2. Concepts évolués de la programmation objet

Cours Stéphane Loiseau, univ. Angers

- Introduction
  - Trois points de vue sur la notion d'objet
    - abstraction (type abstrait) -point de vue structure
      - cache les structures (ex: pile) par les manipulations légales (ex: empiler, dépiler)

      On parle *d'encapsulation* de l'objet.
    - réseaux sémantiques (Intelligence artificielle) -point de vue concept
      - sous classe SC *est_une_sorte-de classe* C
      - objet O *est_un (est_une_instance)* de C
    - langage d'acteurs (Intelligence artificielle) -point de vue dynamique
      - l'objet O reçoit un message et y réagit
      - -Système multi agents

  - Trois grandes classes d'applications
    - la conception = représenter le monde réel pour l'informatiser
      - représente les entités (classe) et les liens (héritage) naturellement
      - visualisation facile (graphe des classes) synthèse et modifications
    - la programmation = le codage
      - la programmation jaillit de la modélisation
      - les fonctions sont relatives aux données
      - un même nom de fonction pour même opération, même si codage différent (langage abstrait)
    - les bases de données = la persistance
      - pourquoi utiliser des tables ?
      - objets persistants

– Principes
  - principe 1 : on raccroche à la structure de données (champs), les codes qui peuvent les manipuler (méthode) -> classe
  - principe 2 : on hiérarchise les classes -> héritage
  - principe 3 : on assure la sécurité des accès aux données et méthodes sous forme d'autorisation; on gère les erreurs

- A. Propriétés des accès
  Qui peut accéder aux champs, méthodes, classes ?
  – 1. Le constructeur
    - appliqué à une classe retourne un objet de la classe
      – 1) Créer un objet à partir d'une classe :
        - `X <- nomClasse <= new()` /*LOLO
        - `X = new nomClasse()` `/*JAVA`
      – 2) Initialiser l'objet par les valeurs adéquates
        - Initialiser après le new les champs
    - Programmer des constructeurs paramétrés
      – (**LOLO**): `nomClasse <= new(val1,..,valn)`
      – (**en JAVA**): définir des méthodes du nom de la classe
        - quand aucun constructeur n'est défini -> constructeur par défaut
        - le constructeur par défaut est désactivé dès qu'il existe un constructeur programmé
        - on peut reconstruire à la main un constructeur sans paramètre

- L'utilisation de `super`
  - `super()` appel le constructeur de la sur-classe
  - `super.méthode(...)` appel de méthode de la classe supérieure

- 2. Accès privé/public
  - A. Accès aux champs et méthodes d'un objet de classe C
    - L'accès à un champs peut être
      - **`privé`**: seules les méthodes de la classe C peuvent utiliser le champs
      - **`public`**: tout code (ayant accès à la classe C) peut l'utiliser
      - **`protégé`**: seules les méthodes de la classe C et de ses sous classes peuvent utiliser le champs
    - Règles:
      - un champs est généralement privé
      - --> accès par des méthodes adéquates
      - un champs gérant des données d'implémentation est privé

- Accès aux méthodes m d'une classe C
  L'accès à une méthode peut être
  - **`privé`**: seules les méthodes de la classe C peuvent utiliser m
  - **`public`**: tout code (ayant accès à la classe C) peut utiliser m
  - **`protégé`**: seules les méthodes de la classe C et de ses sous classes peuvent utiliser m
- Remarques:
  - `private, public, protected` *(en JAVA)* sont appelés des *modificateurs* (d'accès)
  - Accesseurs (conventions)
    `getVar` : accès lecture à un champs `Var`
    `setVar` : accès écriture à un champs `Var`

- On parle de portée de déclaration

-3. Les **paquetages**

- un paquetage (en JAVA, n'existe pas en Smalltalk)
  - Idées
    - pour de grandes applications
    - regroupe des classes dans un paquetage (`package` en JAVA)
    - Nom de la classe, référé à un paquetage (permet 2 classes de même nom)
    - Les paquetages organisent logiquement les classes dans une hiérarchie; généralement, l'organisation physique est la même
  - Principes du package (logique) et de son positionnement système (physique)
    - Un package est associé à un répertoire :le nom du pakage est celui du répertoire
    - Une hiérarchie de packages correspond à une hiérarchie de répertoires
    - Dans un package, il y a des classes, et des sous packages (= sous répertoires)
    - Le premier niveau de hiérarchie correspond au nom de l'application ou du propriétaire (ex: `loiseau, java`)

