manipulation hors de la classe qui le définit** 
    - Accès à un **attribut** si public

⇒ Par référence, notation pointée

* + - Accès à une **méthode** si public

⇒ Idem, par référence, notation pointée

⇒ Doit être définie dans la classe décrivant l'objet

⇒ L'objet est un paramètre **implicite** de la méthode appelée la méthode est appelé sur l'objet donc connaît l'objet !!

* + - Exemple : appel de la méthode verifCode sur l'objet carte2 avec comme paramètre un int de valeur 1234

//vérification du code de carte2

if(carte2.verifCode(1234)) System.out.println("code bon");

else System.out.println("code faux");

* + **Manipulation dans de la classe qui le définit** 
    - Accès à un **attribut** (privé ou public)

⇒ Accès direct sans notation pointée

* + - Accès à une méthode (privé ou public)

⇒ Idem, accès direct sans notation pointée

* + - Exemple : dans la **méthode** verifCode de la classe CB

-  code désigne l'attribut code de l'objet courant

-  la référence de l'objet courant est implicite

// vérification du code

public boolean verifCode(int c){

return (code == c); // retourne le résultat de la comparaison (vrai ou faux)

}

* Construction d'un objet
  + **Constructeur** 
    - Fixe les valeurs de l'ensemble des attributs d'un objet
    - Toute classe devrait en posséder **au moins un**
    - Il peut en exister plusieurs versions pour une même classe, en principe :

**- Par initialisation**

=> On précise toutes les valeurs de tous les attributs en paramètres

=> Attention : on vérifie que les valeurs sont correctes

**- Par défaut**

=> On ne précise rien (aucun paramètre), les valeurs sont fixées par défaut, selon une convention

**- Par copie**

=> On passe un objet de même type en paramètre, le nouvel objet aura les mêmes valeurs que celui passé en paramètre

On peut en ajouter d'autres avec seulement une partie des attributs, selon les besoins.

* + **Constructeur** 
    - C'est une pseudo méthode
    - Sans type de retour
    - Son nom est celui de la classe
  + **Construire** 
    - Utilisation de l'opérateur **new** : opérateur d'instanciation
  + **Allocation dynamique** 
    - Déclenchement automatique du bon constructeur
      * Se fait en fonction des paramètres passés lors de l'appel : leurs nombres et leurs types
      * Mécanisme de lookup
    - Allocation à l'exécution
* Appel d'un constructeur
  + Exemple

//1. declaration de la variable

CB premiereCB;

//2. instanciation de la classe

premiereCB = new CB("JONQUET", 49787456, 1234, 300.0)

Analyse :

1. premiereCB variable de type CB, non initialisée
2. Réservation en mémoire et initialisation des attributs

Affectation à la variable premiereCB de la référence de cet objet

Représentation mémoire :

(CB)

"JONQUET"

String

49787456

1234

300.0

(String)

int

int

double

plafond

code

num

nom

CB

premiereCB

7

* Les différents constructeurs
  + **Exemple**

//constructeur par défaut

public CB(){

nom = "personne";

num = 0;

code = 0;

plafond = 0;

}

//constructeur par copie

public CB(CB c){

nom = c.nom;

num = c.num;

code = c.code;

plafond = c.plafond;

}

…

class CB {…

//constructeur par initialisation

public CB(String n, int nu, int c, double p){

nom = n;

if(nu<100000000)

num = nu;

else num = 0;

if(c<=9999)

code = c;

else code = 0;

if(p>=0)

plafond = p;

else plafond = 0;

}

* **Suppression d'un objet** 
  + **Uniquement lorsqu'il n'est plus référencé** 
    - C'est-à-dire lorsque plus aucune variable ne possède la référence pour l'atteindre
    - Exemple : mettre la variable à null

=> attention si une seule variable pointe sur lui

* + **Ramasse-miette (garbage-collector)** 
    - Thread de faible priorité : en tâche de fond, pas instantané
    - Libère l'espace mémoire des objets non-référencés
    - Possibilité de forcer l'appel : System.gc();

**CONTRÔLE D'ACCES**

* **Violation d'accès** 
  + Eviter tout accès (si non-autorisé)
    - Exemple : l'attribut code d'un objet CB
  + Eviter toute modification (si non-autorisé)
    - Exemple : le nom du propriétaire de la CB
  + Eviter les valeurs incohérentes
    - Exemple : mettre le plafond de la CB à -500…
    - Permet d'ajouter des vérifications, ici > 0
* Protection
  + Empêcher l'accès direct => **contrôle** **d'accès**

**Sécurité = encapsulation + contrôle d'accès**

* + **Comment** ?
    - Interdire l'accès direct aux attributs
    - Modification uniquement par des méthodes (si autorisé)

=> Permet de faire des vérifications (y compris dans un constructeur !)

* + **Application** 
    - Interdire l'accès / l'utilisation => privé
    - Autoriser l'accès / l'utilisation => public
    - Doit être défini pour tous les membres (attributs et méthodes)
  + **Modificateurs d'accès** 
    - **public** 
      * Accès inconditionnel
      * Depuis tout emplacement
    - **private** 
      * Accès limité
      * Uniquement dans la classe où le membre est déclaré
    - Il y en a d'autres, on les découvrira en temps voulu
  + **Vocabulaire** 
    - Partie visible = public = **Interface** (attributs publics et signature des méthodes publiques)
    - Partie cachée = private = **Implémentation** (attributs privés, méthodes privées et code des méthodes publiques)
  + **Règles de bonne conduite** 
    - Attribut : privé
    - Méthode : public
    - Routine : privé

⇒ Routine : méthode utilitaire, non accessible à l'utilisateur

* + **Schématisation dans les diagrammes de classes** 
    - Public : +
    - Privé : -

**METHODES**

* Généralités
  + **Fonctions-procédures / méthodes** 
    - Regroupement d'un ensemble d'instructions qui constituent un traitement générique
    - On peut leur passer des paramètres
    - Elles peuvent renvoyer une valeur ou non
  + **Appel d'une méthode** 
    - Message qu'on envoie à un objet ou une classe
    - Exemple : carte1.setPlafond(500);

x = Math.sqrt(2);

NB: une fonction/procédure "classique" (INFO0101) est une méthode qui s'applique à une classe (la classe courante)

* + **Fonctions-procédures OU méthodes** 
    - 2 cas => 2 lieux
      * Traitement ponctuel

=> dans la classe de test

* + - * Traitement plus courant

=> dans la classe définissant l'objet

* + - Appel :
      * Traitement réalisé par la classe

⇒ L'objet est passé en paramètre (une référence de l'objet)

* + - * Traitement réalisé par l'objet lui-même

⇒ L'objet est un paramètre implicite

* + **Fonctions-procédures OU méthodes : exemple** 
    - On modélise des villes avec plusieurs caractéristiques dont les coordonnées GPS du centre de la ville.
    - On a besoin pour un traitement de calculer la distance entre 2 villes.

=> Est-ce que nous en auront besoin qu'une seule fois ? Est-ce que d'autres aussi pourraient s'en servir ?

Ponctuel : **fonction** dans la classe de traitement

// dans la classe de test

public static double calculDistance (Ville V1, Ville V2){…}

//appel dans le main de la classe de test

Ville reims = new Ville(…); //idem paris

double x = calculDistance(reims, paris);

Plus courant : **méthode** dans la classe définissant l'objet

// dans la classe Ville

public double calculDistance (Ville V){…}

//appel dans le main de la classe de test

Ville reims = new Ville(…); //idem paris

double x = reims.calculDistance(paris);

* **Passage de paramètre** 
  + **Par valeur**
  + Un appel de méthode **ne peut pas** modifier la valeur d'une variable passée en paramètre

1. Paramètres formels de type primitif

public static void echanger (int a, int b){

int tmp = a;

a = b;

b = tmp;

} // => ne fait rien

=> aucune modification possible

Les traitements réalisés sur les variables locales et les paramètres formels n'agissent pas sur les variables du main car **les paramètres effectifs sont des valeurs.**

1. Paramètres formels de type tableau ou objet

* La valeur de la variable est la référence => non modifiable
* Par contre l'objet ou le tableau référencé peut-être modifié
* Valable aussi pour le paramètre implicite d'un appel de méthode (l'objet sur lequel est appelé la méthode)

carte1.setPlafond(500);

// modifie l'attribut plafond de l'objet référencé par carte1

carte1 la référence : non modifié c'est **la valeur du paramètre effectif**

* Les méthodes classiques

1. **Accesseurs**

* Méthodes de manipulation des attributs
* Attribut par attribut
* Permet d'éviter que l'utilisateur accède directement aux attributs

=> indispensable dès que les attributs sont rendus privés

=> sauf si on ne veut pas permettre de le consulter / modifier

* Consultation
  + **getter** (du verbe anglais *get* => obtenir / récupérer)
  + Consultation de la valeur d'un attribut (toujours sauf cas particulier)
* Modification
  + **setter** (du verbe anglais *set* => fixer)
  + Modification de la valeur d'un attribut (si on le permet)
* **getter**
  + Indispensable pour consulter un attribut si il est privé
  + Nom : get + nom de l'attribut avec une majuscule
  + Type de retour : celui de l'attribut
  + Sans paramètre
  + Public

Exemple : consulter le plafond d'une CB => attribut privé sans accès direct

-  Appel (dans la classe de test)

double x;

~~x = carte1.plafond;~~ // pas d'accès direct

x = carte1.getPlafond();

-  Définition de la méthode (dans la classe personne)

public double getPlafond(){

return plafond; // retourne la valeur de l'attribut plafond de l'objet courant

}

* **Setter**
  + Indispensable pour modifier si attribut privé et si modifications autorisées
  + Nom : set + nom de l'attribut avec une majuscule
  + Type de retour : aucun (void)
  + Paramètre : un seul du type de l'attribut
  + Permet de vérifier la nouvelle valeur avant de l'affecter

Exemple : modifier le plafond d'une CB => attribut privé sans accès direct

-  Appel (dans la classe de test)

~~carte1.plafond = 1000;~~ // pas d'accès direct

~~carte1.setPlafond(-300);~~ // plafond non modifié

carte1.setPlafond(1000);

-  Définition de la méthode (dans la classe personne)

public void setPlafond(double p){

if(p >= 0)

plafond = p; // fixer l'attribut plafond de l'objet courant si >= 0

}

1. La méthode toString()

* Permet d'obtenir la description d'un objet sous forme d'une chaîne de caractères
* Prototype

public String toString() {…}

* + Méthode de transformation de l'état d'un objet en chaîne de caractères de type String
* Utilisation

CB carte1, carte2;

… // instanciation de carte1 et carte2

System.out.println ( carte1.toString() );

System.out.println ( carte2 );

* Cette méthode est dite implicite : on peut donc utiliser directement la référence dans l'affichage et c'est cette méthode qui sera appelée.

