MODULE 2

Conception orientée objet

HUMAN booster
VOTRE SOLUTION COMPETENCE

La conception orientée objet

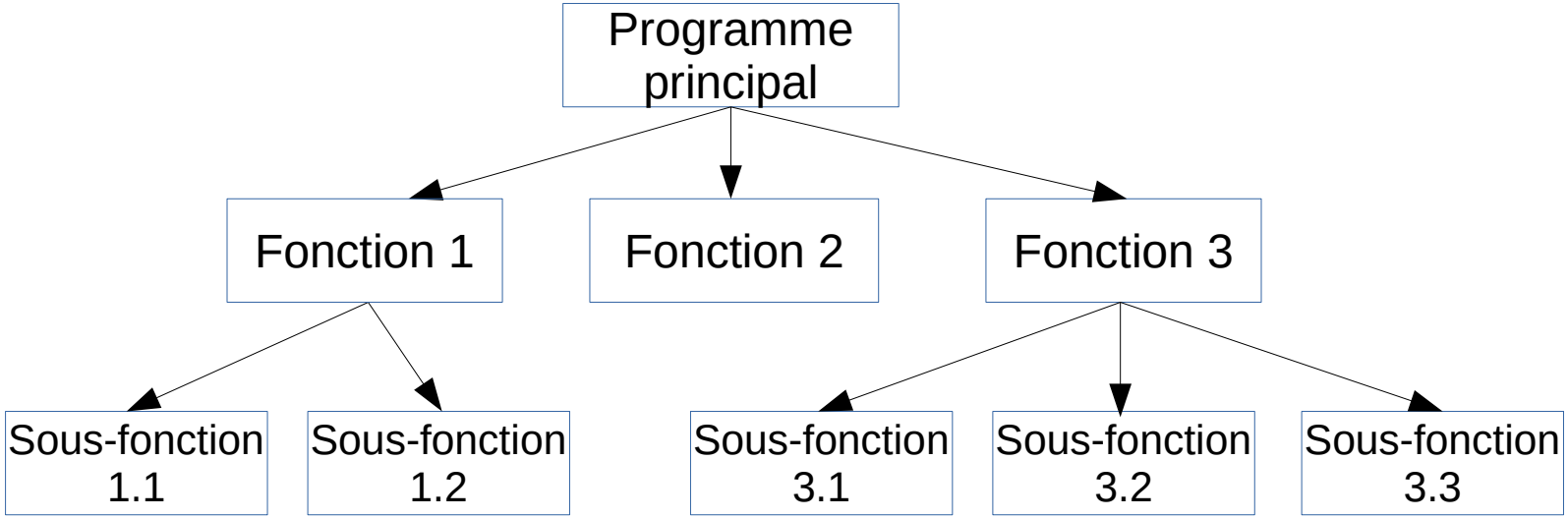
Plan du module “Conception orientée objet”

- L’approche procédurale
- L’approche orientée objet - introduction
- L’approche orientée objet - l’héritage
- L’approche orientée objet - l’encapsulation
- L’approche orientée objet - classes abstraites
- L’approche orientée objet - l’interface
- L’approche orientée objet - comment faire ?

La conception orientée objet

L’approche procédurale

L’analyse d’un problème



Exemple : affichage de formes

Réalisation d’un programme permettant d’afficher à l’écran différentes formes graphiques : cercle, rectangle, carré.

Le programme doit permettre à l’utilisateur de choisir la forme à afficher, et doit permettre d’afficher autant de formes que l’on veut.

Quelles sont les fonctions à réaliser ?

Les fonctions principales

Saisie du choix de la forme à afficher.

Saisie des paramètres de la forme à afficher.

Affichage de la forme.

Boucle principale (pour pouvoir saisir autant de formes que l’on veut).

Affinement des fonctions

Saisie du choix de la forme à afficher :

Affinement des fonctions

Saisie du choix de la forme à afficher :

⇒ un sous-programme retournant un nombre représentant le type de la forme à afficher

Affinement des fonctions

Saisie du choix de la forme à afficher :

⇒ un sous-programme retournant un nombre représentant le type de la forme à afficher

⇒ le nombre associé à une forme doit être fixe et invariable ; par exemple :

- 1 pour le rectangle
- 2 pour le carré
- 3 pour le cercle

Saisie du choix de la forme

```
final int RECTANGLE = 1;   // Enumération plutôt...
final int CARRE = 2;
final int CERCLE = 3;
…
public static int choixForme() {
    int resultat;
    …
    return resultat;
}
```

Affinement des fonctions

Saisie des paramètres de la forme à afficher :

Affinement des fonctions

Saisie des paramètres de la forme à afficher :

⇒ un sous-programme pour saisir les paramètres d’un rectangle

⇒ un sous-programme pour saisir les paramètres d’un carré

⇒ un sous-programme pour saisir les paramètres d’un cercle

Affinement des fonctions

Saisie des paramètres d’un rectangle :

⇒ un sous-programme saisissant chaque paramètre d’un rectangle

- abscisse de l’origine
- ordonnée de l’origine
- longueur du rectangle
- largeur du rectangle

Affinement des fonctions

Saisie des paramètres d’un rectangle :

⇒ un sous-programme saisissant chaque paramètre d’un rectangle

- étant donné que le sous-programme ne peutr retourner qu’une seule valeur, ces paramètres sont enregistrés dans des variables globales (i.e. définies à l’extérieur du sous-programme de saisie)

Extrait du programme

```
// Variables "globales"

int xOrigineRectangle, yOrigineRectangle, longueurRectangle,
largeurRectangle;

float xOrigineCarre, yOrigineCarre, coteCarre;

float xCentreCercle, yCentreCercle, rayonCercle;

public static void main(String[] arguments) {

    … // saisie du type de forme

    …

}
```

Extrait du programme

```
// Sous-programme de saisie des paramètres d'un rectangle
void saisieParametresRectangle() {
   Scanner saisie = new Scanner(System.in);
   xOrigineRectangle = saisie.nextFloat();
   yOrigineRectangle = saisie.nextFloat();
   longueurRectangle = saisie.nextFloat();
   largeurRectangle = saisie.nextFloat();
}
```

Affinement des fonctions

Affichage de la forme :

Affinement des fonctions

Affichage de la forme :

⇒ sous-programme d’affichage d’un rectangle

⇒ sous-programme d’affichage d’un carré

⇒ sous-programme d’affichage d’un cercle

Affinement des fonctions

Affichage de la forme d’un rectangle :

⇒ sous-programme d’affichage d’un rectangle

- à partir des valeurs enregistrées dans les variables globales (abscisse de l’origine, ordonnée de l’origine, longueur et largeur)

Extrait du programme

```
void afficheRectangle() {

   … // Utilisation des variables xOrigineRectangle,
yOrigineRectangle, longueurRectangle et largeurRectangle

}

void afficheCarre() {

   … // Utilisation des variables xOrigineCarre,
yOrigineCarre et coteCarre

}

…
```

Affinement des fonctions

Sous-programme principal du programme :

Affinement des fonctions

Sous-programme principal du programme :

⇒ boucler tant que l'utilisateur ne veut pas terminer le programme

- pour chaque itération de la boucle :
  - demander à l'utilisateur de saisir le type de forme à afficher
  - demander à l'utilisateur de saisir les paramètres de cette forme
  - afficher la forme

Affinement des fonctions

Exemple de solution : voir le projet NetBeans `ProcedureFormes`, classe `ProcedureFormes_1`.

Réutilisation d’un sous-programme

Supposons que l’on veuille réaliser un second programme gérant des formes. Là aussi, il faut saisir les paramètres des formes.

Cette fois cependant, les formes s’enrichissent d’un nouveau paramètre : la couleur du trait.

Comment réutiliser les sous-programmes de saisie des paramètres ? Comment réutiliser la boucle principale? En gros : quels sont les **impacts** de cette modification ?

Réutilisation d’un sous-programme

Tic tac tic tac…

Réutilisation d’un sous-programme

Il faut :

- ajouter trois nouvelles variables globales
- modifier tous les sous-programmes de saisie des paramètres, pour inclure la saisie de la largeur du trait
- modifier tous les sous-programmes d’affichage de forme, pour inclure l’affichage de la largeur du trait.

La boucle principale n’est cependant pas modifiée.

Réutilisation d’un sous-programme

Voir le projet NetBeans `ProcedureFormes`, classe `ProcedureFormes_2`.

Réutilisation d’un sous-programme

Aurait-on pu conserver les sous-programmes de saisie de paramètres ou d’affichage, en ajoutant simplement la saisie ou l’affichage du nouveau paramètre en dehors de ces sous-programmes ?

Réutilisation d’un sous-programme

Tic tac tic tac…

Réutilisation d’un sous-programme

Oui, mais au prix d’une modification du sous-programme principal et l’ajout de nouveaux sous-programmes **devant traiter tous les types de forme**.

Voir le projet NetBeans `ProcedureFormes`, classe `ProcedureFormes_3`.

Extension du programme

Supposons maintenant que l'on veuille ajouter un nouveau type de forme à afficher : l'ellipse.

Une ellipse se définit par : les coordonnées de son centre, son grand axe et son petit axe.

Quels sont les **impacts** de cette modification sur le programme précédent ? En particulier sur la version dans laquelle on essaye de garder **réutilisables** (donc non modifiés) les sous-programmes de saisie et d'affichage des formes ?

Extension du programme

Tic tac tic tac…

Extension du programme

Il faut :

- ajouter des variables globales pour représenter les paramètres de l'ellipse
- ajouter un sous-programme pour saisir les paramètres de l'ellipse
- ajouter un sous-programme pour afficher l'ellipse
- modifier les instructions `switch(forme)` des sous-programmes « génériques » et de la boucle principale

Réutilisation d’un sous-programme

Voir le projet NetBeans `ProcedureFormes`, classe `ProcedureFormes_4`.