- Principes de nommage, d'appartenance et de recherche
  - un mécanisme de nommage:
    `nomPackage.nomSousPackage.NomClasse`
  - appartenance
    - Par défaut: une classe dans un répertoire appartient au package correspondant
    - Par déclaration: `package nomPackage.nomSousPackage` (1ere instr)
  - recherche: variable `CLASSPATH` (Windows)

- Accès aux classes d'un package
  - les classes (`object`, `String`...) du package `java.lang` sont utilisables dans toute classe
  - pour importer une classe C ou un packetage P dans un programme:
    `import C` `import P.*` en tête de fichier
  - Si pas de modificateur de champs et méthode: toutes les classes du package y ont accès
  - Attention : Modificateur d'accès d'une classe C
    `public`: la classe est accessible en dehors du package où elle est définie
    pas de modificateur: seules les classes du même paquetage ont accès à C

• B. Les classes pour elles-même

– 0. Retour à l'héritage

- l'autoréférencement
  - pour appliquer une méthode à l'objet (déjà vu en 1.4)
  - pour lever l'ambiguïté entre nom de champs et variable
- L'option `final` (sécurité et performance)
  - une méthode `final` ne peut pas être redéfinie dans une sous classe
  - une classe définie `final` ne peut pas être héritée
  - une variable `final` est une constante

– 1. Classe abstraite

- Définition: une *classe abstraite* est une classe pour laquelle il ne peut exister d'instances.
  - elle factorise les champs et méthodes de sous classes
  - elle est déclarée avec le mot clef `abstraite` (*`abstract`*)
  - Ex:une classe dont les instances sont les règlements d une facture envoyés aux clients * il y a trois types de règlements: chèque, virement, traite * un règlement concerne un client, une facture, une date, et pour une somme * une méthode crédite correspond a l encaissement (renvoi la somme), mais dépend du mode de paiement (abstraite) * l encaissement crédite le compte bancaire de notre entreprise, * il existe plusieurs sorte de paiements mais champs communs et méthodes communes (contrôle...)
- Définition:une *méthode abstraite* d'une classe C est une méthode ne possédant pas de corps. La méthode doit être implémentée dans chaque sous-classe de la classe C
  - elle est redéfinie au niveau des sous classes
  - vérification à la compilation des redéfinitions obligatoires dans les Sous Classes
  - elle est déclarée avec le mot clef `abstraite` (`abstract`)
  - Vérification: Si C a une méthode abstraite => C est abstraite

– 2. Variables de classe et méthodes de classe

- Définition:
  une *variable de classe* est une variable rattachée à la classe.
  - elle existe donc qu'en un seul exemplaire en mémoire
  - elle est déclarée avec le modificateur *static*
  - elle peut être privé, public, protégé
  - elle est automatiquement crée en mémoire dès que la classe est chargée
- Définition:
  une *méthode de classe* est une méthode qui s'applique à la classe
  - elle est déclarée avec le modificateur *static*
  - elle peut être privé, public, protégé
  - elle peut avoir des arguments

– 3. Redéfinition, transtypage, polymorphisme, surcharge

- Redéfinition
  une méthode est *redéfinie* si elle est définie dans une classe et (re)définie dans une des classes dont elle hérite

- le transtypage
  - Définition:
    le *transtypage* est l'opération qui consiste à changer le type d'un objet
  - Il existe deux types de transtypage: l'implicite et l'explicite (cast).

  - Règle: un objet de classe SC, sous classe de C, est implicitement transtypé en objet de classe C si besoin est. La réciproque est fausse et provoque une erreur.