=> si on ne la prévoit pas dans le code de la classe, elle est générée automatiquement par le compilateur et affiche la référence de l'objet

1. La méthode egalA()

* Test d'égalité : compare l'objet courant (paramètre implicite) avec un autre objet passé en paramètre
* Type de retour : boolean
* Prototype

public boolean egalA(Type\_de\_la\_classe ref) {…}

* Utilisation

CB carte1, carte2;

… // instanciation de carte1 et carte2

if( carte1.egalA(carte2) )

…

* Il existe aussi la méthode equals qui compare l'objet courant avec un objet de type quelconque : plus technique, nous verrons cela plus loin
* Les méthodes classiques

**Ajout de ces méthodes dans notre exemple**

class CB {

…

//methode toString : description de l'objet sous forme d'une chaine de caractere

public String toString(){

String s = "La carte bancaire n°"+num+" appartenant a "+nom+" a un plafond autorise de "+plafond+" euros.";

return s;

}

//methode egalA : compare l'objet courant a celui passe en parametre

//reflexion : qu'est-ce que 2 CB identiques ? Plusieurs possibilités

public boolean egalA( CB ref ){

return (num==ref.num && nom.equals(ref.nom) && code==ref.code && plafond==ref.plafond);

}

}

**DIAGRAMME DE CLASSES**

* Pour une classe

|  |
| --- |
| **NOM** |
| Les attributs |
| Les méthodes |

* + NOM de la classe

Tous les attributs de la classe

=> avec leurs modificateurs d'accès (+ / -)

* + Toutes les méthodes

⇒ avec leurs modificateurs d'accès (+ / -)

⇒ attention pas les constructeurs

|  |
| --- |
| **CB** |
| - nom : String  - num : int  - code : int  - plafond : double |
| + getNom() : String  + getNum() : int  + getPlafond() :double  + setPlafond(double) : -  + verifCode(int) : boolean  +egalA(CB) : boolean  +toString() : String |

* Notre classe exemple
  + Ajout des getters / setters manquants
  + Ajout des méthodes classiques

**COMPLEMENT TECHNIQUES**

* Auto-référence : this
  + **Analyse du code d'une méthode**

//issue de la classe CB

// vérification du code

public boolean verifCode(int c){

boolean ok = false;

if ( code == c)

ok = true;

return ok; // retourne le résultat de la comparaison (vrai ou faux)

3 types de variables :

* + c paramètre formel

-  La valeur est transmise depuis l'extérieur de la méthode

-  Dans le bloc, il joue le même rôle que les variables locales

* + ok variable locale
  + code attribut de l'objet courant

=> déclaration en dehors de la méthode

* **this pour expliciter la référence de l'objet dans le code d'une méthode**

public double getPlafond(){

return **this**.plafond; // retourne la valeur de l'attribut plafond de l'objet courant

}

Possible aussi pour les constructeurs

public CB(String n, int nu, int c, double p){

**this**.nom = n; **this**.num = nu;

**this**.code = c; **this**.plafond = p;

}

**Synthèse**

* Attention : obligatoirement dans la classe décrivant l'objet
* On définit les attributs (avec leur nom)
* Dans les constructeurs et méthodes, on les utilise

-  Soit directement => la référence de l'objet courant est implicite

-  Soit via l'auto-référence **this** => explicite et notation pointée

* **this pour résoudre un masquage**

public void setPlafond(double plafond){

if(plafond>=0)

this.plafond = plafond; //????

}

Résoudre le masquage de l'attribut par un paramètre ou une variable locale : l'expliciter

public void setPlafond(double **plafond**){

if(plafond>=0)

**this.plafond** = **plafond**;

}

* **this(…) appel d'un autre constructeur**

this(…) désigne un autre constructeur choisi en fonction des paramètres passés

=> En pratique on part du constructeur par initialisation que l'on fait parfaitement avec toutes les vérifications et on l'appel dans les autres constructeurs

//constructeur par défaut

public CB(){

nom = "personne";

num = 0;

code = 0;

plafond = 0;

//ou this("personne", 0, 0, 0);

}

//constructeur par copie

public CB(CB c){

nom = c.nom;

num = c.num;

code = c.code;

plafond = c.plafond;

//ou this(c.nom, c.num, c.code, c.plafond);

}

* Surchage
  + **Surchage = surdéfinition (overloading)** 
    - Plusieurs méthodes de même nom dans une même classe
    - Type de retour identique
    - Elles diffèrent par leurs paramètres : nombres et/ou types
  + Choix de la version à appeler
    - Selon le contexte = nombre et types des paramètres
    - ***static lookup*** : choix statique (à la compilation)
  + Exemples :
    - Les constructeurs (par initialisation / par défaut / par copie)
    - Autre exemple :

public boolean verifCode(int c){…} // un entier inférieur ou égal à 9999

public boolean verifCode(int a, int b, int c, int d){…} // 4 chiffres entre 0 et 9

* + Une version peut en appeler une autre
    - Exemple : les constructeurs
    - Autre exemple :

public boolean verifCode(int c){ // un entier inférieur ou égal à 9999

return (c <= 9999);

}

public boolean verifCode(int a, int b, int c, int d){ // 4 chiffres entre 0 et 9

boolean ok = false;

if( a >= 0 && a < 10 && b >= 0 && b < 10 && c >= 0 && c < 10 && d>=0 && d<10)

ok = true;

// ou

// ok = verifCode( 1000 \* a + 100 \* b + 10 \* c + d );

return ok;

}

**CONCEPTION EXEMPLE COMPLET**

* Les personnes

Les questions à se poser pour établir le diagramme de classes et le code de la classe

* + **Informations définissant une personne ?**
  + **Données caractérisant une Personne**
  + **Constructeur par initialisation : quelles vérifications ?**
  + **Constructeur par défaut : quelles valeurs par défauts ?**
  + **Constructeur par copie : est-il judicieux ?**
  + **Getters : pour tous les attributs ?**
  + **Setters : pour tous les attributs ?**
  + **toString : quelles données affichent-on et comment ?**
  + **egalA : comment compare-t-on 2 Personnes ?**
  + **Méthodes supplémentaires**
* Les personnes
  + **Informations définissant une personne ?**

Nom, prénom, sexe, date de naissance, âge, taille, couleur de peau, couleur de cheveux, langue parlée (plusieurs peut-être même), numéro de sécu, adresse, tatouages, …

=> Tout dépend de ce que l'on veut en faire

* + **Données caractérisant une Personne**

Selon ce qu'on souhaite faire, il faut choisir les données que nous considérons comme caractérisant notre classe Personne, ici :

- nom : String

- prenom : String

- age : int

- adresse : String

=> Les attributs de la classe Personne

* **Constructeur par initialisation : quelles vérifications ?**

Nom, prénom, adresse : limite de taille ? Non pas spécialement

Age : positif ! (>=0 et <150 ?)

* **Constructeur par défaut : quelles valeurs par défauts ?**

- nom : "X"

- prenom : "x"

- age : 0

- adresse : "Lune"

* **Constructeur par copie : est-il judicieux ?**

Peut-on faire la copie d'une personne ? => non

Et finalement le constructeur par défaut ? …

* **Getters : pour tous les attributs ?**

Oui

Et si on avait modélisé les tatouages ?

* **Setters : pour tous les attributs ?**

- nom : peut-on changer de nom ? Oui (mariage, …)

- prenom : peut-on changer de prénom ? Oui (intérêt légitime)

- age : peut-on changer d'âge ? Non mais on vieillit !

- adresse : peut-on changer d'adresse ? Oui (déménagement, …)

* **toString : quelles données affiche-t-on et comment ?**

Ex : Christophe Jaillet a ?? ans et vit à Reims.

* **egalA : comment compare-t-on deux Personne ?**

Mêmes nom, prénom, âge et adresse ? Mais même comme cela sont-elles vraiment identiques ?

Même âge est-il suffisant ?

=> Tout dépend du contexte : ici je choisis arbitrairement même âge et même adresse !

* **Méthodes supplémentaires**

Tout dépend de ce que l'on a besoin de faire comme traitement.