Inconvénients de l’approche procédurale

Donc en synthèse :

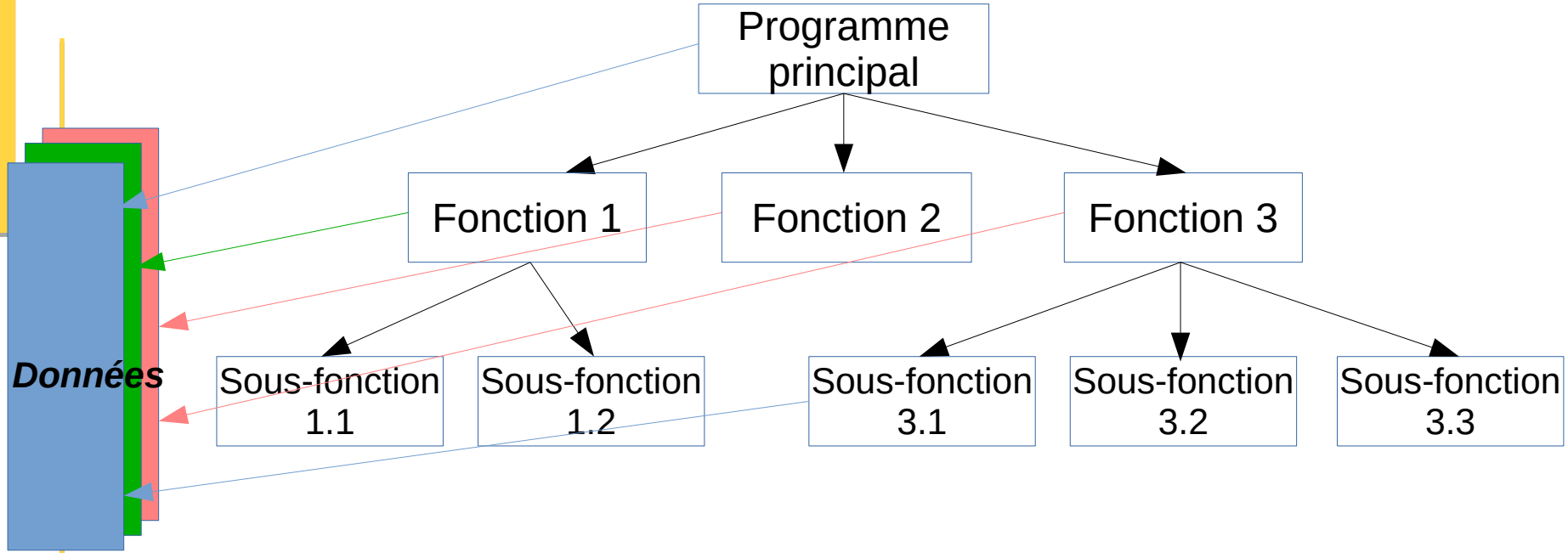
- un sous-programme « fonctionnel » est adapté au problème qu’il résoud, il est difficile de le généraliser et donc de le réutiliser
  - cas des fonctions de saisie des paramètres des figures
- ajouter de nouveaux cas de figure entraine de nombreuses modifications du code existant
  - cas de l’ajout d’une nouvelle figure (l’ellipse)

Principes de l’approche procédurale

L’approche procédurale part des principes suivants :

- l’analyse du besoin consiste à déterminer les traitements à réaliser
- les données sont séparées des traitements
  - exemple précédent : les variables sont globales, et les sous-programmes manipulent ces variables

La gestion des données



L’approche procédurale

Il est possible de concevoir les choses différemment...

La conception orientée objet

L’approche orientée objet - introduction

Les grandes lignes

L’approche orientée objet propose :

- de faire apparaitre les **entités du monde “réel”** dans les programmes
- de faire apparaître les **relations** entre ces entités
- de **regrouper** les données et les traitements sur ces données
- de **manipuler** les entités au travers d’une définition **abstraite** de ces entités
- de **masquer** certaines informations.

Les entités du monde réel

C’est le point le plus apparent de la COO :

- on fait apparaître dans un programme les entités réelles dont parle le cahier des charges
  - voiture, commande, facture, utilisateur, document, etc. ⇒ ce sont des **objets**
- ces entités sont simplifiées ⇒ on parle d’**abstraction** des objets réels.

Les entités du monde réel

Exemples d’abstraction :

- un rectangle affichable est considéré comme parfaitement défini par les propriétés suivantes :
  - abscisse et ordonnée de son point d’origine
  - longueur
  - largeur
  - largeur de son trait

Les entités du monde réel

Exemples d’abstraction :

- un cercle affichable est défini par :
  - l’abscisse et l’ordonnée de son centre
  - son rayon
  - la largeur de son trait

Les entités du monde réel

Exemples d’abstraction :

- une voiture est définie par :
  - sa marque, son modèle, son numéro de série
  - son type de carburant
  - son année de construction
  - son numéro d’immatriculation

Les entités du monde réel

Exemples d’abstraction :

- une moto est définie par :
  - sa marque, son modèle, son numéro de série
  - sa cylindrée
  - son année de construction
  - son numéro d’immatriculation

La conception orientée objet

L’approche orientée objet - l’héritage

Les relations entre entités

On s’intéresse en particulier à deux types de relation :

- la relation qui lie une entité à une autre entité du même type
  - relation « **est-un** » (« is-a » en anglais)
- la relation qui lie éventuellement une entité à l’entité qui la contient
  - relation d’**agrégation** (alias **composition**)

Les relations entre entités

Exemple de relation « is-a » :

- un rectangle **est** une forme géométrique
- un cercle **est** aussi une forme géométrique
- une voiture **est** un véhicule
- une moto **est** aussi un véhicule

Les relations entre entités

Quelle est la relation entre une voiture et un véhicule, ou entre une moto et un véhicule ?

Entre un rectangle et une forme géométrique, ou entre un cercle et une forme géométrique ?

Les relations entre entités

Tic tac tic tac…

Les relations entre entités

Une voiture est **un genre de** véhicule.

Une moto est **un genre de** véhicule.

Ce sont des **types particuliers** de véhicule.

En informatique, on parle souvent de **sous-types** de véhicule.

Les relations entre entités

On peut donc définir une entité (un type) Véhicule.

On peut aussi définir un type Voiture.

Une Voiture « est-un » (is-a) Véhicule.

De la même manière, un Cercle « est-une » FormeGéométrique.

Les relations entre entités

On peut aussi définir un type Moteur, un type Roue, un type Volant, etc.

Une Voiture comporte (d'habitude) un seul Volant, un ou plusieurs Moteurs, et en général au moins 4 Roues.

Un Moteur, un Volant ou des Roues sont **contenus** dans une Voiture.

Un Moteur, un Volant ou des Roues ne sont **pas** des types de Voiture par contre, ni des types de Véhicule.

Les relations entre entités

La relation qui lie une Voiture au Volant qu'elle contient est une relation d'**agrégation** (alias **composition**).

Un peu de terminologie

Quelles sont les informations caractérisant une Voiture ?

- sa marque et son modèle
- son numéro d’immatriculation
- son type de carburant
- etc.
- et ce qu’on peut faire avec
  - la démarrer
  - l’arrêter
  - etc.

Un peu de terminologie

Est-ce que toutes les Voitures possèdent les mêmes caractéristiques (même marque, même modèle, même année de fabrication, etc.) ?

Un peu de terminologie

Non, bien sûr.

Il existe donc des voitures particulières : la mienne, celle de mon voisin, celles de mes collègues, etc.

Mais toutes ces voitures possèdent les caractéristiques d’une Voiture.

Les voitures « particulières » sont appelées des **instances** du type général **Voiture**.

Un peu de terminologie

En logiciel, on dit que les voitures particulières sont des **objets**.

Ces objets sont des instances du **type** Voiture.

Les objets sont donc des instances d’un type – voire de plusieurs types parfois.

Un peu de terminologie

On a parlé en algorithmique des types `int`, `float`, `double`, `char`, `String`, etc.

Ces types sont des **types prédéfinis** du langage Java.

Il est également possible de définir de nouveaux types dans le cadre d'un programme ou d'une librairie ; on parle alors de **classes**.

En Java, une classe est une définition de type.

Un peu de terminologie

On distingue les **types primitifs** des classes. (Les 8 types primitifs de Java sont : `byte`, `short`, `int`, `long`, `float`, `double`, `boolean` et `char`.)

Le type `String` est en fait une classe.

Création d'une instance

Rappel : une variable d'un type primitif **doit** être déclarée.

```
int i;
```

```
float temperature = 21.7f;
```

Une variable appartenant à une classe (une instance de cette classe donc) doit aussi être déclarée mais de plus l'instance doit être **créée** :

```
MaClasse objet = new MaClasse();
```

Création d’une instance

La création d’une instance de classe est réalisée en pratique par une méthode spéciale appelée **constructeur** de la classe**.**

C’est cette méthode qui est appelée lorsque l’instruction `new` est utilisée.

Un constructeur est une méthode sans type (ni `void` ni autre chose). Ce qui est logique car son rôle est simplement d’**initialiser** l’instance - typiquement **donner une valeur** aux attributs de cette instance.

Création d’une instance

Un constructeur porte **obligatoirement** le nom de la classe des objets qu’il permet de construire. Il peut prendre autant d’arguments que nécessaire.

Une classe peut contenir autant de constructeurs que jugé utile. La distinction de ces constructeurs se fera par… ?

Création d’une instance

Tic tac tic tac…

Création d’une instance

La distinction de ces constructeurs se fera par les **arguments**.

Cela permet de créer une instance même quand on ne dispose pas de toutes les informations utiles pour la créer.

Création d’une instance

```
class Cercle {
    // Champs
    float xCentre, yCentre, rayon;
    // Constructeur par défaut
    Cercle() {
        this.xCentre = 0.0f;
        this.yCentre = 0.0f;
        this.rayon = 0.0f;
    }
```

Création d’une instance

```
// Autre constructeur
Cercle(float xCentre, float yCentre, float rayon) {
   this.xCentre = xCentre;
   this.yCentre = yCentre;
   this.rayon = rayon;
}
```

Création d’une instance

Passer par un constructeur est obligatoire pour créer une instance. Mais il n’est pas obligatoire d’en définir un, le compilateur en crée un à la volée si nécessaire (constructeur **par défaut**, i.e. **sans** arguments).