- le polymorphisme (en C++ : --)
  - Définition:
    le *polymorphisme* est la capacité à faire correspondre à un même message plusieurs méthodes selon le type de l'objet receveur; quand l'objet est stocké dans une variable de type général.
- Surcharge
  - Définition:
    la *signature* d'une méthode est composé du nom de la méthode, des types des paramètres formels de la méthode
  - Définition
    Il y a *surcharge (= surdéfinition)* lorsqu'une méthode est définie avec plusieurs signatures dans la même classe

  Remarque: souvent en programmation classique, on ne peut pas donner deux noms identiques à des fonctions ou procédures, même si les signatures différent.

- Les classes internes
  - Principe
    - une classe contenue dans une autre classe
  - Définition
    - une *classe interne CI* (par opposition à classe globale) est une classe déclarée dans une autre classe C
  - Règles
    - Seul C a accès à CI.
    - La création d ’une instance de CI ne peut se faire que dans une méthode de C
    - l ’instance de CI peut avoir accès aux champs de l ’objet O (de C) qui l ’a créée.
  - Intérêts
    - une classe interne CI est invisible aux autres classes que C
    - une classe interne CI peut accéder aux champs et méthodes de sa surclasse

- **4. Héritage multiple et interface**
  - Héritage multiple en C++,
    mais ni en Smalltalk, ni en JAVA (héritage simple)
  - Définition
    Quand une classe peut être sous classe de plusieurs classes, on dit que l'on a un *héritage multiple*
  - Problèmes
    soit SC une sous-classe de C1 et C2
    - si deux champs identiques dans super classes, duquel hérite l'objet ?
    - si deux méthodes identiques dans super classes, de laquelle hérite l'objet ?

Le concept d'interface

- Un concept
  - uniquement de modélisation de l'héritage multiple
  - Une interface est un concept qui définie des comportements sous forme de méthodes absraites
  - Ces comportements doivent être redéfinis dans les classes qui implémentent l'interface
- Caractéristiques d'une interface
    - Comme une classe abstraite, n'a pas d'instance
    - Toutes ses méthodes sont abstraites
    - Ne possède aucun champs
    - Peut avoir des variables dites d'interface (= variables de classes)
    - Une classe peut hériter, c'est à dire implémenter, autant d'interfaces que souhaitée
    - une interface ne peut hériter que d'interfaces
- Syntaxe: `interface nom_inter {...}`
  `classe C implements nom_inter {...}`

Le concept d'interface

- Un concept
  - uniquement de modélisation de la partie « sorte-de » multiple
  - Une interface définie des comportements sous forme de méthodes abstraites
  - Ces comportements doivent être redéfinis dans les classes qui implémentent l'interface
- Caractéristiques d'une interface
    - Comme une classe abstraite, n'a pas d'instance
    - Ne possède aucun champs
    - Une classe peut implémenter, autant d'interfaces que souhaitées
    - une interface ne peut `être-une-sorte-de` que d'interfaces
  - Syntaxe: `interface NomInterface {...}`
    `classe C implements NomInterface {...}`
- Règle: à la compilation, on vérifie que les classes implémenteuses ont bien les méthodes de l'interface
- Note:
  une interface peut avoir des variables dites d'interface (= variables de classes)

- C. Gestion des exceptions
  - le problème
    - gestion des erreurs systèmes (erreur de type, erreur d'héritage...)
    - gestion des erreurs prévues
    - l'informatique : la science des erreurs ?
    - temps réel: récupération dans tous les cas (selon T, ...)

  - Le principe
    - zone dangereuse : une partie de code où une exception peut avoir lieu
    - la levée d'exception : endroit de la zone dangereuse où est déclenchée l'exception qui fait sortir violemment de la zone dangereuse
    - traitement de l'exception : instructions à exécuter lors d'une levée d'exception, avant de poursuivre le programme

- En POO
  - Principes de la zone dangereuse, de la levée d'exception et du traitement de l'exception,
    - la levée de l'exception se fait par la création d'un objet qui est instance d'une **classe d'exception (`Throwable`)**, et qu'on envoie
    - Le traitement de l'exception dépend de la classe dont est issu l'objet qui l'a levée
  - Zone dangereuse
    - On pose une zone dangereuse avec le mot clef `try`, l'instruction (ou le bloc) qui suit peut contenir des levées d'exceptions,
      le traitement des exceptions suit cette instruction.

```
Try {opération_dangeureuse1, opération_dangeureuse2 ;}
  …
```

– Le traitement des exceptions se programme de la manière suivante: on précise pour chaque classe d'exception l'instruction (ou le bloc) à effectuer si une exception de cette classe a été levée; pour cela on fait suivre le mot clef `catch` du nom de la classe de l'exception et d'un nom de variable qui va contenir l'objet exception qui a été créé à l'appel. L'instruction à effectuer suit.