On a vu qu'on peut vieillir :

- année par année ? oui

- de plusieurs années d'un coup ? Mise à jour ponctuelle, oui mais > 0

* **On peut maintenant établir le diagramme de classes**

|  |
| --- |
| **Personne** |
| - nom : String  - prenom : String  - age : int  - adresse : String |
| + getNom() : String  + getPrenom() : String  + getAge() : int  + getAdresse() : String  + setNom(String) : -  + setPrenom(String) : -  + setAdresse(String) : -  + toString() : String  + egalA(Personne) : boolean  + vieillir() : -  + vieillir(int) : - |

* + **Diagramme de classes**

**Et le code JAVA ?**

* + **Code java de la classe**

class Personne {

//attributs

private String nom;

private String prenom;

private int age;

private String adresse;

//constructeur par initialisation avec verification

public Personne(String n, String p, int a, String ad){

nom = n;

prenom = p;

adresse = ad;

age = 0;

if(a>=0) age = a;

}

//constructeur par defaut

public Personne(){

this("X","x",0,"Lune");

}

//constructeur par copie : non

//getters

public String getNom(){return nom;}

public String getPrenom(){return prenom;}

public int getAge(){return age;}

public String getAdresse(){return adresse;}

//setters

public void setNom(String nom)

{this.nom = nom;}

public void setPrenom(String p)

{prenom = p;}

public void setAdresse(String ad)

{adresse = ad;}

//description

public String toString(){

return Prénom + " " + Nom + " a " + age + "ans et vit à " + adresse + ".";

}

//egalite

public boolean egalA(Personne ref){

return age == ref.age && adresse.equals(ref.adresse);

}

//vieillir d'un an public void vieillir()

{ age++; }

//vieillir : mise a jour ponctuelle

public void vieillir(int a){

if(a > 0 && age+a < 150)

{ age = age + a; }

}

}

* + **Et les traitements ? code java de la classe de test**

class TestPersonne {

public static void main (String[ ] args) {

// déclarations des variables

Personne p1 = new Personne(" JAILLET", "Christophe", 20, "Reims");

Personne p2 = new Personne("RABAT", "Cyril", 15, "Reims");

//déménagement

p2.setAdresse("Cernay");

//modification des ages

p1.vieillir(25); p2. vieillir();

//affichage

S.o.p(p1.toString());

S.o.p(p2);

//on remodifie qu'ils soient identiques

p2.vieillir( p1.getAge() - p2.getAge() );

p2.setAdresse( p1.getAdresse() );

//comparaison

if(p1.egalA(p2) == true)

S.o.p("C'est pas les mêmes !");

else

S.o.p(p1.getNom() + " et " + p2.getNom() + "sont identiques !");

}

}

* + **Représentation mémoire et affichage**

"JAILLET"

String

(String)

int

(String)

adresse

age

nom

"Christophe"

prenom

String

(String)

P1

~~20~~ **45**

(Personne)

"Reims"

String

Personne

"RABAT"

nom

String

(String)

prenom

"Cyril"

(String)

(String)

int

~~"Reims"~~

age

adresse

(Personne)

P1

~~15~~ **~~16~~ 45**

String

String

**"Reims"**

**~~"Cernay"~~**

**Affichage**

Christophe Jaillet a 45 ans et vit à Reims.

Cyril RABAT a 16 ans et vit à Cernay.

JAILLET et RABAT sont identiques !

Partie 2 : Membres de classe

Plan :

* + Membres de classe
  + Attributs et méthides
  + Digramme de classes
  + Représentation mémoire
  + Retour sur les exemples
  + Modificateurs (synthèse)

**ATTRIBUT DE CLASSE**

Objectif

Définir un attribut dont la valeur sera partagée par toutes les instances de cette classe

=> Attribut commun à tous les objets

Ex :

Pour une classe Cercle, la valeur PI

Un compteur d'instance de la classe

Le min et le max d'un ensemble d'objets d'une même classe

Attribut de classe

* + Un seul exemplaire en mémoire, avec une valeur dès sa déclaration
  + Indépendant de tout objet de la classe
  + Mot clé : static

Attribut d'instance

* + Autant d'exemplaires en mémoire que d'instance de la classe
  + Dépend de l'objet instancié
  + Mot clé : aucun

Un attribut de classe doit exister complètement avant qu'on crée des instances, donc on doit lui donner une valeur lors de sa déclaration.

Utilisation

* + Depuis une méthode de sa classe : comme un attribut d'instance
  + Depuis une autre classe (classe de test, …) :
    - Si privé => inaccessible
    - si public => notation pointée
      * À partir du nom de la classe

double pi = Cercle.PI;

Un attribut de classe peut être manipulé sans qu'un objet de cette classe ne soit créé

* + - * A partir d'une instance de la classe

Cercle c = new Cercle(…);

double d = c.PI;

Une instance dispose des "éléments collectifs" de sa classe

**METHODE DE CLASSE**

Objectif

Méthode pouvant être appelée indépendamment de toute instance de la classe, voire sans instance de la classe

* + Indépendant de tout objet de la classe
  + Mot clé : static
  + Peut accéder qu’à des attributs ou méthodes static

=> ne peut pas accéder à des membres d’instance

Ex : Pour une classe Cercle, une méthode getPI()

class Cercle {

private static double PI = 3,14;

…

public static double getPI() { return PI; }

…

}

utilisation comme les attributs de classe

* + Depuis une méthode de sa classe : comme une méthode d'instance
  + Depuis une autre classe (classe de test, …) :
    - Si privé => inaccessible
    - si public => notation pointée
      * À partir du nom de la classe

double x = Math.sqrt(5.5);

Une méthode de classe peut être appelée sans qu'un objet de cette classe ne soit créé

* + - * A partir d'une instance de la classe

Cercle c = new Cercle(…) ;

double d = c.getPI();

Une instance dispose des "éléments collectifs" de sa classe

**EXEMPLE**

Une classe Truc

* + Deux attributs d'instance : 1 privé et 1 public => a et b : un entier, un réel

Un attribut de classe : privé => x : un entier (valeur 4)

* + Méthodes (d'instance / de classe) : getter(s) + setter(s) + toString

class Truc {

// déclarations des variables

private int a;

public double b;

private static int x = 4;

//constructeur par init.

public Truc(int a, double b){

this.a = a;

this.b = b;

}

//getters

public int getA(){ return a; }

public static int getX(){ return x; }

//setters

public void setA(int a){ this.a = a; }

public static void setX(int x){ this.x = x; }

//affichage

public String toString(){

return "a : "+ a + "; b : "+ b +"; x : " + x;

}

}

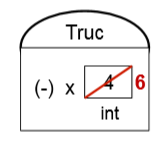
**DIAGRAMME DE CLASSE**

Les membres de classes (attributs et méthodes) sont soulignés dans le diagramme de classe.

Exemple :

|  |
| --- |
| **Truc** |
| - a : int  + b : double  - x : int |
| + getISBN() : int  +getPrix() : double  + getA() : int  + getX() : int  + setA(int) : -  + setX : -  toString() : String |
|  |

**Attributs / méthodes de classe**



**REPRESENTATION MEMOIRE**

\\dans une classe de test Truc

ref = new Truc(2, 3.5);

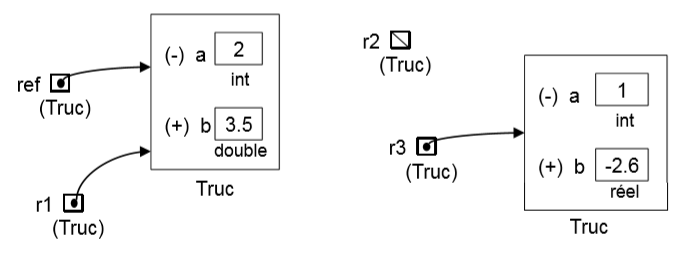
Truc r1 = ref;

r1.setX(6);

Truc r2;

Truc r3 = new Truc(1, -2.6);

**1 case mémoire pour toutes les instances de la classe**



**RETOUR EXEMPLE : LES PERSONNES**

On souhaite compter le nombre total de personnes générées.

⇒ Ajout d'un attribut de classe nbTotalPers

Peut-on y accéder ? Le modifier ?

⇒ Privé mais une méthode getNbTotalPers() publique

Initialisation ? Quand faut-il l'augmenter ?

⇒ Initialisation à zéro lors de sa déclaration et on l'augmente de 1 dans les constructeurs

class Personne {

//ajout attribut de classe

private static int nbTotalPers = 0;

…

//modification cons. par init.

public Personne(String n, String p, int a, String ad){

nbTotalPers++;

…

}

//cons. par defaut appelle le cons. par init. => pas de modif

//ajout getter (methode de classe)

public static int getNbTotalPers(){ return nbTotalPers; }

…

}

**RETOUR EXEMPLE : CARTE BANCAIRE**

On souhaite identifier nos cartes bancaires avec un numéro unique propre à notre classe. Comment faire ?

⇒ Ajout d'un attribut d'instance identifiant et d'un attribut de classe compteur

⇒ Init. compteur : 0 Init. identifiant : compteur

A chaque nouvelle carte, on augmente le compteur de 1 puis on l'affecte à l'identifiant donc modification des constructeurs et getter pour l'identifiant

class CB {

//ajout attributs

private static int compteur = 0;

private int identifiant;

…

//ajout getter (methode d'instance)

public int getIdentifiant(){ return identifiant; }

//modification cons. par init.

public CB(String n, int nu, int c, double p){

compteur++;

identifiant = compteur;

…

}

//autres cons. Appelle le cons. par init => pas de modif

…}

**Pas de paramètres**

**supplémentaire**

**MODIFICATEURS : SYNTHESE**

* + Portée
    - Public => accessible depuis n'importe quelle classe
    - Private => accessible uniquement depuis la classe où ce membre (attribut/méthode) est défini
  + Contexte
    - Static => de classe
    - non-static => d'instance
  + Modification
    - final => non modifiable
    - non-final => la valeur peut être modifiée par une instruction

NB : existe aussi pour les méthodes, et même les classes (cf prochains cours)

Partie 3 : interface et implémentation Javadoc

**PLAN**

* + Interface et implémentation
    - Mode d'emploi
    - Diagramme de classes
    - Réalisation en java
    - Contrat de comportement
    - Classification
    - Exemples
    - Typage dynamique
  + Javadoc

**INTERFACE : MODE D'EMPLOI D'UNE CLASSE**

* + Encapsulation

Les données et les méthodes sont rassemblées au sein de l'objet

* + Interface [partie publique]

Ensemble de méthodes accessibles depuis l'extérieur de la classe. Le monde extérieur n’a accès aux données que par l’intermédiaire de cet ensemble de méthodes.

* + Implémentation [partie privée]

C'est le détail [non visible] de la représentation des données et du codage des méthodes

**=> dans l’interface, les détails de l’implémentation sont cachés**

L'interface peut être vue comme le **mode d'emploi** d'une classe.