Le but d’un constructeur est de s’assurer que l’instance créée est dans un **état correct**, c’est-à-dire que toutes ses données (variables) ont des valeurs **adaptées** et **cohérentes**.

Création d’une instance

Corollaire :

Un constructeur se doit d’initialiser chacun des attributs de l’instance.

Création d’une instance

Pourquoi est-ce aussi important ?

Création d’une instance

Pas question (sauf cas exceptionnel) de n’initialiser que **quelques uns** des attributs, voire **aucun**, et d’espérer que le créateur de l’instance pensera à initialiser les autres attributs – typiquement au moyen d’un **setter** (modificateur).

Malheureusement, cette grave erreur est **souvent** commise, y compris par des développeurs soit-disant expérimentés.

Création d’une instance

NB. Cette règle souffre des exceptions, typiquement lorsque les instances sont **créées automatiquement par un framework**, genre Spring ou JEE.

En effet, les valeurs des attributs peuvent alors être définies dans un second temps, après création d’une instance « vide », par des méthodes dédiées appelées **setters** ou **modificateurs** : on parle alors d’**injection de dépendances** ⇒ voir le cours Spring.

Création d’une instance

L’avantage d’un framework, c’est qu’il fait les choses automatiquement, et que donc l’erreur humaine (risque d’oublier d’invoquer un setter) disparaît.

En synthèse :

- Vous utilisez un framework d’injection de dépendances ? Vous pouvez définir des constructeurs n’initialisant pas tous les attributs, **à condition** de définir les setters nécessaires
- Sinon ? Alors, il est plus sage de définir des constructeurs qui initialisent **tous** les attributs

Retour sur le concept de classe

On a vu qu'un programme conçu de manière procédurale distingue les traitements des données sur lesquelles s'appliquent ces traitements.

En COO, une classe définit à la fois ce qui **représente** une entité (ses données) et les **traitements** que supporte cette entité (ses sous-programmes).

En COO, les sous-programmes attachés à une entité sont appelés des **méthodes**.

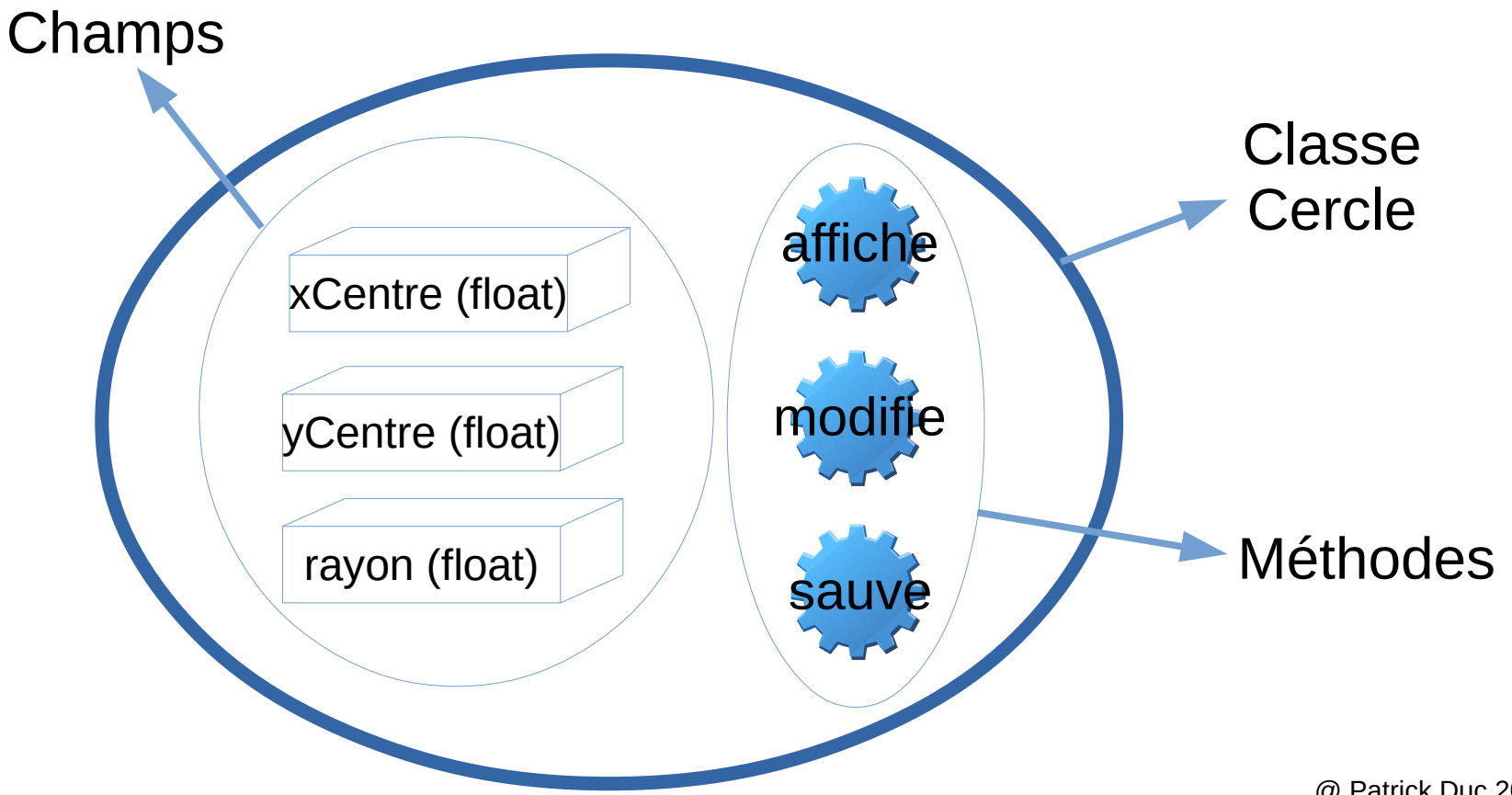
Retour sur le concept de classe

Une classe regroupe donc :

- des variables, souvent appelées **champs** (**fields** en anglais) ou **attributs** (**attributes** en anglais) ou **propriétés** (**properties** en anglais)
- et des méthodes, parfois appelées **services**.

Les variables et méthodes d'une classe forment les **membres** de cette classe.

Contenu d’une classe



Contenu d'une classe

```
class Cercle {
    // Champs
    float xCentre, yCentre, rayon;

    // Méthodes
    public void affiche() { … }
    public void modifie(float x, float y, float rayon) { … }
    public void sauve(String nomDeFichier) { … }
}
```

La notion de classe dérivée

On a vu qu'on peut définir une relation **is-a** entre par exemple :

- une Voiture et un Véhicule
- une Moto et un Véhicule
- un Rectangle et une FormeGéométrique
- un Cercle et une FormeGéométrique

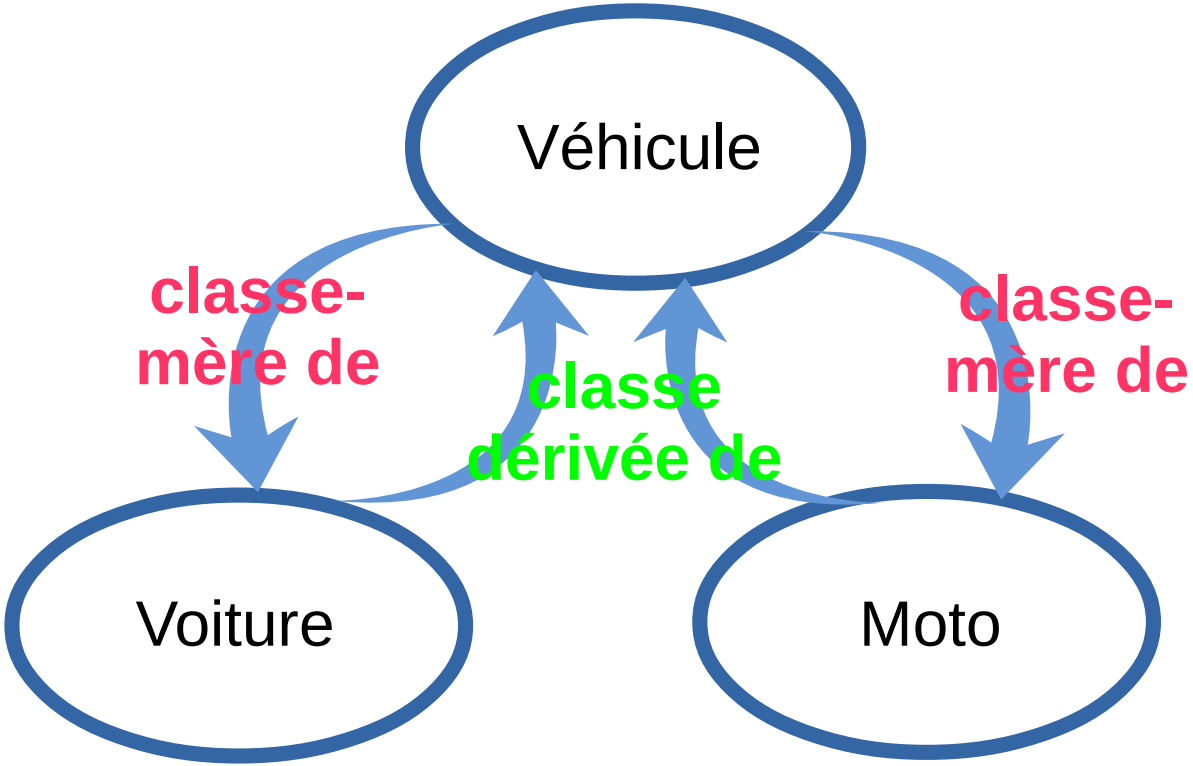
La notion de classe dérivée

Un Véhicule représente un concept plus général qu'une Voiture ou une Moto.

On dit que les classes Voiture et Moto sont des **classes dérivées** de la classe Véhicule.

La classe Véhicule représente la **classe-mère** des classes Voiture et Moto.