- `Catch(mon_exception_a_moi2 02) {Traitement2 ;}`

```
…
// o2 est l objet créé dans une opération dangereuse, il
est de la classe mon_exception_a_moi2, et peut donc être
référencé sous le nom o2 dans le traitement Traitement2
```

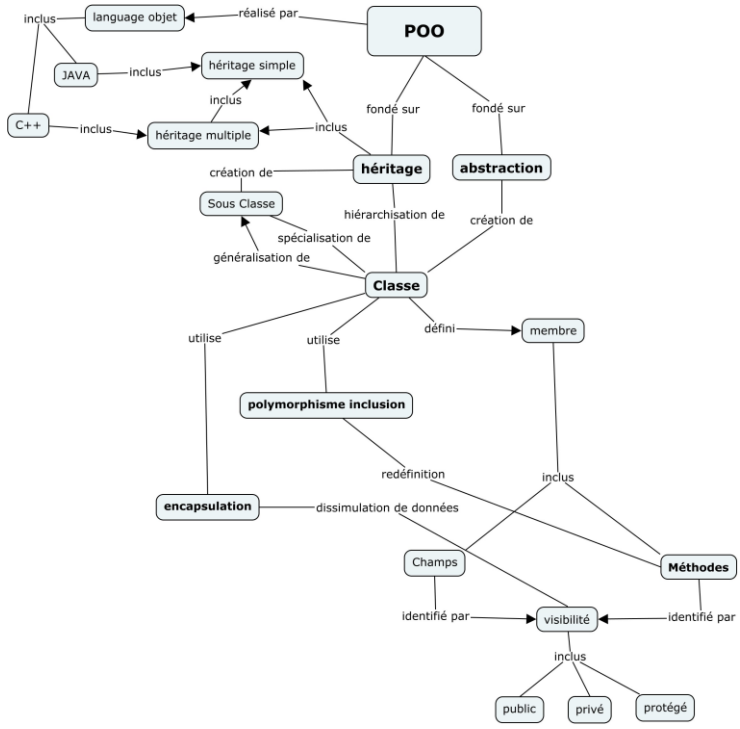
- On peut éventuellement ajouter à la fin du bloc d'exception, une instruction (ou un bloc) à exécuter dans tous les cas. Celle-ci est introduite par le mot clef `Finally`.

- `Finally {est-toujours-exécuté-si-present;}`

- Il faut évidemment avoir programmer les classes d'exception.

– Lever (et Création) d'exception

- Pour lever une exception (dans une opération dangereuse), on utilise le mot clef `throw` suivi de l'objet qui est l'exception. `throw new mon_exception_a_moi()`
- la levée d'exception est possible sur tout objet de type sous-classe de `Throwable`, dont deux classes fournies par JAVA sont `Exception` et `Error`
- Si le bloc de traitement d'exception n'est pas dans la même méthode que la levée d'exception, il faut associer à chaque déclaration de méthode que peut traverser l'exception, le nom de la classe d'exception avec le mot clef `throws` suivi du nom des classes d'exceptions qu'elle peut transmettre.
- En SmallTalk, après le traitement d'une exception, on peut si on le souhaite demander à reprendre l'instruction qui a levé l'exception, ou celle qui suit la demande de levée. Ex: `Play, Replay`


POO
réalisé par
language objet
inclus
JAVA
inclus
C++
inclus
héritage simple
héritage multiple
inclus
inclus
fondé sur
fondé sur
héritage
abstraction
création de
Sous Classe
hiérarchisation de
création de
spécialisation de
généralisation de
Classe
utilise
utilise
défini
membre
polymorphisme inclusion
redéfinition
inclus
encapsulation
dissimulation de données
Champs
Méthodes
identifié par
visibilité
identifié par
inclus
public
privé
protégé