Répartition des rôles

* + **Interface**

=> Juste la signature des méthodes publiques (leur déclaration)

* + **Implémentation**

=> Les attributs privés

=> La définition de toutes les méthodes (leur code)

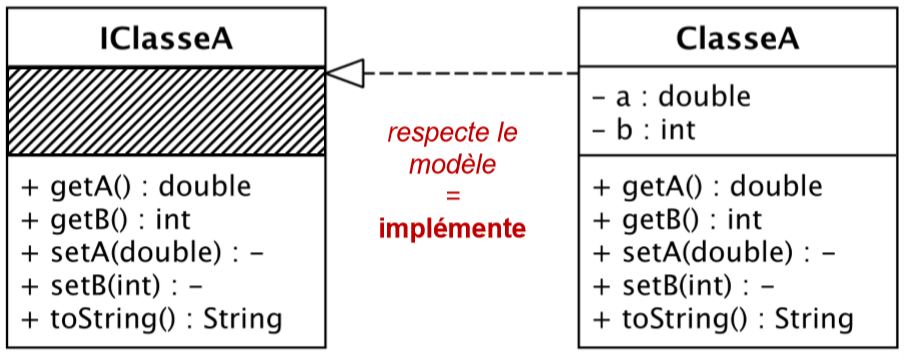
La classe **implémente** l'interface :

⇒ signifie qu'elle **respecte** le modèle

⇒ possède **au** **minimum** la définition des méthodes de l'interface

**DIAGRAMME DE CLASSE**

Exemple



**REALISATION EN JAVA**

Exemple

//fichier IClasseA.java

interface IClasseA {

public double getA() ;

public int getB() ;

public void setA(double a) ;

public void setB(int b) ;

public String toString() ;

}

//fichier ClasseA.java

class ClasseA **implements** IClasseA {

//attributs

private double a;

private int b;

//constructeurs

…

//methodes

public double getA(){return a;}

public int getB(){return b;}

public void setA(double a){ this.a = a; }

public void setB(int b){ this.b = b; }

…

}

**INTERFACE : CONTRAT DE COMPORTEMENT**

* + Interface
    - Déclinaison d'un ensemble de comportements (méthodes)
    - Pas d'attribut d'instance
    - Eventuellement des constantes publiques de classe
  + class + implements

Une classe utlise une interface en définissant les méthodes déclarées dans l'interface

⇒ Obligation de développement (respect du contrat)

**L'interface impose un contrat à respecter.**

**CLASSIFICATION**

Catégories de comportement

* + Une classe peut implémenter plusieurs interfaces
  + Chaque interface représente une catégorie de comportement à suivre

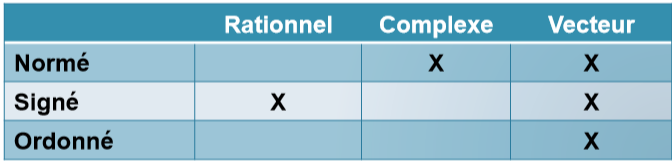
**=> Classification suivant des comportements**

Exemple : types numériques

3 classes : Rationnel, Complexe et Vecteur

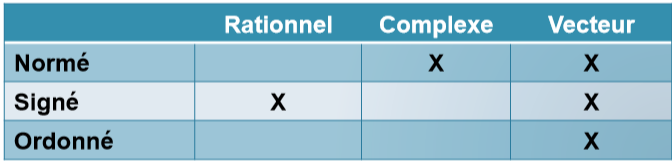
3 comportements : - Normé : norme()

- Signé : positif() et negatif()

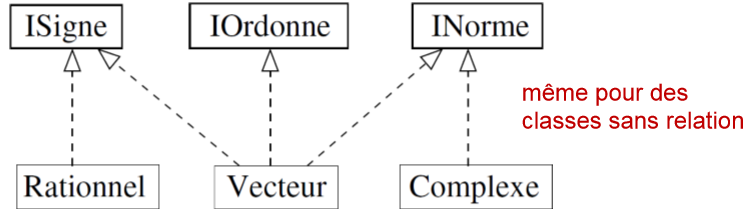
- Ordonné : croissant(), decroissant(), constant()

* + Catégories de comportement

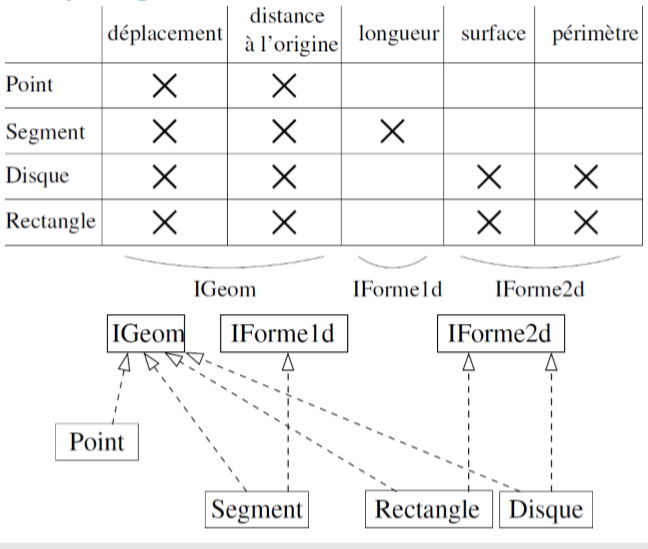
Exemple : types numériques



**=> classification suivant des comportements**



Autre exemple : géométrie



interface IGeom {

public void deplacement(double a, double b) ;

public double distanceALOrigine() ;

}

class Point implements IGeom {

… // attributs et constructeurs

public void deplacement(double a, double b){

x += a; y += b;

}

public double distanceALOrigine(){

return Math.sqrt( x \* x + y \* y );

}

… //autres methodes de la classe

}

class Segment implements IGeom, IForme1d {

… // plus d'obligations (celles des deux interfaces)

}

**TYPAGE DYNAMIQUE**

* + On peut déclarer des variables du type d'une interface
  + On peut affecter à une référence du type de l'interface un objet qui implémente cette interface

Exemples pour géométrie

class TestGeom {

public static void main (String[ ] args){

Point p = new Point(…);

Segment s = new Segment(…) IGeom ref;

ref = s;

ref.deplacer(3.5, -2.6);

~~double x = ref.longueur();~~ **impossible pas dans IGeom**

}

}

IGeom ref;

…

Ref = new Rectangle(…);

Ref.deplacer(3.5, -2.6);

~~double x = ref.surface();~~  **impossible pas dans IGeom**

…

Intérêt

Permettre la généricité

Par exemple : rassembler dans un tableau des objets de type différents

Un tableau ne peut stocker QUE des références du même types

=> Possible s'il existe un type de référence commun

IGeom[ ] tab = new IGeom[3];

tab[0] = new Rectangle(…);

tab[1] = new Disque(…);

tab[2] = new Rectangle(…);

for(int i = 0 ; i < 3 ; I++){

tab[i].deplacer(1.0, 2.0);

S.o.p~~.( tab[i].surface()~~ ); **impossible : existe dans Rectangle et disque Mais PAS dans IGeom**

}

…

**JAVADOC**

Outil javadoc

* + **javadoc** est le 3e programme essentiel fourni avec le SDK (javac, java)
  + Outil qui sert à générer la documentation au format Sun (API)
  + Permet à un tiers d'utiliser vos programmes et vos bibliothèques de manière simple et efficace
  + Permet aussi au programmeur de s'y retrouver plus rapidement

⇒ Maintien de votre documentation à jour

Si vous changez le nom d'une classe, d'une fonction ou même modifier la hiérarchie de votre programme, une simple nouvelle génération de la Javadoc appliquera automatiquement les changements

* + Outil fondamental pour programmer en Java
  + Génère des pages HTML contenant au minimum
    - La liste des classes
    - La liste des méthodes
    - La liste des variables
  + Possible d'ajouter des informations, des commentaires

⇒ Génère une véritable documentation exhaustive et facilement lisible

* + Avantage du format HTML : les liens hypertextes pour une navigation facilité
  + Une classe **correctement commentée** permet de générer une documentation précise et complète

Commentaires javadoc : conventions générales

Il existe une grande quantité de conseils pour la mise en page des commentaires Javadoc. Voici les principaux :

* + La première phrase Javadoc de la classe (ou de l’interface) doit être une description de la classe (ou de l’interface)
  + Ce conseil vaut aussi pour les méthodes, variables, …
  + Utilisez la troisième personne pour commenter une méthode
  + Détaillez le fonctionnement des méthodes si besoin
  + Utilisez les tags (cf. page suivante)
  + Les tags @param et @return doivent être systématiquement indiqués (sauf méthodes sans paramètres ou méthodes sans retour void)

Commentaires et *tags*

Il existe des commentaires spécialement conçus pour la Javadoc, avec des tags précis permettent de détailler des informations sur chaque classe, chaque méthode, … En voici une liste non-exhaustive :

* + @param sert à renseigner le ou les paramètres de la méthode. Derrière le tag, il faut renseigner le nom du paramètre (son type sera inclus automatiquement dans la Javadoc).
  + @return sert à décrire l’objet ou la valeur retourné par la méthode.
  + @throws et @exception indique la présence d’une exception qui sera propagée si elle se lève. Il faut bien indiquer le type de l’exception, et la raison de l’exception.
  + @author renseigne le nom de l’auteur de la classe. Si il y a plusieurs auteurs, il faut mettre plusieurs balises. Ce tag est un tag de classe ou d’interface uniquement.
  + @version indique le numéro de version de la classe. Ce tag ne peut être utilisé que pour une classe ou une interface, jamais pour une méthode.
  + @see permet de faire une référence à une autre méthode, classe, … Concrètement, cela se symbolisera par un lien hypertexte dans la Javadoc. C’est donc un des tags les plus importants.
  + @inheritDoc recopie la documentation de la classe mère. Utile pour éviter des copier/coller.

/\*\*

\* commentaire pour la documentation javadoc

\* avec utilisation des tags

\*/

Exemples avec les personnes

/∗∗

∗ La classe Personne modelise une personne.