La notion de classe dérivée



Notation UML

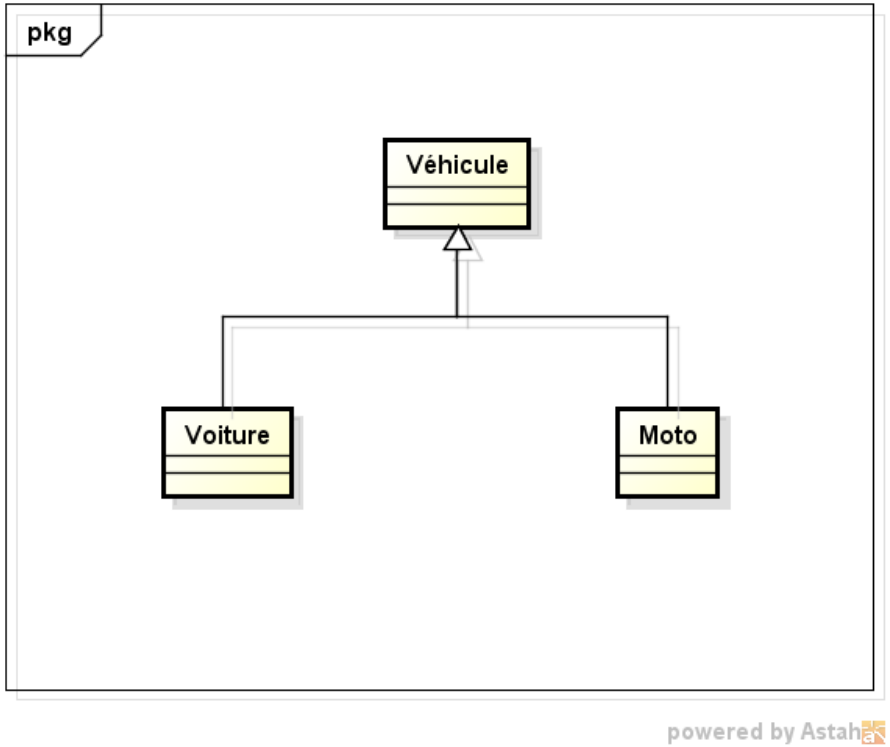


Diagramme de classe

La notion de classe dérivée

La relation de dérivation de classe en Java s'exprime au moyen du mot-clé `extends` (« étend » en français).

On parle aussi de relation de **généralisation** ou de **spécialisation**.

La notion de classe dérivée

```
class Forme {

    …

}

class Cercle extends Forme {

    …

}

class Rectangle extends Forme {

    …

}
```

La notion d’héritage

Considérons qu’un véhicule a toujours un moteur, un type de carburant, une année de fabrication et un numéro d’immatriculation. On peut aussi toujours le démarrer ou l’arrêter.

Une voiture a aussi un volant, alors qu’une moto a un guidon.

Ces deux types de véhicule partagent donc certaines caractéristiques, mais pas toutes.

La notion d’héritage

Certaines informations sont donc associées à un Véhicule :

- moteur
- type de carburant
- année de fabrication
- numéro d’immatriculation
- démarrer
- arrêter

La notion d’héritage

Une Voiture étant un Véhicule, elle possède une valeur pour chacun de ces champs :

- moteur
- type de carburant
- année de fabrication
- numéro d’immatriculation
- démarrer
- arrêter

C’est la même chose pour une Moto.

La notion d’héritage

La relation is-a qui relie une Voiture à un Véhicule fait que la Voiture dispose automatiquement de ces attributs: on dit qu’une Voiture **hérite** de ces attributs de la classe Véhicule.

C’est la même chose bien sûr pour une Moto.

Autrement dit, il n’est pas **nécessaire** de définir ces attributs dans les classes dérivées…

La notion d’héritage

```
class Vehicule {
    Moteur moteur;
    int anneeDeFabrication, typeDeCarburant;
    String numeroDImmatriculation;
    void demarrer() { … }
    void arreter() { … }
    void sauver() { … }
}
```

La notion d’héritage

```
class Voiture extends Vehicule {

   Volant volant;

}


class Moto extends Vehicule {

   Guidon guidon;

}
```

La notion d’héritage

```
class Voiture extends Vehicule {

   Volant volant;

   …

   void afficher() {

      System.out.println(”Voiture fabriquee en ” + anneeDeFabrication + ” et avec comme numero d’immatriculation ” + numeroDImmatriculation);

   }

}
```

La notion d’héritage

```
class Programme {

   public static void main(String[] arguments) {

      Voiture voiture = new Voiture();

      voiture.afficher();  // Accès aux champs de la classe
mère

      voiture.demarrer();  // Héritage des méthodes aussi

      voiture.anneeDeFabrication = 2018;

   }

}
```

La notion d’héritage

Dans l’exemple précédent, les champs `anneeDeFabrication` et `numeroDImmatriculation` de la classe `Vehicule` sont accessibles dans sa classe dérivée `Voiture`, qui a hérité de ces champs.

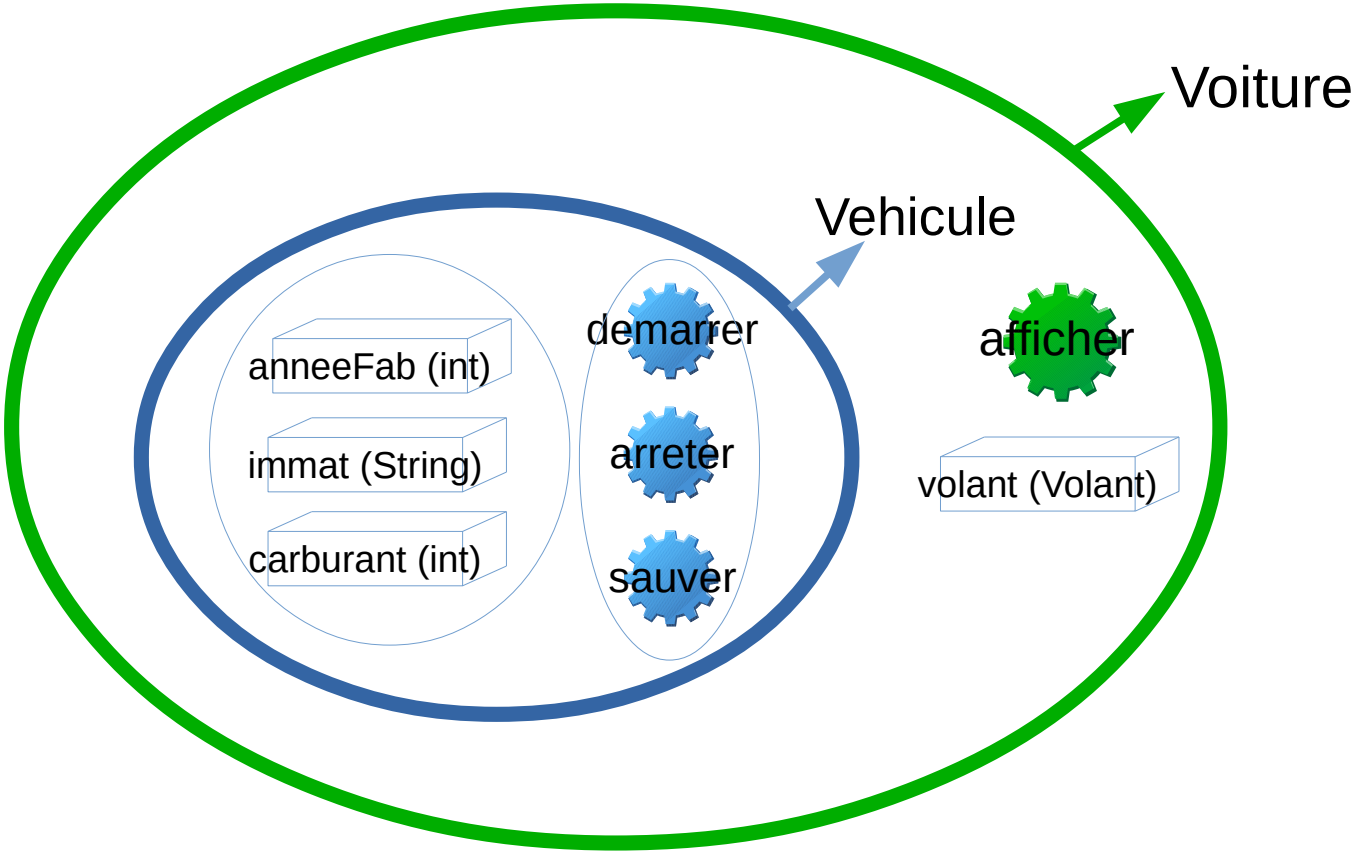
De même, la classe `Voiture` a hérité des méthodes de sa classe-mère, comme `demarrer()`.

Chaque instance de Voiture aura **ses propres variables** `anneeDeFabrication` et `numeroDImmatriculation`.

La notion d’héritage

```
public static void main(String[] arguments) {
    Voiture voiture1 = new Voiture();
    Voiture voiture2 = new Voiture();
    voiture1.anneeDeFabrication = 2018;
    voiture1.demarrer();
    voiture2.anneeDeFabrication = 2006;
    voiture2.arreter();
}
```

La notion d’héritage



La notion d’héritage

Questions :

- dans la figure précédente, quelle est la relation entre la classe `Vehicule` et la classe `Voiture` ?
- dans la figure précédente, quelle est la relation entre la classe `Voiture` et le champ `volant` ?

La notion d’héritage

Remarque importante : on peut affecter une instance d’une classe dérivée à une variable représentant une instance de la classe-mère.

Seuls les membres définis dans la classe-mère seront accessibles via cette variable.

La notion d'héritage

```
public static void main(String[] arguments) {
    Voiture voiture = new Voiture();
    voiture.anneeDeFabrication = 2018;
    voiture.demarrer();
    Vehicule vehicule = voiture;
    vehicule.arreter();
    vehicule.afficher();  // Erreur de compilation !
}
```

Exercice

On veut réaliser un programme permettant de gérer des comptes bancaires.