∗ Elle sert pour …

∗ @see NegativeAgeException

∗ @author J. Jonquet

\* @version 1.0

∗/

class Personne {

/\*\*

\* le nom de la personne, une simple chaine de caractere

\*/

private String nom;

/\*\*

\* l'age de la personne, doit etre positif

\*/

private int age;

… // la meme chose pour les autres attributs

/\*\*

\* Constructeur par initialisation pour la classe Personne

\*

\* @param n le nom de la personne

\* @param p le prenom de la personne

\* @param a l'age de la personne >= 0

\* @param ad l'adresse de la personne

\* @throws NegativeAgeException si age < 0

\*/

public Personne(String n, String p, int a, String ad) throws NegativeAgeException {

nom = n;

prenom = p;

adresse = ad;

this.setAge(a); //gere l'exception, on verra cela dans la prochaine partie

}

…

/\*\*

\* Cree une representation sous forme de String de l'objet Personne

\*

\* @return une chaine de caracteres

\*/

public String toString(){

return prenom + " " + nom + " a " + age + "ans et vit à " + adresse + ".";

}

/\*\*

\* Compare une personne passee en parametre a l'objet courant

\* sachant que 2 personnes sont identiques si elles ont le meme age et la meme adresse

\*

\* @param ref la personne a compare

\* @return vrai si les personnes sont identiques, faux sinon

\*/

public boolean egalA(Personne ref){

return age == ref.age && adresse.equals(ref.adresse);

}

…

Utilisation du programme **javadoc**

En ligne de commande

* + option –sourcepath : racine des sources (quand utilisation des packages)
  + option –d : répertoire destination
  + arguments : noms de package et/ou noms de fichiers .java

Plusieurs niveaux de visibilité possibles

* + Option –public : seuls les membres public
  + Option –protected : seuls les membres public/protected
  + Option –package : seuls les membres public/protected/package
  + Option –private : tout !

javadoc –public \*.java –d docs //fichiers .java du repertoire courant

javadoc –private –sourcepath src –d docs cours.javadoc //package cours.javadoc

Partie 4 : énumérations, exceptions

**PLAN**

* + Enumérations
    - Principe
    - Utilisation
    - Exemples
  + Exceptions
    - Détecter et lever les erreurs
    - Lever et propager les erreurs
    - Exemples

**ENUMERATIONS**

Problématique

On souhaite pour une modélisation, enregistrer une couleur :

* + Quel type utiliser ?

Chaîne de caractère…

* + Comment s'assurer que la couleur est correcte ?

Avoir une liste de couleurs

Vérifier dans les constructeurs et setters

* + - * Peu pratique, assez lourd !

La solution depuis Java 5 : **Les énumérations**

* + Type de données particulier
  + Ne peut prendre qu'un nombre restreint de valeurs
  + Ces valeurs sont des constantes nommées

Définition d'une énumération

* + Mot-clé : enum
  + Dans un fichier indépendant .java
  + Contient uniquement les choix possibles

Exemple : Enumération Couleur et utilisation dans une classe TypeFruit

//fichier TypeFruit.java

class TypeFruit {

private String nom;

private Couleur couleur;

…

}

//fichier Couleur.java

**Enum** Couleur {

rouge,

bleu,

vert,

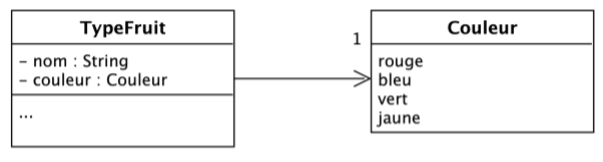
jaune;

}

Diagramme de classes

**Représentation en UML**

Exemple avec la classe TypeFruit et l’énumération Couleur



* + Utilisation d’une flèche de la classe vers l’énumération utilisée
  + Précision de la cardinalité
    - On détermine le nombre d’attributs du type de l’énumération
    - On ajoute ce nombre près de l’extrémité de la flèche

Utilisation

* + Dans le constructeur par initialisation de TypeFruit

public TypeFruit(String nom, Couleur couleur){

this.nom = nom;

this.couleur = couleur;

}

* + Dans la méthode toString de TypeFruit

public String toString(){

return nom + " " + couleur;

}

* + Dans la class e de test TestTypeFruit

TypeFruit type = new TypeFruit("pomme", Couleur.vert);

* + Pour récupérer la liste des valeurs d'une énumération
    - Méthode de la classe Enum : values
    - Retourne un tableau de Couleur

Couleur[ ] tabCouleurs = Couleur.values();

* + - Pour comparer une chaîne de caractère et une couleur :

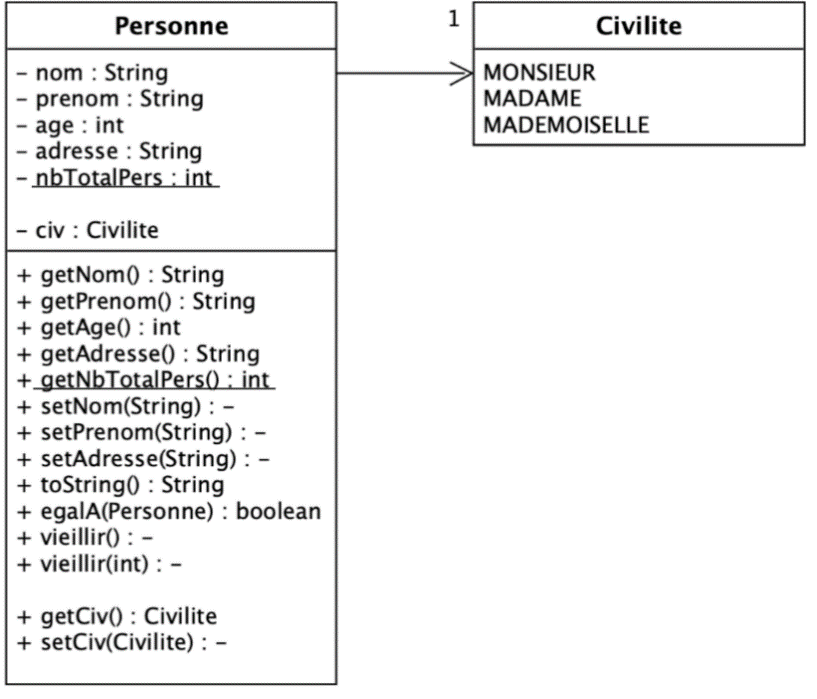
String coul = "rouge";

If(coul.equals(Couleur.rouge.toString())

System.out.println("Identique !!");

Retour exemple : les personnes

On souhaite ajouter la civilité d'une personne

* Ajout d’une énumération Civilite
* Dans la classe Personne : ajout d’un attribut d’instance civ de type Civilite
* Ajout des getters/setters pour cet attribut

Retour exemple : les personnes

Le code

**EXCEPTIONS**

|  |  |
| --- | --- |
| class Personne {  //ajout attribut d’instance  private Civilite civ;  …  //modification cons. par init.  public Personne(String n, String p, int a, String ad, Civilite c){  civ = c; …  }  //modification cons. par defaut  public Personne(){  this ("X","x",0,"Lune", Civilite.MONSIEUR);  } | //ajout getter civilite  public Civilite getCiv(){  return civ;  }  //ajout setter civilite  public void setCiv(Civilite c){  civ = c;  }  //modification toString  public String toString(){  return civ.toString() + prenom + " " + nom + " a " + age + "ans et vit a " + adresse + ".";  }  //modification egalA? non |

//fichier Civilite.java

enum Civilite{

MONSIEUR,

MADAME,

MADEMOISELLE ;

}

Problématique

La sécurité d'une application peut être rendue instable par toute une série de facteurs :

* + Des actions imprévues de l'utilisateur, entrainant par exemple une division par zéro…
  + Des débordements de stockage dans les structures de données…
  + Des problèmes liés au matériel : par exemple la perte subite d'une connexion à un périphérique…

Le logiciel doit donc se " **défendre**" contre de tels incidents potentiels

Programmation défensive

* + Prévoir les défaillances dues à certains paramètres interne ou externe
  + Protéger directement le code à l'aide de la notion **d'exception**

=> pour la construction rapide et efficace d'applications robustes

Rôle

* + Chargée de signaler un comportement exceptionnel (mais prévu!) d'une partie spécifique du code.
  + Souvent intégré aux langages de programmation actuels
  + En Java:
    - la classe Exception et ses nombreuses classes dérivées
    - Dès qu'une erreur se produit, un objet de la classe adéquate dérivée de la classe Exception est instancié, nous dirons que l'application **déclenche une exception.**

Exemple : programme sans gestion de l'exception

classTestClasse1 {

public static void main(String{ } Args){

Classe1 obj = new Classe1();

S.o.p.("Debut du programme");

obj.meth();

S.o.p.("Fin du programme");

}

}

class Classe1 { … //att + cons + …

public void meth(){

int x;

S.o.p.("Avant l'incident");

x = 1 / 0;

S.o.p.("Apres l'incident");

}

}

**Que se passe-t-il à l'exécution de la classe TestClasse1 ?**

Affichage :

Début du programme

Avant incident

Exception in thread "main" java.lang.ArithmeticException: / by zero

**Le programme s'arrête et la machine virtuelle Java (JVM) signale une erreur**

**=> Comment intercepter cette exception afin que notre programme ne s'arrête pas brutalement ?**

Principe

**Objets représentant les erreurs**

* + 3 mots clés pour détecter et traiter ces erreurs :
    - try
    - catch
    - finally
  + 2 mots clés pour les lever (déclencher) ou les propager
    - throw
    - throws

Principe : Détecter et traiter les erreurs

**try** (essayer)

Bloc rassemblant les appels de méthodes susceptibles de produire des exceptions, il "surveille".

**catch** (attraper)

Une exception levée dans un bloc surveillé peut être capturée et traitée.

* + - Traitement adapté de l'exception produite
    - Un catch par type d'exception à traiter

**finally**

Bloc qui sera toujours exécuté qu'une exception soit levée ou non

=> facultatif

**try** {

I1 ;

I2 ;

I3 ;

} **catch** (TypeExc1 e) {

T11;

} **catch** (TypeExc2 e) {

T21;

T22;

} **catch** (TypeExc3 e) {

. . .

} **finally** {

F;

}

**Fonctionnement**

1. Le gestionnaire d'exception (try) "déroute" l'exécution du programme vers le bloc d'interception catch
2. Celui-ci traite l'exception (exécute le code contenu dans le bloc catch)
3. Puis reprend l'exécution du programme (à l’instruction située après le gestionnaire lui-même)

**Levée exception**

**Après traitement**

**try** {

I1 ;

I2 ;

I3 ;

} **catch** (TypeExc1 e) {

T11;

} **catch** (TypeExc2 e) {

T21;

T22;

} **catch** (TypeExc3 e) {

. . .

} **finally** {

F;

}

…

Exemple : programme avec gestion de l'exception

**Traitement puis poursuite de l'exécution**

**Levée d'ArithmeticException**

classTestClasse1 {

public static void main(String{ } Args){

Classe1 obj = new Classe1();

S.o.p.("Debut du programme");

**try** {

obj.meth();

} **catch**(ArithmeticException e){

S.o.p.("Interception de l'exception");

}

S.o.p.("Fin du programme");

}

}

**Affichage obtenu**

Debut du programme

Avant incident

Interception exception

Fin du programme

**Engendre une exception**

class Classe1 { … //att + cons + …

public void meth(){

int x;

S.o.p.("Avant l'incident");

x = 1 / 0;

S.o.p.("Apres l'incident");

}…

Principe : Détecter et traiter les erreurs

**Traitement adapter à l'erreur**

Le gestionnaire d'exception (***try***) lève une exception il renvoie vers le bloc ***catch*** du **type** de l'exception levée : TypeExc1 ou TypeExc2 ou TypeExc3.

Recherche séquentielle (dans l'ordre des lignes de code) Þ Un bloc catch par type d'exception à traiter

* TypeExc1, TypeExc2, TypeExc3 sont des classes d'exceptions toutes distinctes (types différents)
* Un seul bloc catch est exécuté (celui du type de l'erreur)

**Levée exception**

try {

I1 ;

I2 ;

I3 ;

} catch (TypeExc1 e) {

T11;

} catch (TypeExc2 e) {

T21;

T22;

} catch (TypeExc3 e) {

. . .

} finally {

F;

}

….

Exemple : programme avec gestion de plusieurs exceptions

**Affichage obtenu**

Debut du programme

…

Interc ArrayStoreException

Fin du programme

**Traitement puis poursuite de l'exécution**

**Levée ArrayStoreException**

classTestClasse2 {

public static void main(String{ } Args){

Classe2 obj = new Classe2();

S.o.p.("Debut du programme");

try {

obj.meth();

} catch(ArithmeticException e){

S.o.p.("Intercepte ArithmeticException");

} catch(ArrayStroreException e){

S.o.p.("Intercepte ArrayStoreException");

}

S.o.p.("Fin du programme");

}

}

**Engendre une exception ArrayStoreException**

class Classe2 { … //att + cons + …

public void meth(){

…

}

…

**Attention à l'ordre des erreurs lorsqu'il y a une hiérarchie d'exceptions**

Exemple : soit une hiérarchie d'exceptions java

Supposons que nous souhaitions intercepter une troisième classe d'exception RuntimeException nous devons ajouter une clause **catch**

java.lang.Exception

|

+--java.lang.RuntimeException

|

+--java.lang.ArithmeticException

|

+--java.lang.ArrayStoreException

**Impossible erreur à la compilation**

**A mettre en dernier**

try {

//bloc de code a proteger generant un objet exception

}

catch(RuntimeException e){ /\*traitement RuntimeException\*/ } catch(ArithmeticException e){ /\*traitement ArithmeticException\*/ } catch(ArrayStroreException e){ /\*traitement ArrayStoreException\*/ } catch(RuntimeException e){ /\*traitement RuntimeException\*/ }

Placer en premier la clause **catch** chargée d'intercepter une RuntimeException implique que n'importe quelle exception héritant de RuntimeException (par ex. ArithmeticException) est considérée comme une RuntimeException.

**Il faut les ranger du plus précis au moins précis**

Principe : lever et propager les erreurs

**throw** Déclenchement manuel d'une exception

Exemple avec une exception **existante** :

**Affichage obtenu**

Debut du programme

Avant l'incident

Interception : Mauvais calcul

Fin du programme

classTestClasse3 {

public static void main(String{ } Args){

Classe3 obj = new Classe3();

S.o.p.("Debut du programme");

try {

obj.meth();

} catch(ArithmeticException e){

S.o.p.("Interception : "+ e.getMessage());

}

S.o.p.("Fin du programme");

}

…

class Classe3 { … //att + cons + …

public void meth(){

int x = 0;

S.o.p.("Avant l'incident");

if ( x == 0 ) **throw** **new** ArithmeticException("Mauvais calcul");

S.o.p.("Apres l'incident");

}

}

**On peut aussi créer des exceptions, cf. un prochain cours**

**throws** Pour les exceptions vérifiées

* + Exception **non vérifiée** (implicite) car très courante

=> hérite de l'une des deux classes Error ou RuntimeException

* + Exception **vérifiée**

=> toutes les autres

* + **Différence** : syntaxe à adopter dans une **méthode propageant** une exception
    - non vérifiée => comme vu avant
    - vérifiée => doit obligatoirement **signaler** au compilateur quelles sont les classes d'exceptions qu'elle laisse se propager sans traitement par un gestionnaire

***throws*** Pour les exceptions vérifiées

Reprise de l'exemple pour les exceptions non vérifiées mais avec la classe d'exception IOException qui est une **exception vérifiée**

**Affichage obtenu**

Debut du programme

Avant l'incident

Interception : Probleme E/S

Fin du programme

**Signaler l'exception susceptible d'être propagée**

class Classe4 { … //att + cons + …

public void meth() throws IOException{

int x = 0;

S.o.p.("Avant l'incident");

if ( x == 0 ) thrownew IOException("Probleme E/S");

S.o.p.("Apres l'incident");

}

}

Classe de test identique à l'exempleprécédent

Partie 5 : Agrégation / Composition

**PLAN**

* Agrégation / Composition
  + Exemple
  + Agrégation
  + Composition
  + Diagramme de classes
  + Représentation mémoire
  + Copie de surface / Copie profonde

**STRUCTURATION**

Les classes peuvent être reliées structurellement par des relations, en particulier :

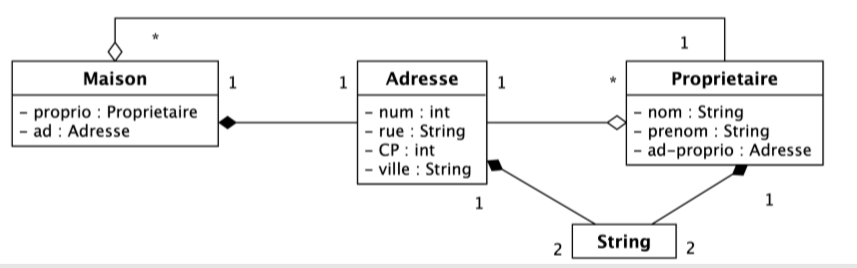
* + L’agrégation
  + La composition

Exemple

Considérons une maison, elle est situé à une adresse et a un propriétaire.

Que veut-on enregistrer pour cette adresse et ce propriétaire ?

Une simple chaine de caractère ?

Non => création d’une classe pour l’adresse et le propriétaire

**AGREGATION**

Définition

L’agrégation permet de définir qu’un objet est l’assemblage d’un ou de plusieurs sous-objets.

Des objets peuvent utiliser d’autres objets parmi leurs caractéristiques (ils ont comme attribut des références d’objets).

* relation : agrégat élément(s) agrégé(s)
* En POO :
  + L’objet contient les sous-objets
  + L’objet communique lui-même avec les sous-objets
* Règle à suivre :

Les sous-objets doivent avoir une relation structurelle ou fonctionnelle avec l'objet dont ils sont les constituants.

Ex : un clavier fait partie d’un ordinateur

**AGREGATION ET COMPOSITION**

Composition

La composition est un cas particulier de l’agrégation qui implique une dépendance plus forte de l’objet agrégé dans l’objet agrégeant.

Un objet peut être **agrégé**(partagé) par plusieurs objets, mais un objet ne peut être **composé** que par **un** objet unique (qui lui est dédié).

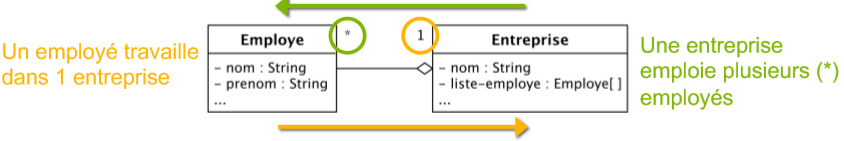
* Composition : Cycles de vie liés

Cycles de vie

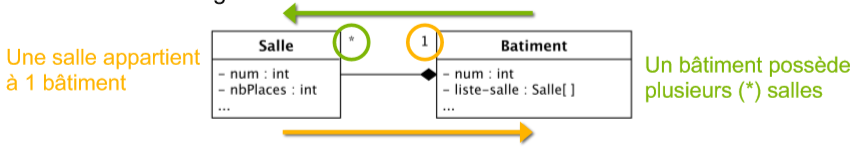
* **Agrégation** : une instance d'élément agrégé peut exister sans agrégat (et inversement), les cycles de vies de l'agrégat et de ses éléments agrégés peuvent être **indépendants**.
* **Composition** : les cycles de vies des éléments (les "composants") et de l'agrégat sont **liés**, si l'agrégat est détruit (ou copié), ses composants le sont aussi (les objets qui composent un objet A sont détruit en même temps que A)

Diagramme de classes

* Agrégation : relation non symétrique terminée sur le côté agrégeant avec un losange vide



* Composition : relation non symétrique terminée sur le côté agrégeant avec un losange noirci



* Cardinalités : précise le nombre d'instances qui participent à une relation

Le code : exemple Adresse Propriétaire Maison

* Agrégation : cycles de vie indépendants

// construction de l’adresse avant

Adresse **ad**= new Adresse(11, "rue du canal", 51100, "Reims");

Proprietaire p = new Proprietaire("Jonquet", "Jessica", **ad**);

// ajout de l’adresse après avec une reference

p = new Proprietaire("Jonquet", "Jessica", **null**);

p.setAdresse(ad);

// ou a la volee

p.setAdresse(new Adresse(11, "rue du canal", 51100, "Reims"));