Ce programme doit permettre de définir et afficher différents types de compte : compte de dépôt, compte Moneo, compte d'épargne, compte sécurisé et compte jeune.

Un compte de dépôt a un propriétaire (identifié par un simple nom) et permet l'alimentation du compte (dépôt) et le retrait d'argent. On peut connaître le solde du compte de dépôt, qui peut être négatif.

Exercice

Un compte Moneo possède en plus un numéro de carte Moneo (simple entier).

Un compte d'épargne permet le dépôt d'argent et le retrait d'argent, et le solde du compte rapporte des intérêts à un taux donné. Le solde d'un tel compte ne peut jamais être négatif.

Un compte sécurisé est un compte de dépôt dont le solde ne peut être inférieur à un montant donné (positif ou négatif : réserve minimale ou découvert autorisé).

Exercice

Enfin, un compte jeune est un compte de dépôt dont le solde ne peut être que positif.

Quelles classes doit-on définir dans ce programme ?
Quelles seront les relations entre ces différentes classes ?

Exemple de solution

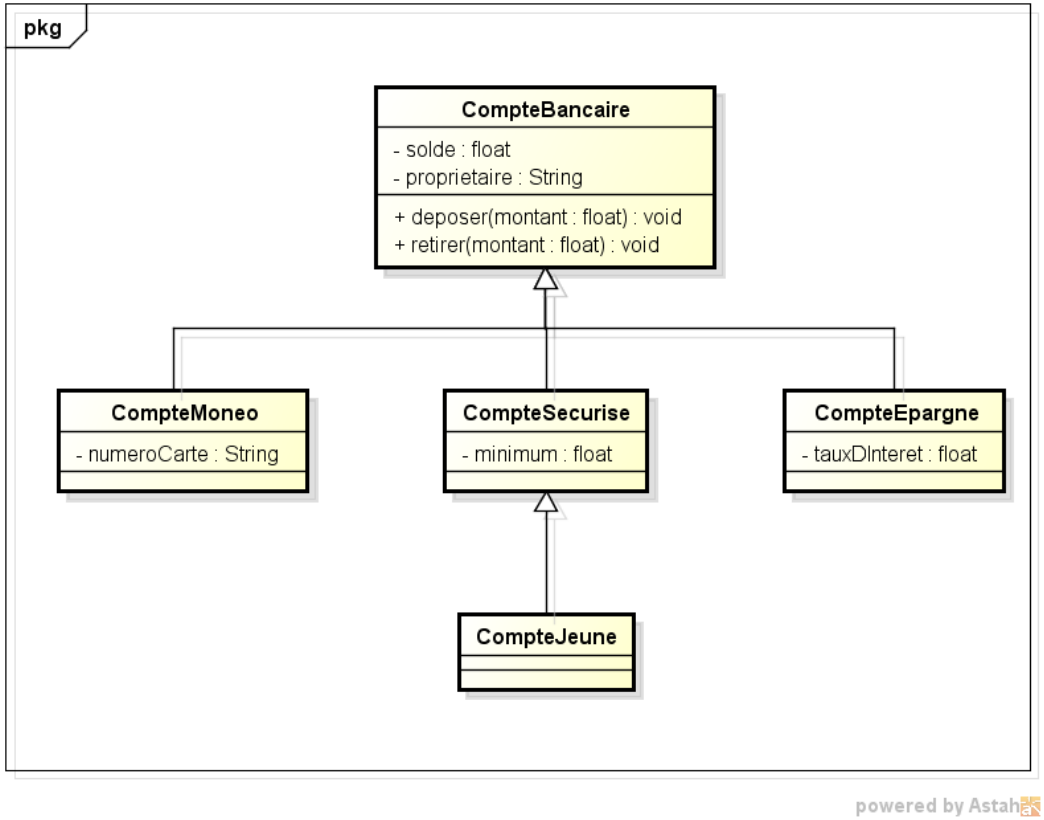


Diagramme de classe

Redéfinition de méthodes dérivées

Les méthodes définies dans une classe-mère peuvent être **redéfinies** dans une classe dérivée.

En Java, la signature de la méthode doit être la même que celle de la méthode dans la classe-mère pour qu'il y ait redéfinition, sinon il y a simplement définition d'une **nouvelle** méthode dans la classe dérivée.

Redéfinition de méthodes dérivées

Exemple : retour sur les formes géométriques.

On veut afficher à l'écran le type d'une forme géométrique.

Si la forme géométrique est une `FormeGéométrique`, le message sera "Forme géométrique".

Si la forme géométrique est un `Rectangle`, le message sera "Rectangle".

Si la forme géométrique est un `Cercle`, le message sera "Cercle".

Redéfinition de méthodes dérivées

```
public class FormeGeometrique {
    void afficher() {
        System.out.println(“Forme géométrique”);
    }
}
```

Redéfinition de méthodes dérivées

```
public class Rectangle extends FormeGeometrique {
    void afficher() {
        System.out.println(“Rectangle”);
    }
}
```

Redéfinition de méthodes dérivées

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

      FormeGeometrique formeGeometrique = new
FormeGeometrique();

      formeGeometrique.afficher();

      Rectangle rectangle = new Rectangle();

      rectangle.afficher();

      Cercle cercle = new Cercle();

      cercle.afficher();

}
```

Redéfinition de méthodes dérivées

Résultat affiché :

Forme géométrique

Rectangle

Cercle

Redéfinition de méthodes dérivées

Retour sur l’affectation d’une instance de classe dérivée à une variable du type de la classe de base.

Redéfinition de méthodes dérivées

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

      FormeGeometrique formeGeometrique = new
FormeGeometrique();

      formeGeometrique.afficher();

      Rectangle rectangle = new Rectangle();

      rectangle.afficher();

      formeGeometrique = rectangle;

      formeGeometrique.afficher();

}
```

Redéfinition de méthodes dérivées

Résultat affiché :

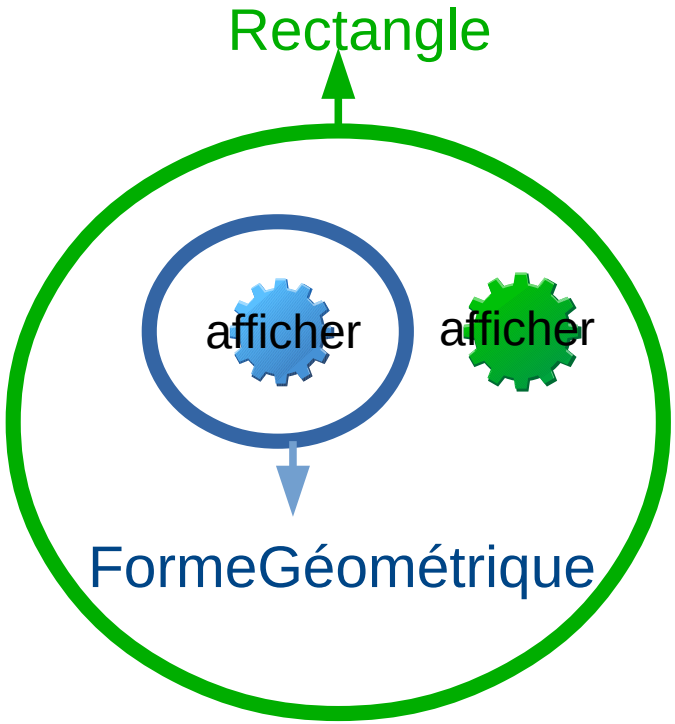
Forme géométrique

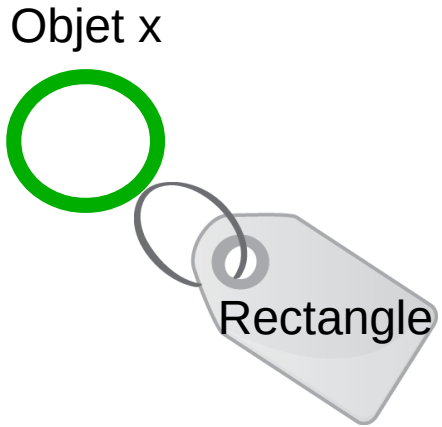
Rectangle

Rectangle

Autrement dit, le type **réel** de l'objet sera utilisé, pas celui de la variable qui indique l'instance.

Redéfinition de méthodes dérivées






```
FormeGeometrique x;
x = …;
x.afficher();
```

Le résultat affiché sera :
Rectangle

Type réel d'un objet

Le type **réel** d'un objet est celui qui est indiqué dans l'instruction `new`.

```
FormeGeometrique x = new Rectangle();
```

Ici, le type réel de x ne sera pas `FormeGeometrique` mais `Rectangle`, puisque c'est un objet de type `Rectangle` qui a été créé.

Avertissement

La notion d’héritage est simple à comprendre et très utile. **Attention cependant à ne pas en abuser**, et à rechercher une relation **is-a** là où elle n’existe pas vraiment. Le risque serait de tomber dans une hiérarchie de classes complexe et nécessitant l’utilisation de l’**héritage multiple**.

Une hiérarchie de classes complexes est contraire au principe de **simplicité** d’un programme. L’héritage multiple est source de casse-têtes et d’erreurs.

Alternative à l’héritage complexe

NB. Java interdit l’héritage multiple.

Il vaut souvent mieux utiliser une **relation de composition** pour représenter la relation entre plusieurs concepts que d’utiliser la notion d’héritage.

Alternative à l’héritage complexe

Exemple : le cas d’une université.

Une université comprend un corps enseignant, du personnel administratif et des élèves. Ces trois types de personnes partagent des caractéristiques : nom, prénom, adresse, date de naissance, numéro de sécurité sociale, etc.

Ils diffèrent cependant : un enseignant perçoit un salaire, assure un certain nombre de cours / TP / TD, possède un numéro de contrat d’enseignant, etc.

Alternative à l’héritage complexe

Un élève ne perçoit pas de salaire mais paye des frais d’entrée, il est inscrit à un ensemble de cours, il possède une carte d’étudiant, etc.

Un membre du personnel administratif perçoit un salaire et assure un certain nombre d’activités mais n’assure pas de cours.