/\* l’adresse peut exister sans le proprietaire \*/

* Composition : cycles de vie liés => composant créé avec le composite

Adresse **ad**= new Adresse(11, "rue du canal", 51100, "Reims");

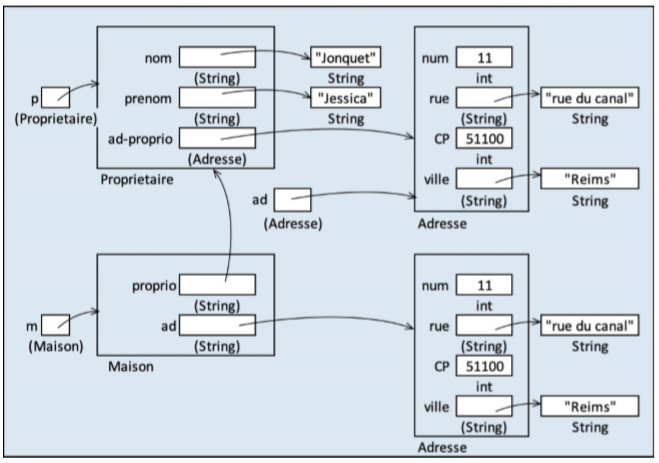
Proprietaire p= new Proprietaire("Jonquet", "Jessica", **ad**);

Maison m = new Maison(**p**, **~~ad~~**);

Maison m = new Maison(**p**, 11, "rue du canal", 51100, "Reims");

/\* l’adresse est cree par le constructeur de Maison qui appelle le constructeur de Adresse\*/

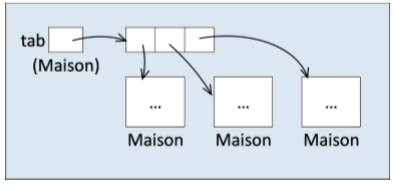
Représentation mémoire : Adresse Propriétaire Maison



Copie de surface / copie profonde

Principe :

* + Copie de surface : recopie de la référence (objet partagé)
  + Copie profonde : clonage de l’objet (avec le constructeur par copie de l’objet utilisé)



Exemple avec un tableau d’objets

* + Copie de surface

Maison[ ] tab2 = tab;

tab et tab2 contiennent la même référence vers le même tableau

* + Copie profonde

Maison[ ] tab3 = new Maison[tab.length];

For(int i=0 ; i<tab.length ; i++)

tab3[ i ] = ta[ i ];

tab et tab3 sont identiques mais totalement indépendants

Quel choix dans le constructeur ?

* + Composition : composant créé pour l’objet composite (n’existe que pour lui)

=> En cas de construction par copie, cloner le composant

* + Agrégation : l’agrégat existe indépendamment des éléments agrégés

=> Pas forcément nécessaire de s’en occuper au moment de la construction

=> Si on s’en occupe, il faut faire le bon choix

Quel choix dans le constructeur ?

Exemple constructeur Maison (Proprietaire – Adresse)

Maison – Proprietaire (agrégation) : le propriétaire peut être créé avant et on copie juste la référence ou on la crée à la volée.

* Le propriétaire peut avoir plusieurs maisons
* Changement de propriétaire possible
* On ne souhaite pas détruire le propriétaire quand on détruit la maison
* Si on lui passe une instance de Proprietaire **=>** **copie de surface**

Maison – Adresse (composition) : L’adresse n’appartient qu’à la maison, elle est créée par la maison qui est la seule à la détenir

* La maison de change pas d’adresse
* Pas de modification possible par d’autres classes
* On détruit l’adresse quand on détruit la maison
* Si on permet le passage d’une instance d’Adresse **=> copie profonde**

Partie 6 : Packages

**PLAN**

* Packages (paquetages)
  + Introduction
  + Packages de l’API Java
  + Packages utilisateurs
  + Notion de bibliothèques (library)

**INTRODUCTION**

Problématique : organisation

* Fichiers .java : un fichier par classe, interface, énumération
* Fichiers .class : un fichier par classe, interface, énumération

Une application : des dizaines/centaines de fichiers

=> besoin de structurer

Réalisation (Java) : les packages

* Constitution de groupements
  + Logiques (par thème)
  + Hiérarchisés
* Organisation arborescente (répertoires imbriqués)
* Permet de faciliter la recherche

Autre intérêt : la visibilité

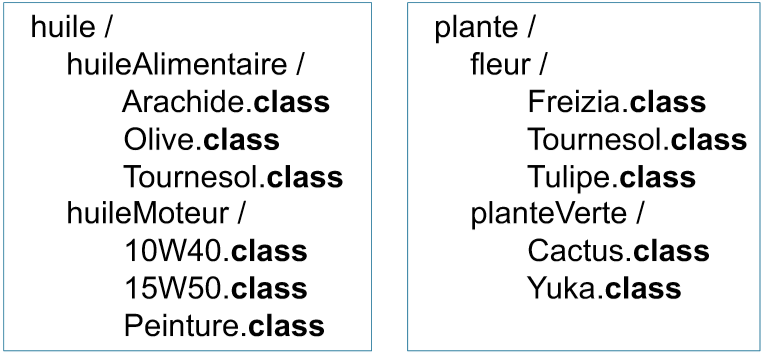
* Masquage d’une partie des classes (pas toutes utiles)

=> classes déclarées public ou non

* Nuancer le niveau de visibilité entre les classes
  + Pas de visibilité entre classes de packages différents
  + Les classes au sein d’un même package « se voient »
* Réutilisation des noms de classe

=> dans des packages différents, pour des concepts différents

Exemples



**LES PACKAGES DE L'API JAVA**

Structuration et exemples

* API : Application Programing Interface
* API Java 8 : Structuration logique en plus de 250 packages
  + Classes de base => java.lang
  + Entrées/sorties => java.io
  + Outils graphiques => java.awt
  + …
* Exemples
  + Package java.math :
    - 3 classes : BigDecimal, BigInteger, MathContext
    - 1 énumération : RoundingMode
  + Classe Math : dans le package java.lang
  + Package java.lang : 9 interfaces, 37 classes, 3 énumérations, 27 exceptions, 23 erreurs

API Java – Comment s’y retrouver

* Documentation en ligne (web ou copie locale)
  + https://docs.oracle.com/javase/8/docs/api/
  + Parcours facilité de l’arborescence
* Commande javap (p pour print)

...> $ javap java.lang.Math

public final class java.lang.Math extends java.lang.Object {

public staticfinal double E;

public staticfinal double PI ;

public static double sin (double) ;

public staticdouble cos (double) ;

public staticdouble tan (double) ;

public staticdouble asin (double) ;

...

}

Utilisation

=> Exemple avec la classe BigInteger, du package java.math

* Nommage explicite de la classe

java.math.BigInteger n;

n = java.math.BigInteger.valueOf(5); //methode de la classe

* Import de la classe, pour utiliser son nom court
  + Instruction d’import dans l’en-tête de la classe

import java.math.BigInteger;

* + Après import, inutile de préciser le nom long de la classe

BigInteger n;

n = BigInteger.valueOf(5); //methode de la classe

* + Possibilité d’importer toutes les classes d’un package en une fois

import java.math.\*;

**NB : import implicite de java.lang et ses sous-packages**

**LES PACKAGES** UTILISATEUR

Nommage

* Un package correspond à un répertoire
* Utiliser des noms les plus explicites possibles
* Convention : commencer le nom du package par une minuscule
* Nommage des répertoires identique à celui des packages

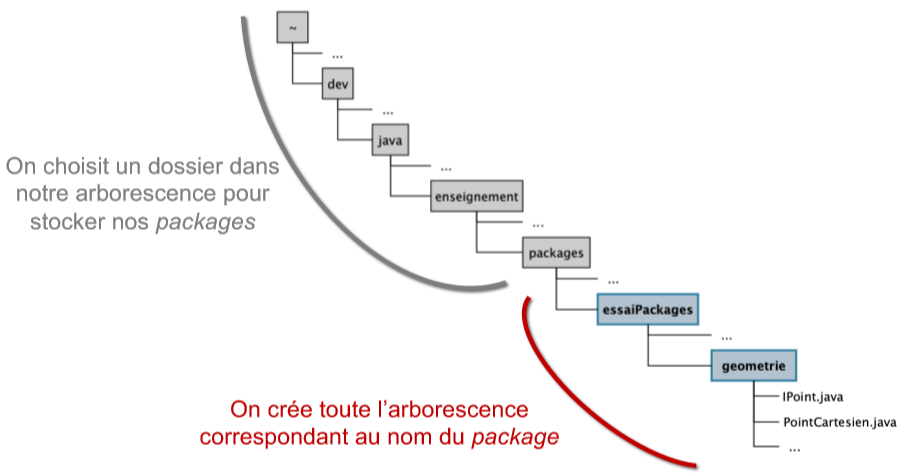
Organisation

* Placer tous les packages dans le même dossier (ou quelques dossiers stratégiques)
* Organisation en répertoires imbriqués
* Le nommage rend compte de l’imbrication

Exemple : essaiPackages.geometrie

Organisation et Nommage

Exemple essaiPackages.geometrie



Organisation et Nommage

Recommandation pour assurer l’unicité absolue des noms

* ajouter en préfixe une base d’identification universelle
* Exemple courant : utiliser le nom de domaine, à l’envers (du plus général au plus particulier)