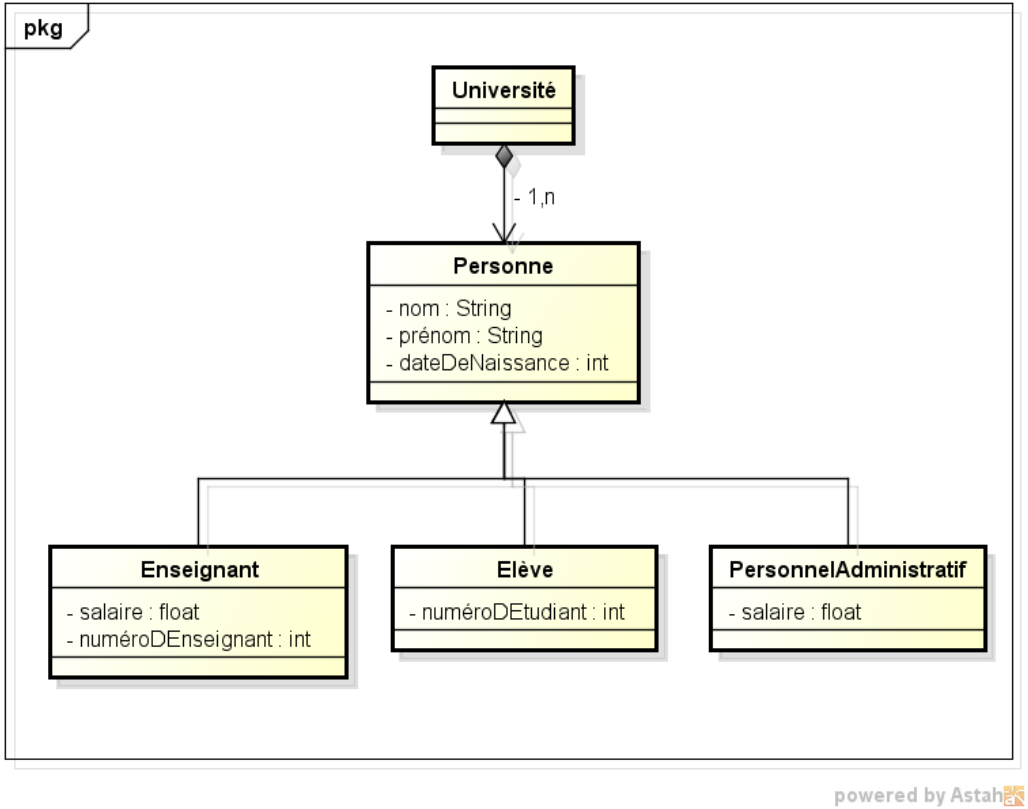
Pourquoi ne pas définir une classe `Personne` pour regrouper toutes les informations partagées (nom, prénom, date de naissance, adresse, numéro de sécurité sociale, etc.) ?

Alternative à l’héritage complexe

On pourrait ensuite définir une classe `Enseignant`, une classe `Elève` et une classe `PersonnelAdministratif`.

Bien évidemment, ces trois dernières classes dériveraient de la classe `Personne`.

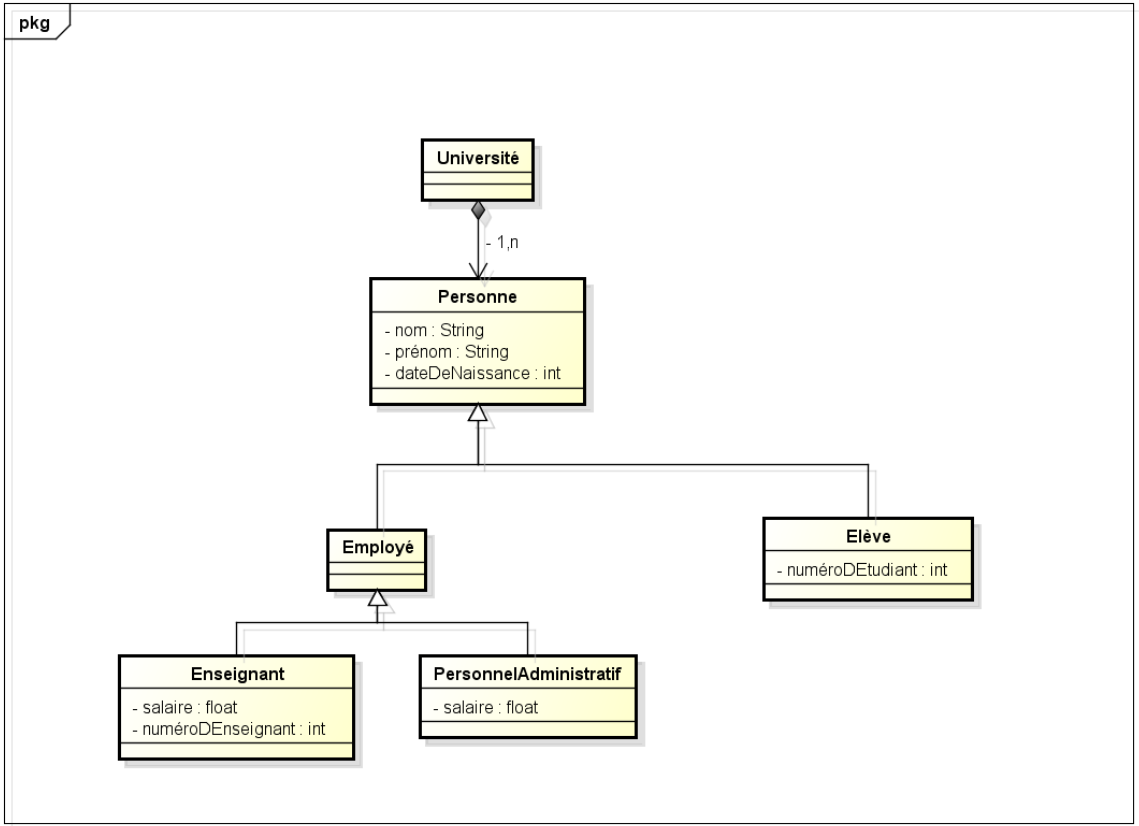
Alternative à l'héritage complexe



Alternative à l’héritage complexe

Et si on définissait une classe Employé dont dériveraient les enseignants et le personnel administratif ?

Alternative à l'héritage complexe

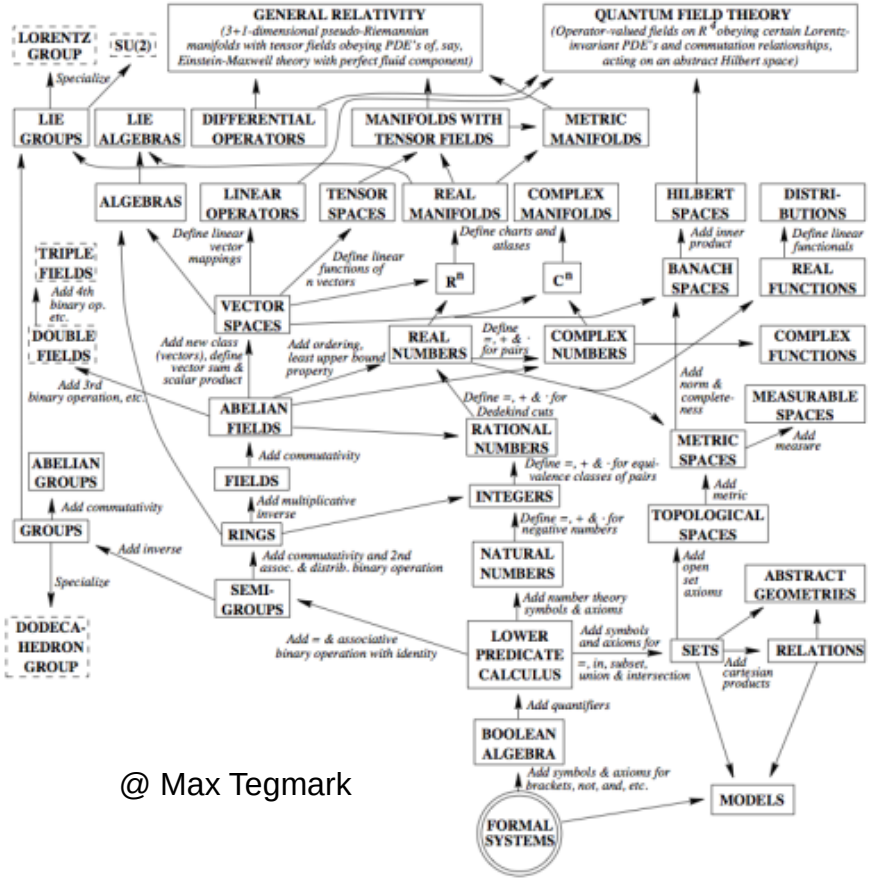


Alternative à l’héritage complexe

Mais quid des élèves travaillant pour l’Université afin de payer leurs études ?

Et quid des enseignants qui suivent des cours à l’Université ?