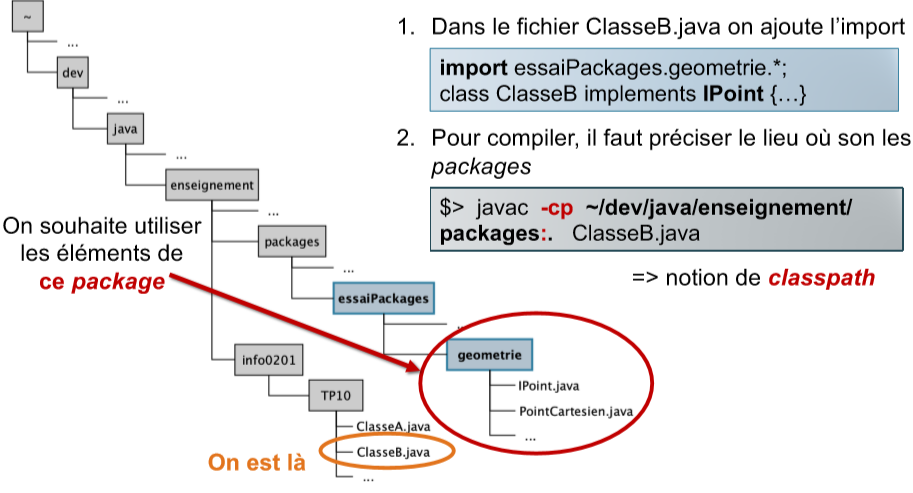
Exemple

fr.jonquet.enseignement.info0201.essaiPackages.geometrie

* + préfixe (nommage absolu)
  + repérage personel
  + nom naturel du paquetage

Par rapport à l’arborescence précédente il faut donc créer les dossiers imbriqués suivants : fr, jonquet, enseignement, info0201 et placer dans le dernier notre dossier essaiPackages

Utilisation d’un package personnel



Gestion du classpath

* Par défaut c’est le répertoire courant
* Sinon on doit le préciser
  + A la volée avec l’option cp

$> javac -cp ~/dev/java/enseignement/packages:. ClasseB.java

**A chaque compilation et exécution !!**

* + Avec la variable d’environnement CLASSPATH

$> export CLASSPATH=$CLASSPATH:~/dev/java/enseignement/packages

**Limité au temps d’exécution du terminal**

* + De façon pérenne en ajoutant l’export du CLASSPATH au fichier .bashrc (à la racine du répertoire de connexion)

Et sous Windows ?

* c’est le ; qui est utilisé à la place de :

set CLASSPATH=%CLASSPATH%;Z:\dev\Java\enseignement\packages

* commande dans la console
* variable d’environnement (voir ’’propriétés du poste de travail’’)

Extension de la notion de visibilité entre classes

* Les classes/interfaces/énumérations d’un même package "se voient"
* Pour utiliser une classe/interface/énumération en dehors du package où elle se trouve, il faut la déclarer publique

**Les classes, interfaces et énumérations d’un packagesont déclarées public**

**…sauf cas très particuliers de classes utilitaires**

* Notion d’interface d’un package : sa partie publique

NB : Utile pour que les classes soient documentées dans la javadoc (rappel TP2)

Ajouter des éléments à un package

* **Indiquer dans la classe / interface / énumération le package auquel elle appartient**

Dans le code de la classe / interface / énumération, ajouter à la première ligne

// fichier MaClasse.java

package monPackage;

public class MaClasse {…}

* **Faire adhérer la classe/interface au package**

Lors de la compilation, se placer à l’extérieur de répertoire correspondant au package et compiler à travers ce répertoire

// si besoin on remonte dans l’arborescence pour être en dehors du repertoire

$> cd ..

// compilation de la classe

$> javac monPackage/MaClasse.java

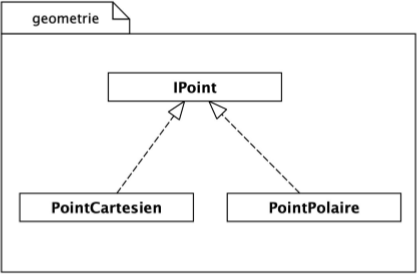
// ou compilation du package complet

$> javac \*.java

Diagramme UML

Il faut aussi indiquer les packages dans le diagramme de classes par un rectangle encadrant les classes/interfaces/énumérations appartenant au package et indiquer son nom dans une étiquette accrochée au package

Exemple :



Notion de bibliothèque (library)

Java propose l'utilitaire **jar** dans le JDK, un utilitaire permettant de rassembler les différentes classes (fichiers .class) d'une application au sein d'une archive compressée.

* **Objectifs / Intérêts** 
  + Tous les éléments d’une application réunis
  + Application compacte et moins volumineuse (en une fois, petite taille)
  + Exécutables directement (application intacte)
  + Archive transportable à travers le réseau et possibilité d’exécution à la volée
* **Création de l’archive** : commande **jar**

$> jar –cvf maBib.jar \*.class // c = create

* **Utilisation de l’archive** (exécution d’une archive et non d’une classe)

$> java –jar maBib.jar

* **Ouverture de l’archive** (pour ajouter des fonctionnalités par exemple)

$> jar –xvf maBib.jar // x = extract

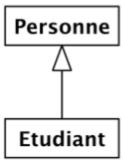
Partie 7 : Héritage

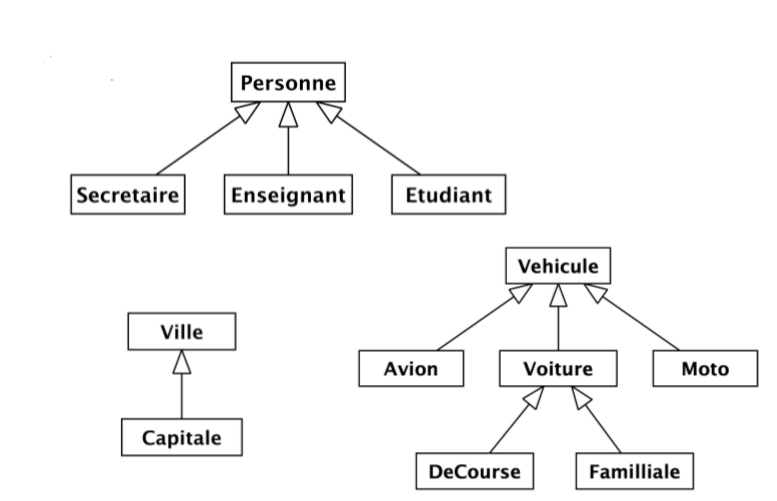
**PLAN**

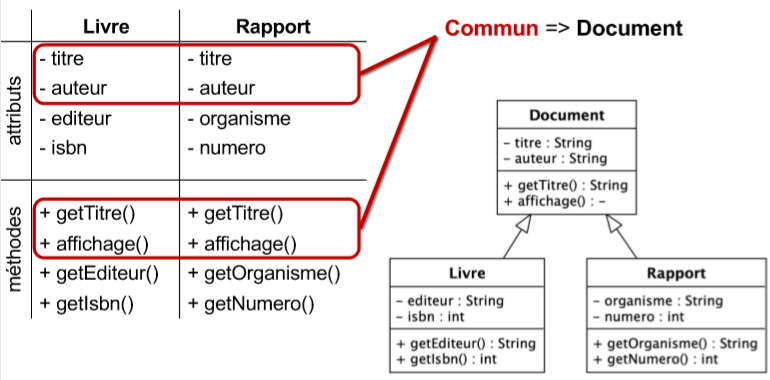
* Héritage
  + Introduction, exemples
  + Principe
  + Classe dérivée
  + Mise en œuvre (Java)
  + Modificateurs de portée
  + Constructeurs
  + Redéfinition de méthode
  + Classes et méthodes finales
  + Typage dynamique et polymorphisme
  + Object

**INTRODUCTION**

Le concept

* Construction de classes dérivées
* Certaines classes sont basées sur d’autres classes
* Relation : EST UN
* Exemple : un étudiant est une personne
  + La classe Etudiant hérite de la classe Personne
  + Le code de la classe Etudiant utilise celui de la classe Personne
  + Pas de besoin de réécrire la base (Personne)
* Relation de généralisation / spécialisation
* L’héritage permet le partage et la réutilisation de propriétés (attributs/méthodes)
* Factoriser de l’information ; factorisation du code

Exemples



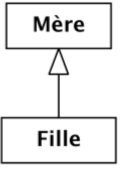
**PRINCIPE**

Factorisation et spécialisation

* Une classe de base, des classes dérivées
* Vision descendante : possibilité de reprendre intégralement tout ce qui a déjà été fait et de pouvoir l'enrichir
* Vision ascendante : possibilité de regrouper en un seul endroit ce qui est commun à plusieurs

**=> Evite la réécriture : factorisation + spécialisation**

Classe de base (mère, super-classe)

Plus générique => Partie commune

Classe(s) dérivée(s) (fille(s), sous-classe(s))

Plus spécifique(s) => Hérite(nt) de la partie commune

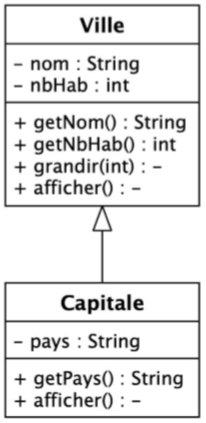
+ traitements spécifiques

+ attributs propres

Classe dérivée

* Cas particulier de la classe de base, enrichie d'informations supplémentaires
* Propriétés :
  + Contient les attributs de la classe de base
    - Peut en posséder de nouveaux
  + Contient les méthodes de la classe de base
    - Peut redéfinir certaines méthodes
    - Peut en avoir de nouvelles

**La classe dérivée hérite des membres de la classe de base**

Mot clé : **extends**

**MISE EN ŒUVRE (JAVA)**

Exemple : Une capitale est une ville

* Ville

Reims, ville de 183 113 hab.

- Nom

- Nombre d’habitants

* Capitale Paris, capitale de France (2 190 327 hab.)

- Nom - Nombre d’habitants

- Pays

=> Factorisation + spécialisation



class Ville {

private String nom;

private int nbHab;

public Ville (String nom, int nb) {

this.nom = nom;

nbHab = nb;

}

public String getNom(){return nom;}

public int getNbHab(){return nbHab;}

public void grandir(int nb){nbHab += nb;}

public void afficher(){

S.o.p. (nom + " , ville de “ + nbHab + " hab . ");

}

} //fin de la classe Ville

partie 7 diapo 10

class Capitale extends Ville {

//ajout d’un attribut

private String pays;

public Capital (String nom, String pays, int nb) {

super(nom, nb);

this.pays = pays;

}

//ajout d’une methode

public String getPays(){return pays;}

//redefinition de la methode afficher

public void afficher(){

S.o.p. (getNom() + " , capitale de " + pays + " ( " + getNbHab() + " hab.) ");

}

} //fin de la classe Capitale