Alternative à l'héritage complexe



@ Max Tegmark

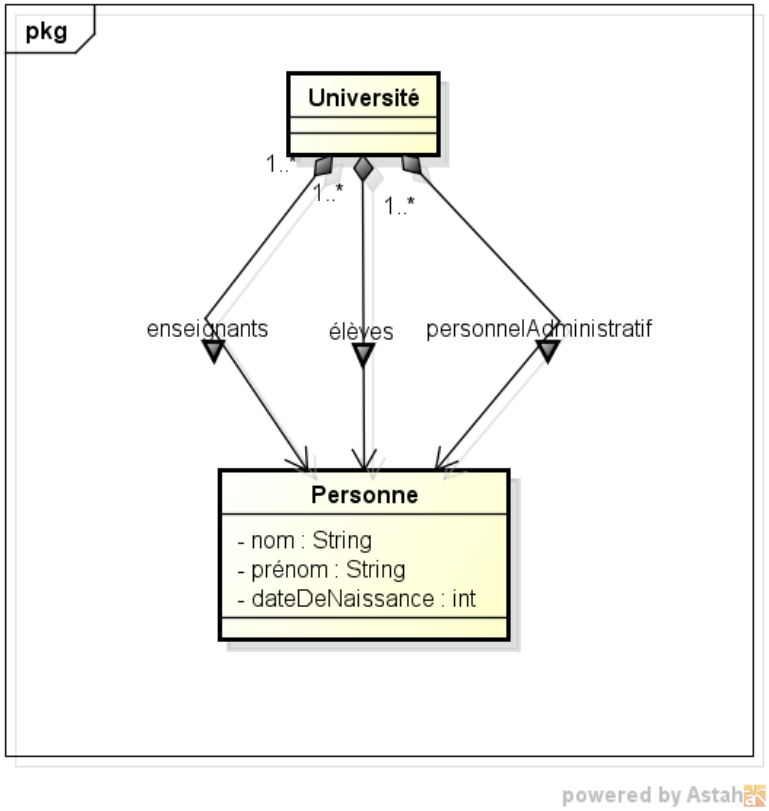
@ Patrick Duc 2018

Alternative à l’héritage complexe

Solution peut-être plus simple : utiliser des relations de composition entre l’université et les personnes qui la fréquentent.

On définirait ainsi une relation `enseignants`, une relation `élèves` et une relation `personnelAdministratif` reliant l’université à différentes personnes.

Alternative à l'héritage complexe



Alternative à l’héritage complexe

Ces relations pourraient être représentées par des dictionnaires avec comme clé une instance de `Personne`, et comme valeur une instance d’une classe `ContratDEnseignant`, `InfosElève` ou `ContratPersonnelAdministratif` mais sans qu’il y ait de relation de sous-typage entre ces classes et la classe `Personne`, ni entre elles.

Alternative à l’héritage complexe

```
class Universite {

   Map<Personne, ContratDEnseignant> enseignants;

   Map<Personne, InfosEleves> eleves;

   Map<Personne, ContratPersonnelAdministratif>
personnelAdministratif;

   …

}
```

Alternative à l’héritage complexe

```
class ContratDEnseignant {
    String numeroContrat;
    float salaire;
    List<ModulesDeFormation> modulesEnseignes ;
    …
}
```

Alternative à l’héritage complexe

```
class InfosEleve {

    String numeroCarteDEtudiant;

    float fraisDInscription;

    List<ModulesDeFormation> modulesSuivis;

    …

}
```

Alternative à l’héritage complexe

```
class ContratPersonnelAdministratif {
    String numeroContrat;
    float salaire;
    List<Taches> responsabilites;
    …
}
```

La conception orientée objet

L’approche orientée objet - l’encapsulation

Utilisation des objets

Un objet représente une **ressource** que peut utiliser un autre objet ou un sous-programme :

- ses données peuvent être lues ou écrites
- ses méthodes peuvent être appelées (on dit qu'elles sont **invoquées**)

Supposons qu'une classe permette de savoir quelle est la structure de donnée utilisée pour représenter une information dans la classe.

Utilisation des objets

Exemple : un programme de gestion de librairie. Ce programme permet d'ajouter des livres dans la bibliothèque, d'en retirer, et d'aficher son contenu.

Il a été défini une classe `Livre` et une classe `Bibliothèque`. Une bibliothèque (instance de `Bibliothèque`) enregistre des livres dans un tableau d'instances de `Livre`.

La classe permet de récupérer le tableau de livres et permet d'initialiser ou de remplacer le tableau de livres.

Un peu de terminologie

Le tableau de livres est un champ de la bibliothèque.

Une méthode permettant de lire (récupérer) la valeur d'un champ est appelé un **accesseur** (**getter** en anglais).

Une méthode permettant d'écrire (modifier) la valeur d'un champ est appelé un **modifieur** (**setter** en anglais).

Utilisation des objets

```
class Bibliotheque {

   Livre[] contenu;

   // Getter du champ ”contenu”

   Livre[] getLivres();

   // Setter du champ ”contenu”

   void setLivres(Livre[] livres);

}
```

Utilisation des objets

Le programme comporte un sous-programme qui ajoute des livres dans la bibliothèque et un sous-programme qui retire des livres de la bibliothèque.

Afin d’avoir de bonnes performances, le sous-programme qui ajoute des livres conserve une copie du tableau de livres.

Le sous-programme qui retire des livres fait de même.

Ces sous-programmes affichent le contenu de la bibliothèque en parcourant le tableau.

Utilisation des objets

```
class RetraitLivre {
   void retire(Bibliotheque bibliotheque, Livre livre) {
      // retrait du livre
      …
      // affichage ensuite
      for(Livre livreCourant : bibliotheque) {
         System.out.println("Livre : " + livre);
      }
   }
}
```

Utilisation des objets

L’utilisateur du programme effectue les opérations suivantes :

- ajout de 3 livres : livre1, livre2, livre3
- retrait de livre2
- ajout de 2 autres livres : livre4 et livre5
- retrait de livre3

Que contiendra la bibliothèque après ces opérations ?

Utilisation des objets

Réponse : la bibliothèque ne contiendra aucun livre !

Utilisation des objets

Pourquoi ?

Parce que la **mauvaise conception** du programme a permis une pratique dangereuse (*dans ce cas de figure*), à savoir garder une copie des données.

Le problème de conception est que les utilisateurs de la classe `Bibliothèque` peuvent accéder **directement** à la structure de données sous-jacente à la `Bibliothèque`, à savoir le tableau de livres.

L’encapsulation

Comment peut-on améliorer la conception de ce programme ?

L’encapsulation

Tic tac tic tac…

L’encapsulation

On peut masquer le tableau de livres, c’est-à-dire ne pas le rendre accessible depuis une classe utilisatrice de la classe `Bibliothèque`.

Le tableau existerait toujours dans la classe, mais les utilisateurs de la classe (les autres classes et sous-programmes) ne le « verraient » plus.

Plus moyen du coup de récupérer une copie du tableau de livres, donc la source de l’erreur disparaitrait.

L’encapsulation

Les utilisateurs manipuleraient la `Bibliothèque` uniquement au travers de ses méthodes :

- une méthode pour ajouter un livre
- une méthode pour supprimer un livre
- … et une méthode pour afficher le contenu de la bibliothèque, puisque parcourir le tableau ne serait plus possible ⇒ on évite ainsi la duplication de code

L’encapsulation

Masquer un membre d’une classe (champ ou méthode) se fait en indiquant le mot-clé `private` avant la déclaration du membre. C’est un qualificatif de **visibilité** d’un membre.

```
private int compteur;

private void maFonction(int nombre)
   { … }
```

L’encapsulation

Inversement, rendre visible de toutes les parties d’un logiciel un membre d’une classe (champ ou méthode) se fait en indiquant un autre qualificatif de visibilité d’un membre : le mot-clé `public`.

```
public String monIdentifiant;

public void ajoute(int nombre) { … }
```

L’encapsulation

```
class Bibliotheque {
    private Livre[] contenu;
    public ajouteLivre() { … }
    public retireLivre() { … }
    public affiche() { … }
}
```

L’encapsulation

Supposons qu’on veuille qu’une classe dérivée puisse accéder aux membres « internes », donc a priori `private`, de sa classe-mère.

Après tout, ces membres « font partie » de la classe dérivée, et il est donc licite que la classe-dérivée puisse manipuler ces membres.

L’encapsulation

L’indicateur de visibilité à utiliser dans ce cas-là est `protected`.

```
protected float longueur, largeur;

protected void setLivre(Livre livre)
   { … }
```

L’encapsulation

Cette pratique qui vise à masquer aux utilisateurs d’une ressource (i.e. une classe en Java) les détails d’implémentation de cette ressource est **fondamentale** !

A strictement parler, il ne s’agit pas d’une caractéristique de la conception orientée objet. Elle est apparue avec le concept de **type de données abstrait** (abstract data type en anglais), et est souvent appelée **information hiding** en anglais.

L’encapsulation

Si les utilisateurs d’une ressource savent comment elle est **implémentée**, c’est-à-dire comment elle esrt représentée en mémoire, comment ses données internes sont manipulables, alors toute **modification** de cette ressource se traduira par des impacts **potentiellement importants** sur les utilisateurs de cette ressource.

L’encapsulation

Inversement, si les utilisateurs ne savent pas comment est implémentée la ressource, s’ils ne peuvent la manipuler qu’au travers de méthodes **attachées** à la ressource, alors ils ne courent pas le risque d’être impactés par une modification de la façon dont la ressource est impémentée.

De plus, la **cohérence interne** de la ressource est plus simple à assurer car seules les méthodes attachées à la ressource la manipulent.

L’encapsulation

Dans l’exemple de la bibliothèque, la représentation interne de la `Bibliothèque` est masquée aux utilisateurs, et sa manipulation est faite uniquement au moyen de méthodes de la classe `Bibliothèque`.

L’encapsulation

Cette pratique consistant à masquer le plus d’informations inutiles pour les utilisateurs peut s’étendre et se généraliser aux **classes**, pas seulement aux **membres** d’une classe.

L’encapsulation

Les membres (champs et méthodes) **publics** d’une classe représentent l’**interface** de cette classe.

En UML, les membres publics sont marqués d’un caractère **+**. (Les membres privés sont marqués du caractère **-**, les membres protégés sont indiqués par le caractère **~**).

L’encapsulation

Une librairie logicielle est un ensemble de classes fournissant des services divers.

L’ensemble des interfaces des classes de la librairie représente l’**API** (**Application Programming Interface**) de cette librairie.

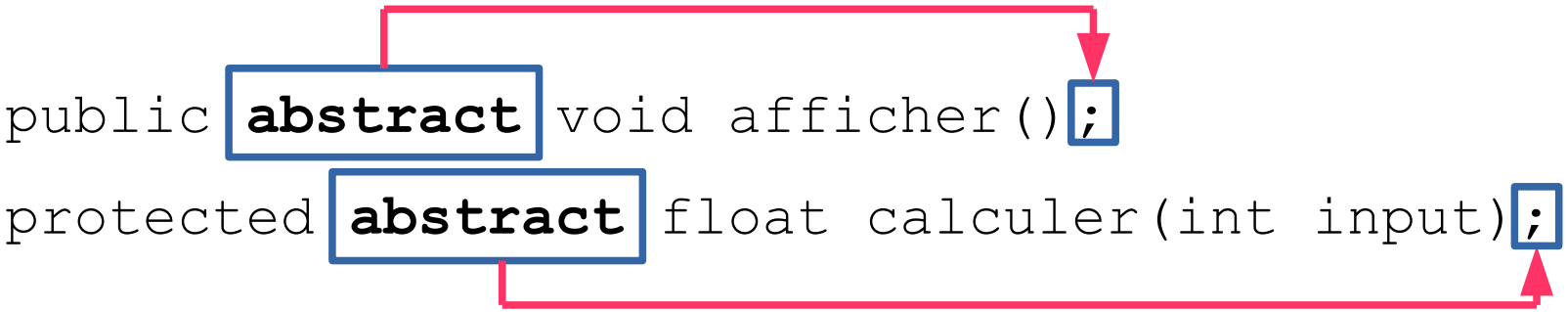
La conception orientée objet

L’approche orientée objet – classes abstraites

Les méthodes abstraites

Il est possible de déclarer une méthode, mais sans lui associer un traitement.

Cette méthode est alors considérée comme abstraite (`abstract` en anglais).



Les classes abstraites

Important : les classes comportant au moins une méthode abstraite **ne peuvent pas** être instanciées (i.e. **aucune instance** de cette classe ne peut être créée). Elles doivent être déclarées **abstraites** elles aussi.

Normal, le compilateur ne saurait pas quoi faire si une méthode abstraite d’une telle instance était invoquée...

Exemple de classe abstraite

Fichier `FormeGeometrique.java`

```
abstract class FormeGeometrique {

   protected abstract affiche();

}
```

Utilisation d’une classe abstraite

Fichier `Formes.java`

```
class Formes {

   public static void main(String[] args) {

      // Erreur de compilation ci-dessous

      FormeGeometrique forme = new FormeGeometrique();

   }

}
```

Utilisation d’une classe abstraite

Les méthodes abstraites sont implémentées si nécessaire dans les classes dérivées :

- “si nécessaire” ⇒ si on veut que les classes dérivées soient instanciables

Le principe est de déléguer l’implémentation de certaines méthodes aux classes qui savent comment elles doivent être implémentées

⇒ l’implémentation est spécifique de chaque classe

Utilisation d’une classe abstraite

Une classe qui implémente toutes les méthodes abstraites de sa classe-mère (éventuellement héritées par la classe-mère elle-même) est appelée une classe **concrète**.

Une classe concrète peut être instanciée.

Intérêt des classes abstraites

Exemple des formes géométriques.

On veut realiser un programme qui permet de calculer le périmètre et la surface de certaines formes géométriques :

- rectangle
- carré
- cercle

Chaque forme géométrique possède au minimum un point d’origine avec une abscisse et une ordonnée.

Intérêt des classes abstraites

De plus, il est possible d’afficher les détails de chaque classe abstraite : abscisse et ordonnée d’origine, longueur, largeur, côté, surface, périmètre…

La fonction principale doit créer un exemplaire de chaque forme géométrique et calculer son périmètre et sa surface, puis afficher les détails de la forme.

Intérêt des classes abstraites

Première approche.

On définit une classe de base pour contenir les informations communes à chaque forme géométrique (abscisse et ordonnée du point d'origine), et on en dérive une classe par forme géométrique :

- rectangle
- carré
- cercle

Intérêt des classes abstraites

Voir le projet NetBeans `Formes`.

Intérêt des classes abstraites

Seconde approche.

On définit une classe de base **abstraite** pour contenir les informations communes à chaque forme géométrique et déclarer les méthodes nécessaires (`perimetre()`, `surface()` et `afficher()`), et on en dérive une classe **concrète** par forme géométrique :

- rectangle
- carré
- cercle

Intérêt des classes abstraites

Voir le projet NetBeans `FormesPolymorphiques`.

Intérêt des classes abstraites

Dans le second cas, il n’est pas possible de créer des formes géométriques.

La propriété d’abstraction permet d’éviter de créer des objets qui n’ont pas vraiment de sens (des formes géométriques, des véhicules, ou autres objets qui sont abstraits, i.e. qui ne peuvent implémenter certaines méthodes bien précises : calcul de surface, démarrer, etc.).

Intérêt des classes abstraites

Une classe abstraite est donc utile si certaines informations et services (méthodes) peuvent être regroupés et partagés par différentes classes concrètes mais sans qu'on puisse rendre concrète la classe qui regroupe ces services.

Exemples : le véhicule et la forme géométrique.

Contre-exemple : le compte de dépôt.

Le concept d’interface

L’approche orientée objet - l’interface

Le concept d’interface

Supposons qu’on veuille pouvoir enregistrer sur disque les formes géométriques créées.

On peut définir pour chaque forme géométrique une méthode `enregistrer()` qui prendrait en argument un nom de fichier et enregistrerait dans ce fichier la forme géométrique.

Est-ce que cette méthode peut être implémentée dans une seule classe (classe `FormeGéométrique` par exemple) ?

Le concept d’interface

Non, car les informations à enregistrer sont spécifiques de chaque classe.

Cependant, on veut pouvoir manipuler les objets géométriques de manière générique, sans savoir de quel type précis il s’agit. (Ce qui est une bonne manière de **réduire la dépendance** d’une partie du programme vers une autre partie du programme).

Comment faire en conception orientée objet ?

Le concept d’interface

On pourrait définir une méthode abstraite `enregistrer()` dans la classe abstraite `FormeGeometrique`.

Ainsi, chaque classe dérivée pourrait implémenter cette méthode de manière adaptée.

Le concept d’interface

```
public abstract class FormeGeometrique {
    float xOrigine, yOrigine;
    abstract void enregistrer(String nomDeFichier);
}
public class Rectangle extends FormeGeometrique {
    void enregistrer(String nomDeFichier) {
        …
    }
}
```

Le concept d’interface

Question : est-ce que le fait d’enregistrer des informations dans un fichier est spécifique à des formes géométriques ?

Le concept d’interface

Non : on peut aussi enregistrer dans un fichier les informations relatives à une voiture ou une moto, à un compte de dépôt, à un compte Moneo, etc.

Le fait de pouvoir enregistrer une instance de classe n’est donc pas lié à la notion de relation `is-a`.

Il s’agit plutôt d’un « service » que ces classes doivent fournir.

Le concept d’interface

Il existe un concept en conception orientée objet pour représenter ce concept de service : l’**interface**.

Une interface représente la déclaration d’un ensemble de services.

Cette interface peut ensuite être « implémentée » par les classes qui désirent fournir ce service.

Il n’y a besoin d’aucune relation de type `is-a` ou composition entre l’interface et les classes qui l’implémentent.

Le concept d’interface

Une interface est uniquement une déclaration de services, elle n’implémente **pas** ces services.

Elle ne peut pas contenir de champs (variables), sauf des constantes (ou variables statiques).

Il s’agit d’une déclaration de contrat. Les classes qui annoncent implémenter cette interface s’engagent à remplir ce contrat.

Le concept d’interface

Point très important : la relation qui lie une interface à une classe qui l’implémente n’est pas une relation d’héritage (`is-a`) ou de composition.

Le concept d’interface

```
interface Enregistreur {

    void enregistrer(String nomDeFichier);

}
```

Le concept d’interface

```
public abstract class FormeGeometrique
implements Enregistreur {

    float xOrigine, yOrigine;

    abstract void enregistrer(String
nomDeFichier);

}
```

Le concept d’interface

```
public class Rectangle extends
FormeGeometrique {

    @Override

    void enregistrer(String nomDeFichier) {

        …

    }

}
```

La méthode de COO

L’approche orientée objet – comment faire ?

La méthode de COO

Les objets (instances de classe), l'héritage, l'encapsulation, les classes abstraites et les interfaces sont des concepts supportant la conception orientée objet.

Mais ils ne fournissent pas de **méthode** de conception orientée objet, c'est-à-dire qu'ils n'aident pas vraiment à **décider** quelles classes doivent être définies, quelles classes doivent hériter de telle autre, quelles méthodes doivent être définies dans une classe, **comment** agréger des informations pour représenter des abstractions du monde réel, quels services doivent être définis, etc.

La méthode de COO

Pour ce qui est du choix des classes à définir et de leurs méthodes, il est bon de se ramener au **cahier des charges**.

(S’il n’existe pas, le rédiger de manière simple).

La méthode de COO

Heuristique basique mais assez efficace :

- les noms communs qui reviennent souvent correspondent à des concepts qui sont de bons candidats pour être des classes
- les verbes associés à ces noms communs peuvent représenter des méthodes de ces classes
- même chose pour les expressions du type « permet de »

La méthode de COO

Heuristique basique mais assez efficace :

- les relations de composition peuvent être marquées par des verbes comme « contenir », « inclure », « avoir », « posséder », etc.
- les relations is-a sont souvent indiquées par des termes comme « genre de », « type de », « est un », etc.

La méthode de COO

Lors de la phase de conception d'un logiciel, attention à bien **distinguer** les concepts et les relations entre ces concepts :

- héritage (`is-a`)
- composition (contient)
- implémentation (réalisation d'interface)

La méthode de COO

Important : une interface (= ensemble de membres publics) d'une classe doit être définie de manière **la plus simple** possible.

La manière d'obtenir cela consiste à se placer du point de vue des utilisateurs (= développeurs) de cette interface.

Autrement dit, on leur évite de devoir faire des choses qui ne les intéressent pas : initialiser la classe, « nettoyer » la classe à la fin, faire des choses qui leur paraissent bizarres ou incompréhensibles ou compliquées...

Exercice

Lire le sujet du projet Fil Rouge et proposer les classes nécessaires pour le développer (champs, méthodes, relations avec les autres classes) et les interfaces (au sens de contrat) éventuellement utiles.

Définir le bon niveau d’encapsulation des membres.